ECES-352

Winter 2019

Lab #5: Synthesis of Sinusoidal Signals – Music Synthesis

(Lab Report Due **Beginning.of.Next.Lab**)

This is the official Lab #5 description; it is similar to the one in Appendix C.3 of the text, but the piece Fugue #5 for the Well-Tempered Clavier has been chosen for the synthesis. **Formal Lab Report:** You must write a formal lab report that describes your approach to music synthesis (Section 4).

Important: **When the lab instructs you to get an updated matlab file (like specgram.m), please refer to http://dspfirst.gatech.edu/matlab/toolbox/**

Introduction

This lab includes a project on music synthesis with sinusoids. The piece, Fugue #5 for the Well-Tempered Clavier by Bach has been selected for doing the synthesis program. The project requires extensive programming effort and should be documented with a lab report. A good report should include the following items: a cover sheet, commented MATLAB code, explanation of your approach, conclusions, and any additional tweaks that you implemented for synthesis. Since the project must be….

evaluated by listening to the quality of the synthesized song, the criteria for judging a good song are given at the end of this lab description. In addition, it may be convenient to place the final song on a web site so that it can be accessed remotely by a lab instructor who can then evaluate its quality. If you would like to try other songs, the *DSP First* CD-ROM includes information about alternative tunes: *Minuet in G*, *Für Elise*, *Beethoven's Fifth*, *Jesu, Joy of Man's Desiring* and *Twinkle, Twinkle, Little Star*.

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. A secondary objective of the lab is the challenge of trying to add other features to the synthesis in order to improve the subjective quality for listening. Students who take this challenge will be motivated to learn more about the spectral representation of signals—a topic that underlies this entire course.

2 Pre-Lab

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 this lab is usually done prior to lectures on sampling, so 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

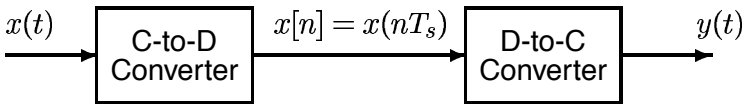

Figure 1: Sampling and reconstruction of a continuous-time signal.

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 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 at this point 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,[2] 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.`

(a) 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 the sample times, i.e., $x[n] = x(nT_s)$ if $t_n = nT_s$. This assumes perfect knowledge of the input signal, but we have already been doing it this way in previous labs.

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 11025 samples/second, and compute a total number of samples 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 11025 samples in one second. You should use the `syn_sin()` function from a previous lab for this part.

Use `soundsc()` to play the resulting vector through the D-to-A converter of the your computer, assuming that the hardware can support the $f_s = 11025$ Hz rate. Listen to the output.

(b) Now create another vector `x2` of samples of a second sinusoidal signal (0.8 secs. in duration) for the case $A_2 = 80$, $\omega_2 = 2\pi(1200)$, and $\phi_2 = +\pi/4$. Listen to the signal reconstructed from these samples. How does its sound compare to the signal in part (a)?

(c) **Concatenate** the two signals `x1` and `x2` with a short duration of 0.1 seconds of silence in between. You should be able to use a statement 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.

