ECE 72
Fall 2018
Project 3

(A) Slide Show

You will prepare a MATLAB program that generates a slide show. Your program will also create a video from the slide show. The video will show the slides, one after the other, from beginning to end. The video will be stored in an MPEG-4 file (having a filename of the form **`*.mp4`**). You will also prepare a document that explains the story of your video.

Each student will submit his/her own work. You may discuss this project with other students. But your submissions must be the fruit of your own labor.

Instructions

Your program will consist of a script m-file and at least one function m-file. These files will be written by you.

Theme

Your first objective is to identify a theme for the slide show. Each slide in the show should be consistent with the theme, yet different from every other slide in ways that can be defined by a small number of parameters.

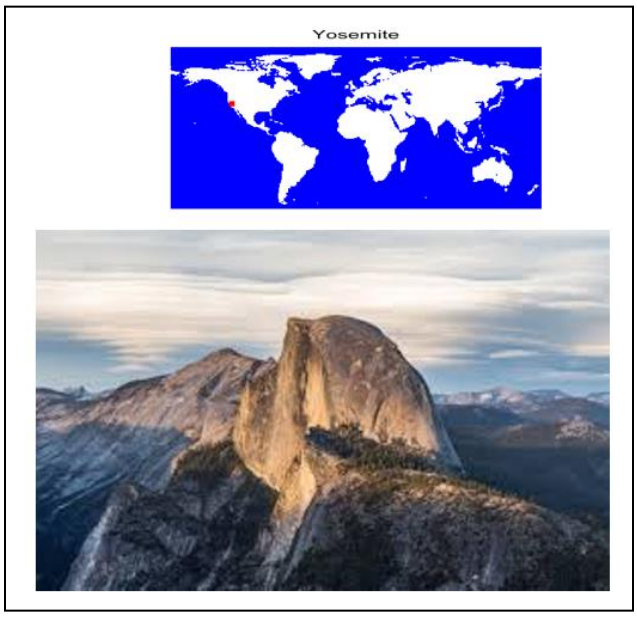
An example slide show, called **`parks`**, will be made available to you. This example slide show has a national parks theme. Each slide is devoted to one national park, and each slide contains the name of the park, a world map indicating the location of the park, and an image of the park. The first slide of **`parks`** appears on the next page. The slide show **`parks`** was created in accordance with the instructions of this document. Four parameters define each slide: the name of the park, the longitude of the park, the latitude of the park, and the filename of a graphics file containing a picture of the park. For the first slide, these parameters have the values:

```
'Yosemite',-119.54,37.87,'Yosemite.jpg'.
```

The ways that every other slide differs from this first slide can be characterized by the values of these four parameters.

You will *not* prepare a slide show that looks like **`parks`**. You will create a slide show of your own design. The example **`parks`** is made available to you in order to show how a small set of parameters can define each slide within a slide show. Use your imagination in selecting a theme,

subject only to the constraints of the school's Academic Policy Manual. Your slide show need not feature maps and need not include images.



First slide in **`parks`** slide show

Script

Your script will identify the filename of the file (the "video file") that is to contain your video. Your script will also define an object of class **`VideoWriter`** that will store properties of the video. It is not important that you understand *exactly* what "object of class **`VideoWriter`**" means. This document will explain as much as you will need to know about objects in MATLAB.

The video file will be located in the file system of your computer, in non-volatile memory. The **`VideoWriter`** object, on the other hand, will be in the MATLAB Workspace. This object will contain properties of the video, including the filename of the video file and the frame rate (the number of slides per second). The **`VideoWriter`** object will not, itself, contain the video. The **`VideoWriter`** object will know how to find the video file within the file system, so that another slide can be added to the video. If you clear the MATLAB Workspace, the **`VideoWriter`** object will disappear but the video file will remain in non-volatile memory. You will be able to run the video outside of MATLAB by double-clicking the icon for the video file.

On the next page is a summary of the script that you will need to make this video. The variable **`myVideo`** will store a string that names the video file (but without the extension because MATLAB will automatically add that to the end of your string). The variable **`sps`** will contain the frame rate (the number of slides per second). Of course, you are free to select other names for these two variables.

Assign string to **myVideo** *(filename, to which* **.mp4** *will be added by* MATLAB*).*
Assign value to **sps** *(slides per second).*

```
writerObj = VideoWriter(myVideo,'MPEG-4');
writerObj.FrameRate = sps;
open(writerObj);
```

Create a cell array containing slide parameters.

