Project 3: Synthesis of Sinusoidal Signals—Music Synthesis

You will write up a formal lab report in IEEE double-column format with figures integrated with the text. The exercises should written up in this week's lab report. You should **label** the axes of your plots, have a caption, and Figure number for every plot. Every plot should be referenced by Figure number in your text discussion.

This project will require the Professor listening to your audio in class either the class session (thursday) before the project is due. The quality of this response will be part of your project grade.

Forgeries and plagiarism are a violation of the honor code and will be referred to the Dean of Students for disciplinary action. You are allowed to discuss lab exercises with other students and you are allowed to consult old lab reports but the submitted work should be original and it should be your own work.

---

1 Introduction to Music Synthesis with Sinusoids

This lab includes a project on music synthesis with sinusoids. The piece *Für Elise* has been selected for doing the synthesis program. The project requires a listening test to judge the correctness of the synthesized song. The music synthesis will be done with sinusoidal waveforms of the form

$$x(t) = \sum_k A_k \cos(\omega_k t + \phi_k) \tag{1}$$

so it will be necessary to establish the connection between musical notes, their frequencies, and sinusoids.

2 Structure of sound waveforms

In this lab, the periodic waveforms and music signals will be created with the intention of playing them out through a speaker. Therefore, it is necessary to take into account the fact that a conversion is needed from the digital samples, which are numbers stored in the computer memory to the actual voltage waveform that will be amplified for the speakers.

2.1 Theory of Sampling

Chapter 4 treats sampling in detail, but we provide a quick summary of essential facts here. The idealized process of sampling a signal and the subsequent reconstruction of the signal from its samples is depicted in Fig. 1. This figure shows a continuous-time input signal $x(t)$, which is sampled by the continuous-to-discrete (C-to-D) converter to produce a sequence of samples $x[n] = x(nT_s)$, where n is the integer sample index and T_s is the sampling period. The sampling rate is $f_s = 1/T_s$ where the units are samples per second. As described in Chapter 4 of the text, the ideal discrete-to-continuous (D-to-C) converter takes the input samples and interpolates a smooth curve between them. The *Sampling Theorem* tells us that if the input signal $x(t)$ is a sum of sine waves, then

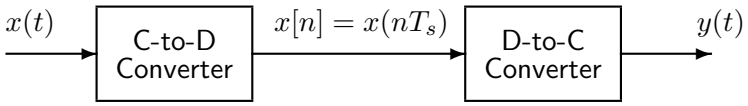

Figure 1: Sampling and reconstruction of a continuous-time signal.

the output $y(t)$ will be equal to the input $x(t)$ if the sampling rate is *more than twice the highest frequency* $f_{\max}$ in the input, i.e., $f_s > 2f_{\max}$. In other words, if we *sample fast enough* then there will be no problems synthesizing the continuous audio signals from $x[n]$.

2.2 D-to-A Conversion

Most computers have a built-in analog-to-digital (A-to-D) converter and a digital-to-analog (D-to-A) converter (usually on the sound card). These hardware systems are physical realizations of the idealized concepts of C-to-D and D-to-C converters respectively, but for purposes of this lab we will assume that the hardware A/D and D/A are perfect realizations.

The digital-to-analog conversion process has a number of aspects, but in its simplest form the only thing we need to worry about in this lab is that the time spacing (T_s) between the signal samples must correspond to the rate of the D-to-A hardware that is being used. From MATLAB, the sound output is done by the `soundsc(xx,fs)` function which does support a variable D-to-A sampling rate if the hardware on the machine has such capability. A convenient choice for the D-to-A conversion rate is 11025 samples per second,[1] so $T_s = 1/11025$ seconds; another common choice is 8000 samples/sec. Both of these rates satisfy the requirement of *sampling fast enough* as explained in the next section. In fact, most piano notes have relatively low frequencies, so an even lower sampling rate could be used. If you are using `soundsc()`, the vector `xx` will be scaled automatically for the D-to-A converter, but if you are using `sound.m`, you must scale the vector `xx` so that it lies between ± 1. Consult `help sound` or `help soundsc`.

- The ideal C-to-D converter is, in effect, being implemented whenever we take samples of a continuous-time formula, e.g., $x(t)$ at $t = t_n$. We do this in MATLAB by first making a vector of times, and then evaluating the formula for the continuous-time signal at t_n, i.e., $x[n] = x(nT_s)$ if $t_n = nT_s$.