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

```
tt = (1/11025)*(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 singal vector `xx`.

(e) Now send the vector `xx` to the D-to-A converter again, but change the sampling rate parameter in `soundsc(xx, fs)` to 22050 samples/second. *Do not recompute the samples in* `xx`, just tell the D-to-A converter that the sampling rate is 22050 samples/second. Describe how the *duration* and *pitch* of the signal were affected. Explain.

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

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 = 11025
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 have 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. Of course, many programmers 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.

In order to learn how to use the MATLAB tools for debugging, 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:

1. Set a breakpoint to stop execution when an error occurs and jump into "Keyboard" mode,
2. display the contents of important vectors while stopped,
3. determine the size of all vectors by using either the `size()` function or the `whos` command.
4. 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.[3] Since these signals require sinusoidal tones to represent piano notes, a quick introduction to the frequency 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. For example, the reference note is the A above

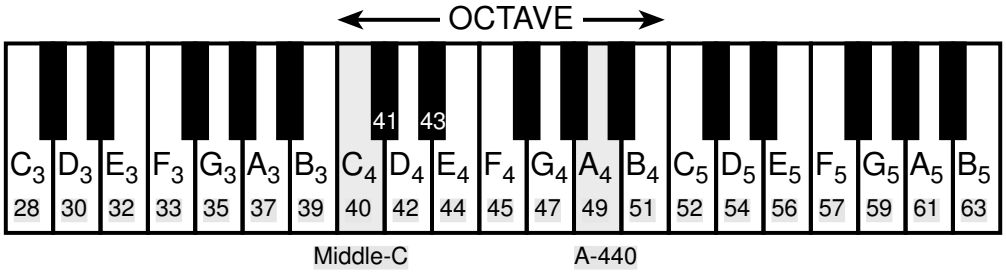

Figure 2: Layout of a piano keyboard. Key numbers are shaded. The notation C_4 means the C-key in the fourth octave.

middle-C which is usually 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 arbitary; 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 chapter 9 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 all 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., $A_4^{\flat}$ is key number 48.

[3] 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 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.

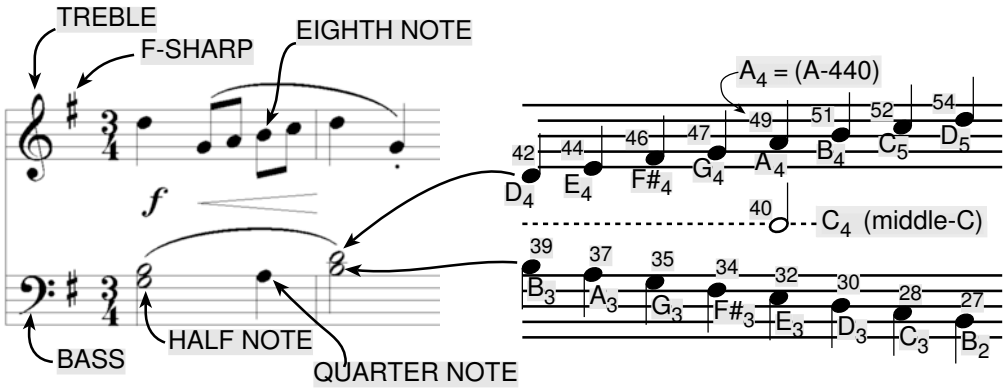

Figure 3: Musical notation is a time-frequency diagram where vertical position indicates which note is to be played. Notice that the shape of the note defines it as a half, quarter or eighth note, which in turn defines the duration of the sound.

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 about 392 Hz which is not exactly 1.5 times 261.6. 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," which is the source of the song for this lab.

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 `key2note.m`. Your function should have the following form:

```
function xx = key2note(X, keynum, dur)
% KEY2NOTE  Produce a sinusoidal waveform corresponding to a
%       given piano key number
%
%  usage:  xx = key2note (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 = 11025;     %-- 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.

Instructor Verification (separate page)

3.2 Synthesize a Scale

In a previous section you completed the `key2note.m` function which synthesizes the correct sinusoidal signal for a particular key number. Now, use that function to finish the following incomplete M-file that will play scales:

```
%--- play_scale.m
%---
scale.keys =   [ 40  42  44  45  47  49  51  52 ];
%--- NOTES: C   D   E   F   G   A   B   C
%  key #40 is middle-C
%
scale.durations  = 0.25 * ones(1,length(scale.keys));
fs  = 11025;                    %-- or 8000 Hz
xx  = zeros(1, sum(scale.durations)*fs+length(scale.keys) );
n1  = 1;
for kk = 1:length(scale.keys)
   keynum = scale.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 )
```

For the `tone =` line, generate the actual sinusoid for `keynum` by making a call to the function `key2note()` written previously. It is important to point out that the code in `play_scale.m` allocates a vector of zeros large enough to hold the entire scale then **inserts** each note into its proper place in the vector `xx`.

Instructor Verification (separate page)

3.3 Spectrogram: Two M-files

In this part, you must display the spectrogram of the scale synthesized in the previous section. Remember that the spectrogram displays an image that shows the *frequency* content of the synthesized *time* signal. Its horizontal axis is time and its vertical axis is frequency.

(a) Generate the signal for the scale with `play_scale.m`.

