GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

**ECE 2025 Spring 2003**
**Lab #4: Synthesis of Sinusoidal Signals—Music Synthesis**

Date: 4–10-Feb-03

---

**FORMAL Lab Report:** You must write a formal lab report that describes your approach to music synthesis (Section 4). ***This lab report will be worth 150 points.***

**You should read the Pre-Lab section of the lab and do all the exercises in the Pre-Lab section before your assigned lab time.** You **MUST** complete the online Pre-Post-Lab exercise on Web-CT at the beginning of your scheduled lab session. You can use MATLAB and also consult your lab report or any notes you might have, but you cannot discuss the exercises with any other students. You will have approximately 20 minutes at the beginning of your lab session to complete the online Pre-Post-Lab exercise. The Pre-Post-Lab exercise for this lab includes some questions about concepts from the previous Lab report as well as questions on the Pre-Lab section of this lab.

The Warm-up section of each lab must be completed **during your assigned Lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time** by one of the laboratory instructors. After completing the warm-up section, turn in the verification sheet to your TA.

It is only necessary to turn in Section 4 as this week's lab report. More information on the lab report format can be found on Web-CT under the "Information" link. You should *label* the axes of your plots and include a title and Figure number for every plot. Every plot should be referenced by Figure number in your text discussion. In order to make it easy to find all the plots, include each plot *inlined* within your report. For more information on how to include figures and plots from MATLAB to your report file, consult the "Information" link on Web-CT. If you still do not know how to do so, ask your TA.

***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.***

The report will be **due during the period 11–17-Feb at the start of your lab.**

---

# 1 Introduction

This lab includes a project on music synthesis with sinusoids. The piece *Beethoven's 5th* has been selected for doing the synthesis program. The project requires an extensive programming effort and should be documented with a complete **formal** lab report.[1] A good report should include the following items: a cover sheet, commented MATLAB code, explanations of your approach, conclusions and any additional tweaks that you implemented for the 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.

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. Many of these additional features are modifications to the spectrum, so it is necessary to learn more about the spectral representation of signals—a topic that underlies this entire course.

[1] Refer to the ECE-2025 Web-CT page for more details on the required format.

# 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

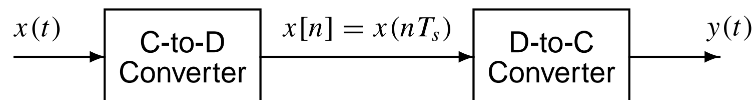



Figure 1: Sampling and reconstruction of a continuous-time signal.

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 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,[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 $\pm 1$. 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.

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

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 `makecos()` 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 $f_s = 11025$ 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` 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 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 signal 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.

## 2.3 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'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:

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 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 designated as the reference. Usually,

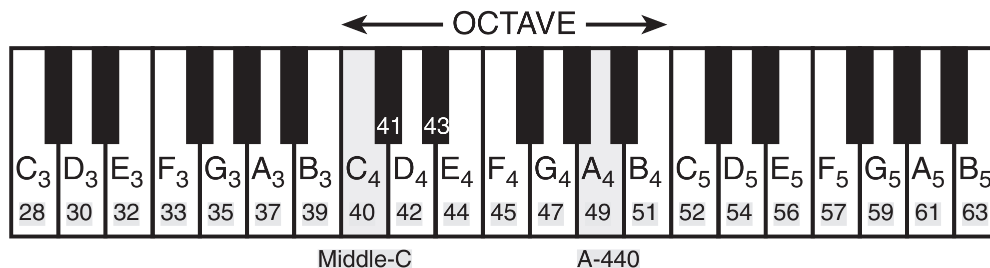



Figure 2: Layout of a piano keyboard. Key numbers are shaded. The notation $C_4$ means the C-key in the fourth octave.

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.

