Module 6: Generating Music with MATLAB

This module may appear disconnected from building a wireless communication system as we are building a basic "synthesizer" that is able to play back a simple melody. However, the components that are required to build such a synthesizer are essentially the same as the ones required when building a rudimentary wireless communication transmitter. ***Remember: As always, ask us in case you have questions!***

---

8 Generating Music with MATLAB

Before we can start building a MATLAB script that is able to play a simple melody, you need to learn a few more MATLAB programming concepts: (i) how to concatenate vectors to form a longer vector, (ii) conditional statements (also known as if-statements), (iii) how to use so-called for-loops, and (iv) how to write a wav-file to your hard-drive. After learning these concepts, we will guide you through the steps required to build a simple synthesizer. Finally, you can spend some time on improving your synthesizer and program it to create a melody of your choice.

8.1 A Few More MATLAB Concepts

Concatenating Vectors The first programming concept is how to take two vectors and combine them to create a longer vector. Create, for example, two row vectors

```
a = [1,2,3,4];
b = [5,6,7,8];
```

You can now use the command

```
c = [a,b]
```

where the new row vector `c` contains both row vectors `a` and `b` as follows:

```
>> c = [a,b]

c =

     1     2     3     4     5     6     7     8
```

Imagine you would like to create a vector `d` that contains the same vector (for example `a`) three times. This can be done with the following command:

```
d = [a,a,a]
```

where we have

```
>> d = [a,a,a]

d =

     1     2     3     4     1     2     3     4     1     2     3     4
```

Another, often useful approach to generate the same vector is as follows. First generate an *empty vector*

```
e = [ ]
```

This new vector `e` has zero entries (but is actually a proper MATLAB vector). One can now successively add vectors to this empty vector. Let us start to append the vector `a` with the following command

```
e = [ e , a ]
```

What this command does is to take the empty vector, concatenate it with the vector `a`, and overwrite the empty vector `e` with the new vector:

```
>> e = [e,a]

e =

     1     2     3     4
```

We can now repeat exactly the same command `e = [e,a]`, which appends the vector `a` to the new vector `e`, i.e., we have

```
>> e = [e,a]

e =

     1     2     3     4     1     2     3     4
```

This approach is useful when you want to successively build a long vector that contains multiple smaller vectors and you do not know beforehand how many vectors you will have and what type of vector. We will use it when adding different tones to from our synthesizer. In later modules, you will use that to generate the transmit signal of our wireless communication system.

If-Statements Conditional statements are a key component in programming languages. Imagine you want to write a MATLAB script that performs different tasks, depending on the contents of a variable. For example, assume that you have a variable called `bits` and you would like to produce a new variable `bots` that is $+1$ if `bits` is one and -1 if `bits` is any other value. Such conditional statements can be implemented using MATLAB's `if` and `else` statements. For example, the commands sequence

```
if bits==1
  bots = 1;
else
  bots = -1;
end
```

would implement exactly this. The first if-statement checks whether `bits` is equal to one. Note that we are using `==` to check for equality (a single equal sign is used to assign values). The statement right below is is executed only if the condition applies (if `bits==1` is true). The statement after `else` will be executed whenever the original condition `bits==1` is not satisfied. Note that other conditions also exist in MATLAB. For example, `bits~=1` would check whether the variable `bits` is *not* euqal to 1. The statement `bits>1` checks whether `bits` is larger than one; the statement `bits>=1` checks whether `bits` is larger or equal to one. MATLAB also supports the expressions `bits<1` and `bits<=1`. Note that we can also compare the contents of two variables, where one would simply replace the constant 1 with another variable.

For-Loops One of the key purposes of programming languages is to automate certain tasks. Imagine you would like to repeat a MATLAB command 10 times. A straightforward way would be to write a MATLAB script that contains the same command 10 times. If you want to build a script that can vary the number of times to repeat the same command, then you are in trouble with this approach. Also, if the number of repetitions increases, then your script would grow in length and would become almost impossible to work with. Fortunately, most programming languages offer a construct that simplifies a repeated execution of commands. In MATLAB, a common construct to repeat a set of commands are so-called for-loops. Let us explain the basic concept using a simple example. Execute the following line in MATLAB:

```
for kk=1:5; disp(kk); end
```

You should see the following output:

```
>> for kk=1:5; disp(kk); end
     1

     2

     3

     4

     5
```