(b) Use the function `specgram(xx,512,fs).` Zoom in to see the progression of three consecutive notes in the scale (`help zoom`), and identify the note A-440 in your spectrogram. The second argument[4] is the *window length* which could be varied to get different looking spectrograms. The

[4]If the second argument is made equal to the "empty matrix" then its default value of 256 is used.

spectrogram is able to "see" the separate spectrum lines with a longer window length, e.g., 1024 or 2048.[5]

Instructor Verification (separate page)

(c) If you are working at home, you might not have the `specgram()` function because it is part of the "Signal Processing Toolbox." In that case, use the function `plotspec(xx,fs)` which can be downloaded from Web-CT (you also need to download `spectgr.m`).[6] Show that you get the same result as in part (b). Explain why the result is correct. If necessary, add a grid so that frequencies can be measured accurately.
• Note: The argument list for `plotspec()` has a different order from `specgram`, because `plotspec()` uses an optional third argument for the *window length* (default value is 256).

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. Five different pieces are given in the book, but we have chosen a different one for the synthesis program in this lab. 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:

(a) Determine a sampling frequency that will be used to play out the sound through the D-to-A system of the computer. This will dictate the time T_s between samples of the sinusoids.

(b) 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 `fall02_fugue.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 `fall02_fugue_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 `fall02_fugue.zip` which is linked from the lab page.

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

(d) Make a plot of a few periods of two or three of the sinusoids to illustrate that you have the correct frequency (or period) for each note.

(e) 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. This piece has many sixteenth notes, so a window length of 512 might be the best choice for `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-v5.3 figure windows.

[5] Usually the window length is chosen to be a power of two, because a special algorithm called the FFT is used in the computation. The fastest FFT programs are those where the signal length is a power of 2.

[6] Actually, you should download the ZIP file with all the new/updated M-files for ECE-2025.

4.1 Spectrogram of the Music

Musical notation describes how a song is composed of different frequencies and when they should be played. This representation can be considered to be a *time-frequency* representation of the signal that synthesizes the song. In MATLAB we can can compute a time-frequency representation from the signal itself. This is called the spectrogram, and its implementation with the MATLAB function `specgram()` or `plotspec()`. To aid your understanding of music and its connection to frequency content, a MATLAB GUI is available so that you can visualize the spectrogram along with musical notation. This GUI also has the capability to synthesize music from a list of notes, but these notes are given in "standard" musical notation, not key number. For more information, consult the `help` on `musicgui.m` which only runs in MATLAB version 5.

CD-ROM
MUSIC GUI

4.2 *Fugue #5 for the Well-Tempered Clavier*

Bach wrote a zillion pieces of music; this is one of the lesser known ones, but the overall sound is much like any other Bach piece. The first few measures are shown in Fig. 4. You must synthesize the entire portion of

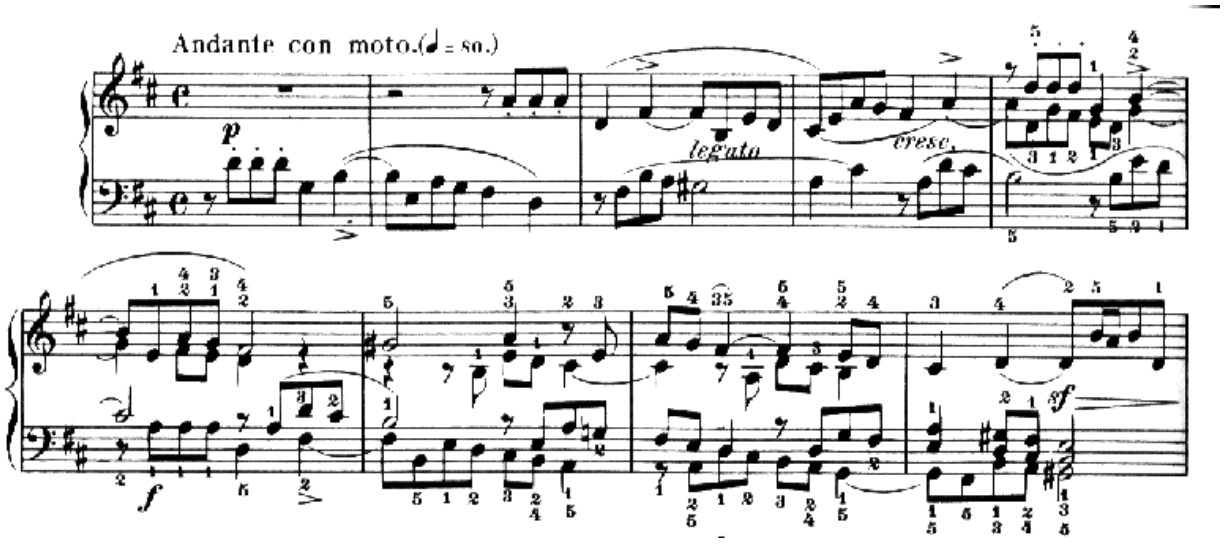

Figure 4: First few measures of the piece *Fugue #5 for the Well-Tempered Clavier*.