  To begin, create a vector `x1` of samples of a sinusoidal signal with $A_1 = 100$, $\omega_1 = 2\pi(800)$, and $\phi_1 = -\pi/3$. Use a sampling rate of 2756.25 samples/second (equal to 44100/16), and compute a total number of samples (approximately) equivalent to a time duration of 0.5 seconds. You may find it helpful to recall that a MATLAB statement such as `tt=(0:0.01:3);` would create a vector of numbers from 0 through 3 with increments of 0.01. Therefore, it is only necessary to determine the time increment needed to obtain 2756.25 samples in one second. You should use the `sum_Cexp()` function from a previous lab for this part (with modification of the sampling rate).

  Use `soundsc()` to play the resulting vector through the D-to-A converter of the your computer, assuming that the hardware can support the $f_s = 2756.25$ Hz rate. Listen to the output.

- Now create another vector `x2` of samples of a second sinusoidal signal (0.8 secs. in duration) for the case $A_2 = 80$, $\omega_2 = 2\pi(1000)$, and $\phi_2 = +\pi/4$. Listen to the signal reconstructed from these samples. How does its sound compare to the signal in part (2.2)?

- ***Concatenate*** the two signals `x1` and `x2` from the previous parts and put a short duration of 0.1 seconds of silence in between. You should be able to concatenate by using a statement

[1] This sampling rate is one quarter of the rate (44,100 Hz) used in audio CD players.

something like:

```
xx = [ x1, zeros(1,N), x2 ];
```

assuming that both `x1` and `x2` are row vectors. Determine the correct value of `N` to make 0.1 seconds of silence. Listen to this new signal to verify that it is correct.

- To verify that the concatenation operation was done correctly in the previous part, make the following plot:

```
tt = (1/2756.25)*(1:length(xx));   plot( tt, xx );
```

This will plot a huge number of points, but it will show the "envelope" of the signal and verify that the amplitude changes from 100 to zero and then to 80 at the correct times. Notice that the time vector `tt` was created to have exactly the same length as the signal vector `xx`.

- Now send the vector `xx` to the D-to-A converter again, but change the sampling rate parameter in `soundsc(xx, fs)` to 11025 samples/second. *Do not recompute the samples in* `xx`, just tell the D-to-A converter that the sampling rate is 11025 samples/second. Describe how the *duration* and *pitch* of the signal were affected. Explain.

2.3 Structures in MATLAB

MATLAB can do structures. Structures are convenient for grouping information together. For example, run the following program which plots a sinusoid:

```
x.Amp = 7;
x.phase = -pi/2;
x.freq = 100;
x.fs = 2756.25;          %--- fs = 44100/16
x.timeInterval = 0:(1/x.fs):0.05;
x.values = x.Amp*cos(2*pi*(x.freq)*(x.timeInterval) + x.phase);
x.name = 'My Signal';
x            %---- echo the contents of the structure "x"
plot( x.timeInterval, x.values )
title( x.name )
```

Notice that the members of the structure can contain different types of variables: scalars, vectors or strings.

2.4 Debugging Skills

Testing and debugging code is a big part of any programming job, as you know if you've been staying up late on the first few labs. Almost any modern programming environment provides a *symbolic debugger* so that break-points can be set and variables examined in the middle of program execution. Nonetheless, many programmers still insist on using the old-fashioned method of inserting print statements in the middle of their code (or the MATLAB equivalent, leaving off a few semi-colons). This is akin to riding a tricycle to commute around Atlanta.

There are two ways to use debugging tools in MATLAB: via buttons in the edit window or via the command line. For help on the edit-window debugging features, access the menu `Help->Using the M-File Editor` which will pop up a browser window at the help page for editing and debugging. For a summary of the command-line debugging tools, try `help debug`. Here is part of what you'll see:

```
dbstop     - Set breakpoint.
dbclear    - Remove breakpoint.
dbcont     - Resume execution.
dbstack    - List who called whom.
dbstatus   - List all breakpoints.
dbstep     - Execute one or more lines.
dbtype     - List M-file with line numbers.
dbquit     - Quit debug mode.

When a breakpoint is hit, MATLAB goes into debug mode.  On the PC
and Macintosh the debugger window becomes active and on UNIX and VMS
the prompt changes to a K>.  Any MATLAB command is allowed at the
prompt.  To resume M-file function execution, use DBCONT or DBSTEP.
To exit from the debugger use DBQUIT.
```