Repeat the following, once per slide:
Call the custom function that creates a color image **(RGB)** *for each slide*
`writeVideo(writerObj, RGB);`

```
writeVideo(writerObj, RGB); % dummy slide
close(writerObj);
```

Define the variables **myVideo** and **sps**. The first will be a string that names the video file (not including the suffix **.mp4**, which will be supplied by the program). The second is the number of frames per second.

writerObj = VideoWriter(myVideo,'MPEG-4');
The function **VideoWriter** creates an object of class **VideoWriter** whose name, in this instance, is **writerObj**. This object stores, within the MATLAB Workspace, properties of the video. The video will be saved in non-volatile memory in an MPEG-4 file. If the variable **myVideo** has the value **'parks'**, then the video is stored in the file **parks.mp4**. One of the properties stored in **writerObj** is the filename of the video file.

writerObj.FrameRate = sps;
Because of the dot (***.***) notation here, you might think that this line of code references a structure and its field. But it actually references an object and its property. An object is more complicated than a structure, and the two should not be confused. (In a previous line of code, **writerObj** was created as an object of class **VideoWriter**, so MATLAB will not be confused about the nature of **writerObj**.) With objects, the dot notation references a property of the object. In this case, the property **FrameRate** of the object **writerObj** is assigned the value of the variable **sps**. If **sps** has previously been assigned the value **0.5**, then the **FrameRate** property of **writerObj** becomes **0.5**.

open(writerObj);
This opens the video file for writing video data.

Create a two-dimensional cell array that will contain the parameters that define the slides. You should use the built-in **`cell`** function to establish the size of this cell array. The number of rows in the cell array should equal the number of unique slides. (There will also be a dummy slide, but we won't count that as one of the "unique" slides. The reason for having a dummy slide is discussed later in this document.) The number of columns should equal the number of parameters required to characterize each slide. Once the size of the cell array is established, there must be a set of assignment statements, one per unique slide, that places parameter values into the cells.

A repetition loop, such as a **`for`**-loop, should be used. Each time through the loop, two actions are required. First, a custom function should be called that creates (and returns) a new slide in the form of a color image (a three-dimensional **`uint8`** array with 3 pages). The appropriate parameter values (stored in the cell array) for the current slide must be passed to this slide-making function. Second, the new slide (the color image) should be added to the slide show, thus lengthening the slide show by one slide. The number of passes through the loop will equal the number of unique slides.

It might occur to you to omit the cell array and the repetition loop. You might think that in your script you can have, for each slide, a pair of statements: one that creates a slide by calling the slide-making function and a second that adds that slide to the slide show. There would be one such pair of statements for each unique slide that is to go into the slide show. It is true that this scheme would work. But if you do this, your instructor will be displeased! Why? The resulting code would be an example of *poor* programming style.

In striving for *good* programming style, we want our code to highlight the patterns in our program so that the architecture of the program is evident. By using the repetition loop, as you are advised to do here, your code will emphasize the fact that each slide is handled in the same way as ever other, that the differences from one slide to the next are neatly summarized in the input variables (the parameters) passed to the slide-making function.

`writeVideo(writerObj, RGB);`
This adds the color image **`RGB`** (a three-dimensional **`uint8`** array having 3 pages) to the video.

Outside and immediately following the repetition loop, you should again add the last unique slide to the slide show. We call this final slide a dummy slide. The reason for adding a dummy slide is discussed later in this document.

`close(writerObj);`
This function closes the video file after writing video data.

Your program will store the video in an MPEG-4 file (having a filename of the form **`*.mp4`**). This type of file contains, in general, compressed visual and audio information. However, for this project, you are only asked to produce a silent video. No audio is required.

Slide-Making Function

Your script will call a slide-making function of your own design. Your script will pass that function a set of input parameters. The differences that exist, one slide from the next, will be defined by these inputs. The output of the slide-making function will be a color image (that is, a three-dimensional array having 3 pages).

For example, the slide-making function that was used in the creation of **`parks`** has the following four input parameters: a string containing the park name, the longitude (a signed number), the latitude (a signed number), and a string containing the filename of the picture to be inserted into the slide.

It is essential that every image (three-dimensional array) returned by the slide-making function has the same size. If this is not the case, an error will occur. You can choose the size of the image, but every image returned by the slide-making function must have that same size.

You might want to create a second custom function to assist your slide-making function. This second function can be called by your slide-making function. This is generally a good idea if the construction of a slide is complicated. Spreading the work load over more than one function is good programming practice.

Your slide-making function should return an image. But it is generally easier to design a slide in a figure. Therefore, it is strongly recommended that you design each slide as a figure and then convert the figure to an image.