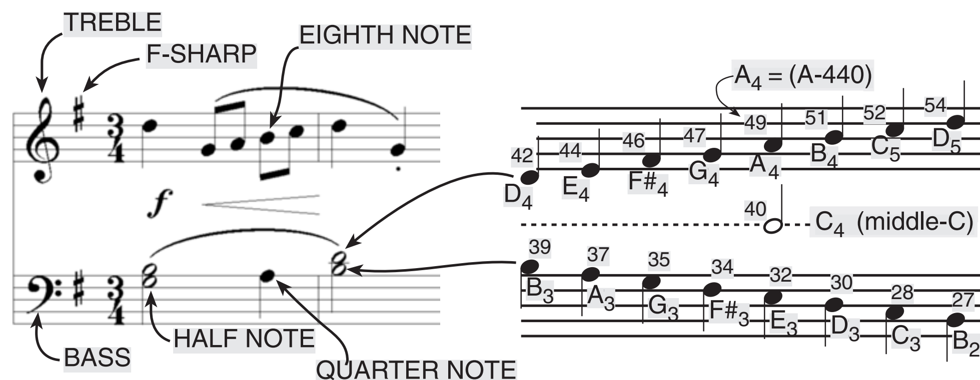



Figure 3: Musical notation is a time-frequency diagram where vertical position indicates which note is to be played. Notice that the shape of the note defines it as a half, quarter or eighth note, which in turn defines the duration of the sound.

Another interesting relationship is the ratio of fifths and fourths as used in a chord. Strictly speaking the

[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 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.

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 `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 an Arpeggio

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 incomplete M-file below 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 `key2note()` 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`.

**Instructor Verification** (separate page)

```
%--- 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  = 11025;                       %-- 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 )
```

### 3.3 Spectrogram: Two M-files

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 *frequency* content of the synthesized *time* signal. Its horizontal axis is time and its vertical axis is frequency.

(a) Generate the signal for the arpeggio with `my_arpeggio.m`.

(b) Use the function `specgram(xx,512,fs)`. Zoom in to see the progression of three consecutive notes in the arpeggio (`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 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 is part of the *SP-First* toolbox that can be downloaded from Web-CT. 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] If the second argument is made equal to the "empty matrix" then its default value of 256 is used.

[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.

# 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:

(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 `bee5th.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 `bee5th_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 `bee5th.zip` which is linked from the lab page.

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

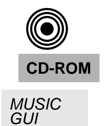
(d) Make a plot of a few periods of two or three of the sinusoids to illustrate that you have the correct frequency (or period) for each note. Measure the period from your plot and compare to the correct frequency.

(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. 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 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 obtain a time-frequency representation from the signal itself by making a spectrogram, 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 is part of the *SP-First* toolbox.




MUSIC GUI

## 4.2 *Beethoven's 5th*

*Beethoven's 5th* is one of those pieces of classical music that everyone has heard. The first few measures are shown in Fig. 4, and all the measures that you must synthesize can be found on the class website. You must synthesize the entire portion of the *Beethoven's 5th* given in `bee5th.mat` by using sinusoids.[6]

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

Figure 4: First few measures of the theme from Beethoven's Fifth.

### 4.3 Data File for Notes

Fortunately, a data file called `bee5th.mat` has been provided with a transcription of the notes and information about their location in the musical score. The data files `bee5th.mat` and `bee5th_short.mat` are contained in a ZIP archive called `bee5th.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 bee5th.mat
```

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

The variable `Bar` is a vector whose elements are structures. Each `Bar(k)` structure gives information about all the notes within one bar (or measure) of the piece. ***Measures*** and ***beats*** are the basic time intervals in a musical score. A ***measure*** is often called a ***bar*** and is denoted in the score by a vertical line that cuts from the top to the bottom of the treble and bass parts in the score. For example, in Fig. 4 there are six such vertical lines dividing that part of the musical score into seven measures, or bars. Each measure contains a fixed number of ***beats*** which, in this case, equals two. The label "**2/4**" at the left of Fig. 4 describes this relationship and is called the *time signature* of the song. By convention, "**2/4**" denotes "2/4 time," in which there are two beats per measure and that a single beat is the length of one quarter note.

For music synthesis, we need to subdivide the beats to represent individual notes. In this case, we notice that the shortest note is an eighth note and that a some measures contain 4 eighth notes, so we divide each (quarter note) beat into 2 equally spaced intervals. These intervals are called "pulses", so we have 4 pulses per measure. Therefore, each structure `Bar(k)` has one field called `pulse`, which in turn has two subfields which are vectors: `pulse.keynum(:)` and `pulse.amp(:)`. A typical structure `Bar(k)` looks like