One of the most useful modes of the debugger causes the program to jump into "debug mode" whenever an error occurs. This mode can be invoked by typing:

```
dbstop if error
```

With this mode active, you can snoop around inside a function and examine local variables that probably caused the error. You can also choose this option from the debugging menu in the MATLAB editor. It's sort of like an automatic call to 911 when you've gotten into an accident. Try `help dbstop` for more information.

Download the file `coscos.m` and use the debugger to find the error(s) in the function. Call the function with the test case: `[xn,tn] = coscos(2,3,20,1)`. Use the debugger to:

- Set a breakpoint to stop execution when an error occurs and jump into "Keyboard" mode,
- display the contents of important vectors while stopped,
- determine the size of all vectors by using either the `size()` function or the `whos` command.
- and, lastly, modify variables while in the "Keyboard" mode of the debugger.

```
function [xx,tt] = coscos( f1, f2, fs, dur )
% COSCOS   multiply two sinusoids
%
t1 = 0:(1/fs):dur;
t2 = 0:(1/f2):dur;
cos1 = cos(2*pi*f1*t1);
cos2 = cos(2*pi*f2*t2);
xx = cos1 .* cos2;
tt = t1;
```

2.5 Piano Keyboard

Section 4 of this lab will consist of synthesizing the notes of a well known piece of music.[2] Since these signals require sinusoidal tones to represent piano notes, a quick introduction to the layout of the piano keyboard is needed. On a piano, the keyboard is divided into octaves—the notes in one octave being twice the frequency of the notes in the next lower octave. The white keys in each octave are named A through G. In order to define the frequencies of all the keys, one key must be

[2]If you have little or no experience reading music, don't be intimidated. Only a little music knowledge is needed to carry out this lab. On the other hand, the experience of working in an application area where you must quickly acquire new knowledge is a valuable one. Many real-world engineering problems have this flavor, especially in signal processing which has such a broad applicability in diverse areas such as geophysics, medicine, radar, speech, etc.

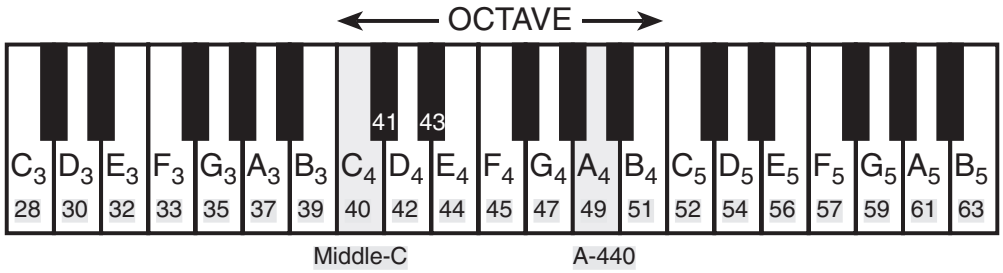

Figure 2: Layout of a piano keyboard. Key numbers are shaded. The notation C_4 means the C-key in the fourth octave.

designated as the reference. Usually, the reference note is the A above middle-C, called A-440 (or A_4) because its frequency is 440 Hz. (In this lab, we are using the number 40 to represent middle C. This is somewhat arbitrary; for instance, the Musical Instrument Digital Interface (MIDI) standard represents middle C with the number 60). Each octave contains 12 notes (5 black keys and 7 white) and the ratio between the frequencies of the notes is constant between successive notes. As a result, this ratio must be $2^{1/12}$. Since middle C is 9 keys below A-440, its frequency is approximately 261 Hz. Consult the text for even more details.

Musical notation shows which notes are to be played and their relative timing (half, quarter, or eighth). Figure 3 shows how the keys on the piano correspond to notes drawn in musical notation. The white keys are labeled as A, B, C, D, E, F, and G; but the black keys are denoted with "sharps" or "flats." A sharp such as $A^{\#}$ is one key number larger than A; a flat is one key lower, e.g., $A_4^{\flat}$ (A-flat) is key number 48.