Here, `disp(kk)` prints the contents of the variable `kk`. But let us look at the rest of this command. First, remember what `kk=1:5` does; if executed alone, it would create a row vector containing the numbers $\{1,2,3,4,5\}$. The for-loop simply repeats the statement (or statements) between the first semicolon and the `end` statement for each entry in the list $\{1,2,3,4,5\}$; in this case five times. Furthermore, in each of these repetitions, the variable `kk` assumes one of the values in $\{1,2,3,4,5\}$ starting from 1. In the above example, in the first repetition `kk=1` and we display 1, in the second repetition `kk=2` and we display 2, etc. Hence, for-loops can be used to repeatedly execute MATLAB commands while having control over the number of repetitions (equal to the length of `kk`. Furthermore, in each repetition, we have access to exactly one element from that vector using the variable `kk`.

Activity 23: Create a more complex for-loop

Let us look at a slightly more elaborate example which requires us to generate a MATLAB script. Generate a new MATlAB script that contains the following statements:

```
a = [];
```

```
N = 4;
for kk=1:N;
    a(kk) = kk*2;
end
```

Run the script and look at the generated vector `a`. Can you explain what happened?

Now, modify your MATLAB script so that the for-loop creates a new vector `a` that contains all squares from $\{1, 2, \ldots, 10\}$, i.e., the output of your for-loop should be

```
a =

     1     4     9    16    25    36    49    64    81   100
```

Please talk to us if you have difficulties with the concept of for-loops. For our music synthesizer, we will use for-loops to cycle through a list of frequencies to be played back and to generate a long vectors containing sinusoids at these frequencies.

Saving Your Own wav-Files The folder `examples` contains a number of pre-generated wav-files. However, you may want to save a signal that you generated or recorded to the hard-disk. Fortunately, we prepared a command for this. Assume that you have a vector containing samples called `y` recorded at a sampling rate of `FS`. If you want to save this signal, simply execute the following command:

```
save_audio('test.wav',y,FS)
```

where the statement `'test.wav'` could be replaced by a variable (e.g., `filename`) containing the file name of your choice. This command writes a wav-file to the disk. Note that you can play this wav-file with your favorite audio player (even with iTunes) or you can use the `load_audio` command to load the wav-file back into your workspace.

8.2 Building a Simple Music Synthesizer

Our goal is to write a MATLAB script that plays a predefined set of notes. On a high level, our script will work as follows. We define a vector `note` that contains the notes we want to play and a vector `secs` that tells us how long (how many seconds) we want to play each of these notes. The script then iterates through both vectors, reads out one note at a time note, converts it into a frequency, and generates a sine wave at that frequency for the given duration—here is where you need the for-loop. This sine wave is then appended to a long vector that contains all samples of the entire tune—here is where you need to know how to append vectors. Finally, you can play back the tune. We next explain the script step-by-step.

Create a new MATLAB script called `test_play_music.m`. First, we generate a vector that contains the frequencies associated with the notes ranging from C4 to C6, i.e., spanning two octaves of a piano. We can enumerate all notes of a grand piano from $n = 1, 2 \ldots, 88$. For example, note A4 is key 49 and corresponds to a frequency of 440 Hz. Fortunately, there exists a formula that maps note index n to frequencies:

$$f(n) = \left(\sqrt[12]{2}\right)^{n-49} * 440\,\text{Hz}. \tag{5}$$

Add the following lines to your MATLAB script:

```
% generate frequencies of notes contained in the vector frequency_list
% index : 1  2   3  4   5  6  7   8  9   10 11  12
% note  : C4 C#4 D4 D#4 E4 F4 F#4 G4 G#4 A4 A#4 B4
```

```
% index : 13 14  15 16  17 18 19  20 21  22 23  24 25
% note  : C5 C#5 D5 D#5 E5 F5 F#5 G5 G#5 A5 A#5 B5 C6
nn = 40:64;
frequency_list = 2.^((nn-49)/12)*440;
```

This piece of code generates a vector called `frequency_list` with 25 entries. As shown in the code comments above, the first entry of this vector, i.e., `frequency_list(1)`, contains the frequency of the note C4; similarly, index 10 contains the frequency of note A4, i.e., `frequency_list(10)` contains 440. We will use this vector to map indices to frequencies which simplifies programming of our synthesizer.