```
Bar(k).pulse(1:4)          =  struct for each eighth note in measure
Bar(k).pulse(j).key(1:6)   =  vector with at most 6 key numbers
                                negative denotes start/end of long note
Bar(k).pulse(j).amp(1:6)   =  vector with amplitude for each key
                                negative denotes member of long note
```

The value of `[Bar(k).pulse(j).key(n)]` is a single note's key number; in this case, the $n^{\text{th}}$ note in the $j^{\text{th}}$ pulse of the $k^{\text{th}}$ bar. For example, typing `Bar(3).pulse(2).key` at the MATLAB command prompt should return the numbers 33 and 45, which describes the two $F$'s in the third measure. Because the third measure is to be played loud, the amplitude, `[Bar(3).pulse(2).amp(1)]` equals 4.0. We will adopt a convention that 1.0 represents a normal amplitude, with loud being 2.0 or 4.0 and soft being 0.5 or 0.25, etc. In general, the `[Bar(3).pulse(2).key]` and `[Bar(3).pulse(2).amp]` vectors have length 6, because there are a few places where 6 keys are played simultaneously

*Note:* You can determine the length of the song by calculating the length of the vector `Bar` with the command `length(Bar)`, and multiplying by the duration of one measure that contains two quarter notes.

### 4.3.1 Long Notes

One disadvantage of the [Bar.pulse.key] structure is that each note's duration is not given explicitly. Instead, a long note has to appear in several consecutive pulses. This presents the following problems:

1. When two consecutive, but distinct, eighth notes are played there should be a bit of silence in between.
2. When a longer note is played, it will be made up of sections whose duration is the same as an eighth note. However, it is crucial that these sections form an overall continuous waveform (i.e., no gaps or discontinuities allowed).

Thus we need a way to distinguish between these two situations. Negative key numbers and amplitudes will be used to denote the situation where a long note is being played. A long note contains many pulses, all of which have the same key number.

1. If [Bar(k).pulse(j).amp(n)] and [Bar(k).pulse(j).key(n)] are both negative, then the $n^{\text{th}}$ pulse is part of a long note, but is not the first or last pulse. The actual amplitude and key number are obtained by taking the absolute value.
2. If [Bar(k).pulse(j).amp(n)] is negative and [Bar(k).pulse(j).key(n)] is positive, then the $n^{\text{th}}$ pulse is the first pulse in a long note. Use `abs( )` to get the actual amplitude.
3. If [Bar(k).pulse(j).amp(n)] is positive and [Bar(k).pulse(j).key(n)] is negative, then the $n^{\text{th}}$ pulse is the last pulse in a long note. Use `abs( )` to get the actual key number.

Given this information about the notes, you should be able to synthesize the notes that are longer than one pulse. In addition, you should be able to implement and "attack" and "release" at the beginning and end of the long note.

### 4.3.2 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 = 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."[7] 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 specified in terms of eighth notes within the measure (two pulses per quarter note), so you have rather poor "timing resolution."

Another timing issue is related to the fact that when a musical instrument is played, repeated notes should not be continuous; whereas, held notes should be continuous. Therefore, inserting very short pauses between repeated notes (or 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. An envelope (discussed below) can accomplish the same thing.

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

## 4.4 Musical Tweaks

The musical passage is likely to sound very artificial, because it is created from pure sinusoids. Some 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. In your synthesis, you must try to improve the quality of the sound by incorporating the following two modifications.

### 4.4.1 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.[8] 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.

### 4.4.2 Envelope

Another 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

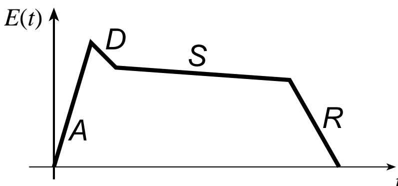



Figure 5: ADSR profile for an envelope function $E(t)$.

`interp1()` for functions that create linearly increasing and decreasing vectors. At a minimum, you must implement the attack and release profiles because these will give consecutive notes an audible beginning and end. Experiment with the duration for the attack a release, but make them rather small to simulate the quick change that would happen when a piano key is struck or released.

[8] 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

# Lab #4
# EE-2025
# Spring-2003
# Instructor 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 `my_arpeggio.m`:

Verified: ____________________ Date/Time: __________

Part 3.3 Demonstrate the spectrogram of the arpeggio generated by `my_arpeggio.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 and envelopes.

*Good:* All notes synthesized and in sync. Good sound quality. Use of one extra feature.

*OK:* Basic sinusoidal synthesis, with good quality of sound or treble only with very good sound quality.

*Poor:* Synthesis does not work. Poor sound quality.