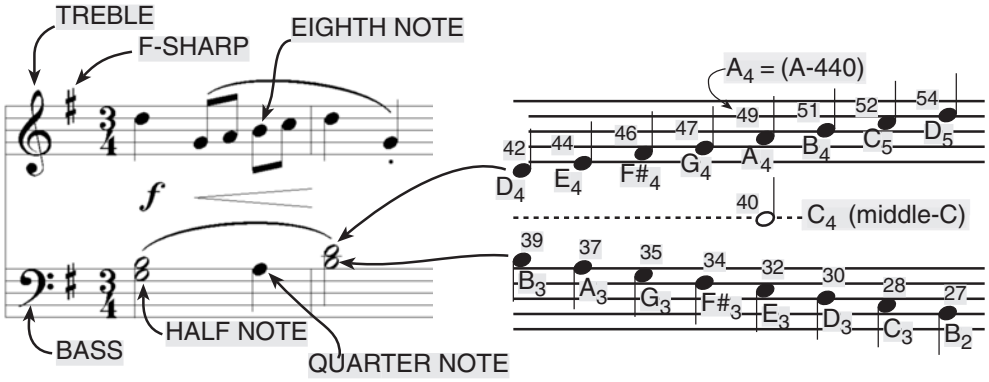

Figure 3: Musical notation is a time-frequency diagram where vertical position indicates which note is to be played. Notice that the shape of the note defines it as a half, quarter or eighth note, which in turn defines the duration of the sound.

Another interesting relationship is the ratio of fifths and fourths as used in a chord. Strictly speaking the fifth note should be 1.5 times the frequency of the base note. For middle-C the fifth is G, but the frequency of G is 391.99 Hz which is not exactly 1.5 times 261.63. It is very close, but the slight detuning introduced by the ratio $2^{1/12}$ gives a better sound to the piano overall. This innovation in tuning is called "equally-tempered" or "well-tempered" and was introduced in Germany in the 1760's and made famous by J. S. Bach in the "Well Tempered Clavier."

Thus, you can use the ratio $2^{1/12}$ to calculate the frequency of notes anywhere on the piano keyboard. For example, the E-flat above middle-C (black key number 43) is 6 keys below A-440, so its frequency should be $f_{43} = 440 \times 2^{-6/12} = 440\sqrt{2} \approx 311$ Hertz.

3 Warm-up

3.1 Note Frequency Function

Now write an M-file to produce a desired note for a given duration. Your M-file should be in the form of a function called `note_synth.m`. Your function should have the following form:

```
function xx = note_synth(X, keynum, dur)
% NOTE_SYNTH  Produce a sinusoidal waveform corresponding to a
%       given piano key number
%
%  usage:  xx = note_synth (X, keynum, dur)
%
%        xx = the output sinusoidal waveform
%         X = complex amplitude for the sinusoid, X = A*exp(j*phi).
%    keynum = the piano keyboard number of the desired note
%       dur = the duration (in seconds) of the output note
%
fs = 2756.25;         %--- fs = 44100/16    %-- or use 8000 Hz
tt = 0:(1/fs):dur;
freq =            %<=============== fill in this line
xx = real( X*exp(j*2*pi*freq*tt) );
```

For the `freq =` line, use the formulas given above to determine the frequency for a sinusoid in terms of its key number. You should start from a reference note (middle-C or A-440 is recommended) and solve for the frequency based on this reference. Notice that the `xx = real( )` line generates the actual sinusoid as the real part of a complex exponential at the proper frequency.

3.2 Synthesize an Arpeggio

In a previous section you completed the `note_synth.m` function which synthesizes the correct sinusoidal signal for a particular key number. Now, use that function to finish the incomplete M-file in Fig. 4 that will play successive notes in a chord (called an arpeggio). For the `tone =` line, generate the actual sinusoid for `keynum` by making a call to the function `note_synth()` written previously. It is important to point out that the code in `my_arpeggio.m` allocates a vector of zeros large enough to hold the entire arpeggio, and then **inserts** each note into its proper place in the vector `xx`.