the *Fugue #5 for the Well-Tempered Clavier* given in fall02_fugue.mat by using sinusoids.[7]

4.3 Data File for Notes

Fortunately, a data file called `fall02_fugue.mat` has been provided with a transcription of the notes and information related to their durations. The data files `fall02_fugue.mat` and `fall02_fugue_short.mat` are contained in a ZIP archive called `fall02_fugue.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 fall02_fugue.mat
```

After the load command is executed a new variable will be present in the workspace, called `ourVoices`. Do `whos` to see that you have this new variable.

[7] Use sinusoids sampled at 11025 samples/sec (a lower sampling rate could be used if you have a computer with limited memory).

The variable `ourVoices` is a vector whose elements are structures. Each structure gives information about a single melody in the song; in fugues, such melodies are often called "voices." You can determine the number of melodies in the song by calculating the length of the vector `ourVoices` with the command `length(ourVoices)`. This number will also equal the maximum number of notes that are ever simultaneously played in the song.

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 the top line of Fig. 4 there are four 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 four — the number of quarter notes in a measure. The label "c" at the left of Fig. 4 describes this relationship and is called the *time signature* of the song. By convention, "c" denotes "common time," in which there are four beats per measure and that a single beat is the length of one quarter note. In our representation, there are four pulses per quarter note, so there are a total of 16 pulses per measure. (One sixteenth note corresponds to one pulse).

Each structure `ourVoices(i)` has three fields: `notes, startingTimes, and durations`. A typical structure `ourVoices(i)` looks like

```
ourVoices(i).notes          =  [ # # # # ...  ]  % Key number
ourVoices(i).startingTimes  =  [ # # # # ...  ]  % Starting pulse
ourVoices(i).durations      =  [ # # # # ...  ]  % Duration in pulses
```

The value of `voices(i).notes(j)` is a single note's key number. The note's starting pulse (where there are four pulses per quarter note, or 16 pulses per measure) and duration in pulses is given by the corresponding elements in the other two fields.

For example, typing `ourVoices(1).notes(4)` at the MATLAB command prompt returns the number 47, which describes the G in the first measure. Because the note is an eighth note and a eigth note is two pulses long, `ourVoices(1).durations(4)` equals two. The value of `ourVoices(1).startingTimes(4)` is 9 because this note begins eleven pulses from the beginning of the song.

4.3.1 Timing

Musicians often think of the tempo, or speed of a song, in terms of "beats per minute" or BPM, where the beats are usually quarter notes. 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 = 120;
```

Computer programs which lets musicians record, modify, an play back notes played on a keyboard or other electronic instrument are called "sequencer."[8] The timing resolution of a sequencer is usually measured in "pulses per quarter note," or PPQ. In this lab, we will employ four pulses per quarter note. A real commercial sequencer would have a much higher PPQ to encapsulate the subtle timing nuances of a real human playing a real instrument. The starting times and durations of notes in the music file provided to you are specified in terms of "pulses," so it will be helpful helpful to compute the number of "seconds per pulse," for instance via:

```
beats_per_second = bpm/60;
seconds_per_beat = 1/beats_per_second;
seconds_per_pulse = seconds_per_beat / 4;
```

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

If the tempo is defined only once, then it could be changed: for example, setting `bpm = 240` would make the whole piece play twice as fast.

Another timing issue is related to the fact that when a musical instrument is played, the notes are not continuous. Therefore, inserting very short pauses between notes usually improves the musical sound because it imitates the natural transition that a musician must make from one note to the next. An envelope (discussed below) can accomplish the same thing.