Second, we create two vectors that define the tune. The first vector contains the notes we want to play and the second vector contains their duration. Add the following lines to your MATLAB script:

```
% define a simple tune
note = [5,3,1];
secs = [2,1,2]/4;
```

The vector `note` contains three notes. If you check the mapping shown in the code snipped above, you can see that 5 maps to E4, 3 maps to D4, and 1 maps to C4. Hence, we will play the sequence: E4, D4, and C4. The vector `secs` must have the same number of entries as the vector `note`. Each entry indicates how long we want to play each note. In the above example, we will play E4 for 2/4 seconds, D4 for 1/4 second, and C4 for 2/4 seconds. Note that dividing a vector by 4 divides each entry individually by this number.

Third, we create a for-loop that takes the information contained in the vectors `note` and `secs`, creates a sampled sine wave of the given frequency and the given length, and appends it to a vector that contains all samples of the tune. Add the following lines to your MATLAB script:

```
y = [];
for kk=1:length(note)
    frequency = frequency_list(note(kk));
    samples = linspace(0,secs(kk),FS*secs(kk));
    sound = sin(2*pi*frequency*samples);
    y = [y , sound , zeros(1,400)];
end
```

This is the heart of our synthesizer. The first line defines an empty vector `y`; this vector will (after the for-loop is executed) contain all samples of our synthesized sound. The for-loop repeats four MATLAB commands for the length of the defined tune (the length of the vector `note`). For each index `kk`, we first create a variable called `frequency`, which takes index `kk`, reads out the corresponding note `note(kk)`, and selects the actual frequency associated with that tone index from the vector `frequency_list`. We then create a vector `samples`, which contains a set of samples from $t = 0$ to $t =$ `secs(kk)` under the assumption that the sampling rate is `FS=44100`. The next line creates a sampled version of a sine wave at that given frequency and stores the result in the vector `sound`. Finally, we concatenate this set of samples to the vector `y`. In order to have short pauses between each note, we also insert 400 zeros by using the command `zeros(1,400)`. (This command creates an all-zero row-vector with 400 entries.).

Fourth, add the following two lines to your MATLAB script:

```
% play the signal
play_audio(y,FS,OutID)
% save audio signal
save_audio('test.wav',y,FS);
```

These commands simply play your tune and write the tune as a wav-file to your disk.

You have now a fully functional, yet rudimentary, synthesizer that is able to play simple tunes. Try to run your script. You should hear three notes as defined in the vectors `note` and `secs`. If the script does not work, try to see where the error is—in case you cannot find it, feel free to let us know.

Activity 24: Create another tune and include amplitude modulation

Modify the vectors `note` and `secs` to create a more elaborate tune. In case you get bored by hearing a sine wave, you can use the following command instead

```
sound = sign(sin(2*pi*frequency*samples));
```

which creates a so-called square wave (a signal that switches between -1 and $+1$ at a given frequency). Square waves were used in the sound-chip of the Nintendo Entertainment System (NES). That is also why playing square waves sounds a bit like old-school gaming consoles.

Also, add another vector called `amps` that defines the amplitude of each note. Extend the MATLAB script so that you can change the amplitudes for each tone you play.

Remember: As you will see, building a rudimentary wireless transmitter will use a similar set of steps as this simple synthesizer. You need information to transmit (which is here the vector containing the notes, for example). You will need a for-loop that takes the information and maps it onto waveforms (this is called modulation). And then, you need to transmit the signal (which is done by playing the audio signal). So basically you already have the skeleton of a wireless transmitter. Just that in the above example, we were transmitting a melody to your ears and not data from a transmitter to a receiver.

Activity 25: Create a polyphonic tune

The synthesizer from the previous activities was monophonic, i.e., you could only play a single note at a time. Make your synthesizer polyphonic, which means that you can play more than one tone at a time. This allows you to play actual chords. As it turns out, you can simply add two signals if you want to hear them at the same time, but you have to be careful that the resulting signal does not exceed values in the range $[-1, +1]$. Create a polyphonic tune and save it as a wave file—then share the wave file with us. ***We want to hear your music!***

Optional Activity 26: Change the waveform to sound like a piano

If you are making quick progress or if you are interested in learning how modern keyboards, synthesizers, or electric pianos create artificial instruments, try the following. Load the C4 piano sample `piano-C4.wav` and try to replace the sine wave with that sample. You can change the frequency of this sample by reading out the indices at different rates; the command `linspace` is extremely helpful for this task. Be careful that you can only read out values from a vector at integer values, so you will also need the rounding function `round`. Note that this activity is not as easy as it may seem, but we can help to make it work.