3.2.1 Spectrogram of the Arpeggio

In this part, you must display the spectrogram of the arpeggio synthesized in the previous section. Remember that the spectrogram displays an image that shows the changing *frequency* content of a *time* signal. The horizontal axis in the spectrogram is time and its vertical axis is frequency.[3]

[3]The vertical axis is in Hz if the sampling rate has been given correctly as one of the arguments to the spectrogram M-file.

```
%--- my_arpeggio.m
%---
arpeggio.keys =   [ 45  49  52  57  52  49  45 ];
%------ NOTES:      F   A   C   F   C   A   F
%  key #49 is A-440
%
arpeggio.durs  = 0.33 * ones(1,length(arpeggio.keys));
fs = 2756.25;         %--- fs = 44100/16      %-- or 8000 Hz
xx = zeros(1,ceil(sum(arpeggio.durs)*fs)+length(arpeggio.keys) );
n1  = 1;
for kk = 1:length(arpeggio.keys)
   keynum = arpeggio.keys(kk);

   tone =                        %<============= FILL IN THIS LINE

   n2 = n1 + length(tone) - 1;
   xx(n1:n2) = xx(n1:n2) + tone;   %<=== Insert the note
   n1 = n2 + 1;
end
soundsc( xx, fs )
```

Figure 4: Prototype code for arpeggio synthesis function.

4 Lab: Synthesis of Musical Notes

The audible range of musical notes consists of well-defined frequencies assigned to each note in a musical score. Before starting the project, make sure that you have a working knowledge of the relationship between a musical score, key number and frequency. In the process of actually synthesizing the music, follow these steps:

- Use a sampling frequency of $f_s = 2756.25$ Hz to play out the sound through the D-to-A system of the computer. The sampling Theorem tells us that this rate is sufficient to represent music signals with frequency less than 1378 Hz. Recall that $T_s = 1/f_s$ is the time between samples of the sinusoids.

- Determine the total time duration needed for each note, and also determine the frequency (in hertz) for each note (see Fig. 2 and the discussion of the well-tempered scale in the warm-up.) A data file called `furElise.mat` will be provided with this information stored in MATLAB structures; this contains the portion of the piece needed for this lab. A second file called `furElise_short.mat` has the same information for the first few measures of the piece; you may find this useful for initial debugging. Both of these files are contained in a ZIP archive called `furElise.zip` which is linked from the lab page.

- Synthesize the waveform as a combination of concatenated sinusoids, and play it out through the computer's built-in speaker or headphones using `soundsc()`.

- Include a spectrogram image of a portion of your synthesized music—probably about 1 or 2 secs—so that you can illustrate the fact that you have all the different notes. The window length length might have to be adjusted, but start with an initial value of 512 for the window length in `specgram()`.

  In addition, the spectrogram M-files will scale the frequency axis to run from zero to half the sampling frequency, so it might be useful to "zoom in" on the region where the notes are. Consult `help zoom`, or use the zoom tool in MATLAB figure windows.

4.1 *Für Elise*

Für Elise is one of those pieces of classical music that everyone has heard. The first few measures are shown in Fig. 5, and all the measures that you must synthesize can be found on the class website. You must synthesize the entire portion of the *Für Elise* given in `furElise.mat` by using

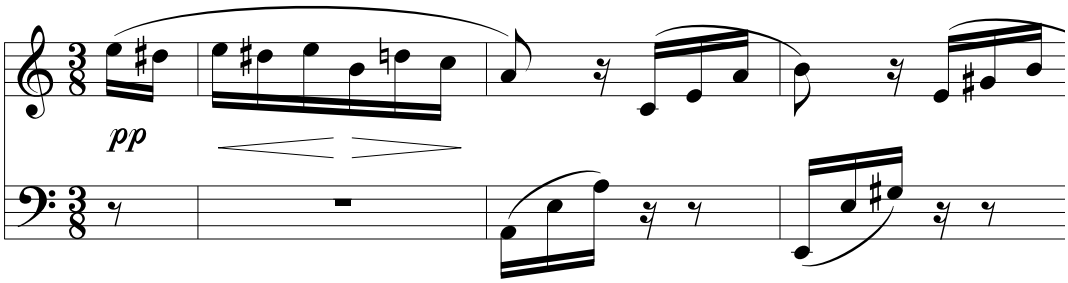

Figure 5: First few measures of the piece *Für Elise*.

sinusoids.[4]

4.2 Data File for Notes

Fortunately, a data file called `furElise.mat` has been provided with a transcription of the notes and information related to their durations. The data files `furElise.mat` and `furElise_short.mat` are contained in a ZIP archive called `furElise.zip` which is linked from the lab page. The format of a MAT file is not text; instead, it contains binary information that must be loaded into MATLAB. This is done with the `load` command, e.g.,

```
load furElise.mat
```

After the load command is executed two new variables will be present in the workspace, called `Treble` and `Bass`. Use `whos` at the command prompt to see that you have these new variables.