4.4 Musical Tweaks

The musical passage is likely to sound very artificial, because it is created from pure sinusoids. Therefore, you should try to improve the quality of the sound by incorporating some modifications. **(Note: in order to achieve the highest number of points on this lab you should implement at least one of the tweaks described in this section, or a similarly cool tweak of your own devising.)**

For example, one improvement comes from using an "envelope," where you multiply each pure tone signal by an envelope $E(t)$ so that it fades in and out.

$$x(t) = E(t)\cos(2\pi f_{\text{key}} t + \phi) \tag{2}$$

If an envelope is used it, should "fade in" quickly and fade out more slowly. An envelope such as a half-cycle of a sine wave $\sin(\pi t/\text{dur})$ is simple to program, but it sounds poor because it does not turn on quickly enough, so simultaneous notes of different durations no longer appear to begin at the same time. A standard way to define the envelope function is to divide $E(t)$ into four sections: attack (A), delay (D), sustain (S), and release (R). Together these are called ADSR. The attack is a quickly rising front edge, the delay is a small short-duration drop, the sustain is more or less constant and the release drops quickly back to zero. Figure 5 shows a linear approximation to the ADSR profile. Consult help on `linspace()` or `interp1()`

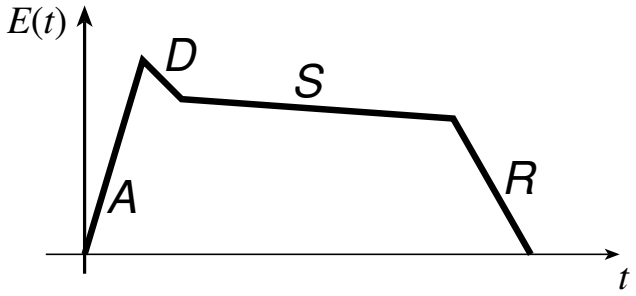

Figure 5: ADSR profile for an envelope function $E(t)$.

for functions that create linearly increasing and decreasing vectors.

Some other issues that affect the quality of your synthesis include relative timing of the notes, correct durations for tempo, rests (pauses) in the appropriate places, relative amplitudes to emphasize certain notes and make others soft, and harmonics. Since true piano sounds have a second and third harmonic content, and we have been studying harmonics, this modification is relatively simple, but be careful to make the amplitudes of the harmonics smaller than the fundamental frequency component.[9]

Furthermore, if you include too many higher harmonics, you might violate the sampling theorem and cause *aliasing*. You should experiment to see what sounds best.

[9] In the early 80's, a company called Digital Keyboards produced a commercial synthesizer called the Synergy in which the user created sounds via "additive synthesis" by specifying the envelops of individual frequency components. This is an quite powerful, albeit tedious and challenging way to create realistic sounds. American composer Wendy Carlos (best known for *Switched-On Bach* and her score for *A Clockwork Orange*) used it extensively in her score for *Tron*. See http://www.synthmuseum.com/synergy/synergy01.html

4.5 Programming Tips

You may want to modify your `key2note()` function to accept additional parameters describing amplitude, duration, etc. In addition, you might choose to add an envelope and/or harmonics. Chords are created on a computer by simply adding the signal vectors of several notes. Although we have provided a MATLAB file containing the note values and durations for *Fugue #5 for the Well-Tempered Clavier*, you are free to modify the duration values or add notes if you think it will improve the quality of the synthesized sound.

Lab #5 Verification Sheet

For each verification, be prepared to explain your answer and respond to other related questions that the lab TA's or professors might ask. Turn this page in at the end of your lab period.

Name: _____ Date of Lab: _____

Part 3.1 Complete and demonstrate the function `key2note.m`:

Verified:_____ Date/Time:_____

Part 3.2 Complete and demonstrate the script file `play_scale.m`:

Verified:_____ Date/Time:_____

Part 3.3 Demonstrate the spectrogram of the scale generated by `play_scale.m`:

Verified:_____ Date/Time:_____

Sound Evaluation Criteria

Does the file play notes? All Notes _____ Most _____ Treble only _____

Overall Impression: _____

Excellent: Enjoyable sound, good use of extra features such as harmonics, envelopes, etc.

Good: Bass and Treble clefs synthesized and in sync, few errors, one or two special features.

OK: Basic sinusoidal synthesis, including the bass, with only a few errors.

Poor: No bass notes, or treble and bass not synchronized, many wrong notes.