The variables `Treble` and `Bass` are structures whose fields are vectors. Each structure gives information about a single melody in the song; in music, such melodies are often called "voices." For example, `Treble` contains information about the treble notes (the notes with higher key numbers), while `Bass` contains information about the bass notes (the notes with lower key numbers). Some melodies contain only a few notes; they only add harmony at a few locations in the the song, but are otherwise silent. The maximum number of notes that will ever be played simultaneously during the song is two.

Each structure has four fields: `keyNum`, `keyDur`, `measureNum`, and `measurePct`. As an example, the `Bass` structure looks like

```
Bass.keyNum      =  [ # # # # ...  ]  % Key Number for Note
                                      %   -1 denotes a rest
Bass.keyDur      =  [ # # # # ...  ]  % Note Duration
Bass.measureNum  =  [ # # # # ...  ]  % Measure Number
Bass.measurePct  =  [ # # # # ...  ]  % Offset within a Measure
```

The value of `Bass.keyNum(j)` is a single note's key number. The note's duration is given in terms of musical notation: a quarter note is denoted as 0.25, an eighth note as 0.125, etc. *This duration is not in seconds.* The actual time duration of the notes in seconds will have to

[4] Use sinusoids sampled at 2756.25 samples/sec.

be determined once the tempo of the song is defined, e.g., your code should be written with a parameter that defines the time duration of a quarter note.

Measures and *beats* are the basic time intervals in a musical score. A *measure* is denoted in the score by a vertical line that cuts from the top to the bottom of one line in the score. For example, in Fig. 5 there are three such vertical lines dividing that part of the musical score into four measures. Each measure contains a fixed number of *beats* which, in this case, equals three. The label "**3/8**" at the left of Fig. 5 describes this relationship and is called the *time signature* of the song. By convention, "**3/8**" denotes "3/8 time," in which there are three beats per measure and a single beat is the length of one eighth note.

The data file specifies the location of each note by giving the the measure number (as an integer starting at 1) and the offset within the measure(as a percentage). For example, these are the fields for the bass:

`Bass.measureNum` and `Bass.measurePct`.

For example, typing `Treble.keyNum(12)` at the MATLAB command prompt returns the number 49, which describes the A-440 in the third measure. The note is a sixteenth note, so `Treble.keyDur(12)` equals 0.0625. The location of this A-440 as the last note in the third measure means `Treble.measureNum(12)` is 3, and `Treble.measurePct(9)` is 83.33333%, because its offset from the beginning is 5 out 6 possible sixteenth notes in the measure.

4.3 Timing

Musicians often think of the tempo, or speed of a song, in terms of "beats per minute" or BPM. You should write the code so that the BPM is a global parameter that can be changed easily. For example, you might let the BPM be defined with the statement:

```
bpm = 200;
```

If the tempo is defined only once, then it could be changed: for example, setting `bpm = 100` would make the whole piece play slower so it would take twice as long.

Computer programs that let musicians record, modify, and play back notes played on a keyboard or other electronic instrument are called "sequencers."[5] The timing resolution of a sequencer is usually measured in "pulses per quarter note," or PPQ. A real commercial sequencer would have a very high PPQ to encapsulate the subtle timing nuances of a real human playing a real instrument. The starting times of notes in the music file provided to you are not specified because the keys are just played in succession. Thus you have rather poor "timing resolution" because the notes have fixed durations.

Another timing issue is related to the fact that when a musical instrument is played, there will be gaps between successive notes. Therefore, inserting very short pauses between individual notes is necessary to replicate the correct musical sound because it imitates the natural transition that a musician must make from one note to the next.

[5]Popular commercial sequencers include Mark of the Unicorn's Digital Performer, Emagic's Logic Audio, Steinberg's Cubase and Opcode's Studio Vision.