Figure and Image

You need to understand the difference between a color image and a figure. A color image specifies the color of every pixel in an image using a three-dimensional **`uint8`** array. Color images are useful for the following reasons. A graphics file can easily be read into a color image (in the MATLAB Workspace) using the function **`imread`**. We can display a color image using the function **`imshow`** or the function **`image`**. We can copy rectangular groups of pixels with a single instruction. (We can also copy more complicated shapes, but that requires more effort, as we must break up the shape to be copied into rectangular components.) But beyond these simple processes, it is typically difficult to design images by manipulating their three-dimensional **`uint8`** arrays.

For example, it is difficult to print text in a color image by manipulating its three-dimensional **`uint8`** array.

We can do a lot of interesting design with relative ease when working with a figure. For example, after placing a color image in a figure, we can add a title by simply using the **`title`** function. It is also easy to add curves and geometric shapes to a figure. To make geometric shapes in a figure, for example, the function **`fill`** is quite useful.

Typically, a figure comes with a set of axes by default. However, if we don't want the axes, we can easily delete them using the following command:

```
axis off
```

A figure can be converted to an image in a two-stage process. First, the figure is converted into a frame, using the built-in function **`getframe`**. (A frame is a special MATLAB structure that captures a snapshot of a figure.) Second, the frame is converted into a color image, using the built-in function **`frame2im`**. This is how it is done in one line of code for a figure having the handle **`1`**:

```
RGB = frame2im(getframe(1));
```

Last (Dummy) Slide

If you name the variable holding the frame rate **`sps`**, then the time (in seconds) between slides of your video will be 1/**`sps`**. However, when you build your video, the show will end *immediately* after displaying the last slide. This will happen so quickly that it will seem like the last slide was dropped from the show. It is recommended that you add an extra slide to your slide show; this last, *dummy* slide will seem to get dropped when you view the slide show, but that will be okay because it will never have been your intention for this last slide to appear in the show. The purpose of that last (dummy) slide is just to make the next-to-last slide (a *real* slide) stay on screen for a duration of 1/**`sps`** before the show ends. For this dummy slide you can use almost anything. For example, you could add the last real slide onto the end of the slide show again, so that the first copy of it stays on the screen for a period 1/**`sps`**, then the second copy appears and disappears almost instantly.

Running the Program

When your script and function (or functions) are completed, you can execute your script from within MATLAB. This will create an **`*.mp4`** file (if you haven't made any mistakes). You can then run the video outside of MATLAB by double-clicking the icon for your newly created **`*.mp4`** file.

Publishing Your Script

As explained above, you will create one script m-file and at least one function m-file for this project. In the script, you will include a **`type`** command for each custom function that you created for this project. Each **`type`** command will be of the form:

`type` ***`functionname`***

where ***`functionname`*** is the name of a function.

You will publish your script to a **`*.pdf`** document. The **`type`** command in the script causes the code in the named function to be printed to the published **`*.pdf`** document.

What You Will Submit

After you run your script to produce a **`*.mp4`** video file, you will publish your script to a **`*.pdf`** document. Make sure that the code for your custom function or functions appear along with the code for your script in your published **`*.pdf`** document. You will submit to Canvas your **`*.mp4`** video file and your published **`*.pdf`** document.

You will also submit to Canvas an essay (preferably submitted as a **`*.pdf`** document) in which you tell the story of your video. You should explain the significance of the video and artistic considerations that went into it. Your document should be approximately one page long.

You will *not* submit your m-files. (The code for your m-files should appear in the published **`*.pdf`** document.) You will *not* submit any graphics files that you used. (These pictures will appear in the video file.)

Checklist:

1. Published **`*.pdf`** document showing the code in your script and function m-files,
2. MPEG-4 video file, and
3. Essay (as a **`*.pdf`** file) containing the story of your collage.

Grade

Your grade will reflect primarily the technical merit of your MATLAB solution. Each slide should contain more than just a picture. Each slide may contain a picture, but it also must contain something of your own design. (In the case of **`parks`**, the original contribution is a world map with title and marker. However, please note that you are *not* being asked to create a map or anything related to geography.) The artistic merit (including, for example, your telling of the story of your collage) will also be taken into account in the evaluation of your project.

(B) Music Synthesis

The file **fugue.mat** is included as part of this project. It contains numerical data that encode about 30-seconds worth of Johann Sebastian Bach's *Fugue #2 for the Well-Tempered Clavier*. You are to write MATLAB code that loads this file, synthesizes the notes with correct timing, and then plays the notes.

You will also prepare a document discussing possibilities for the computer-generated synthesis of music.

In order to complete this project, you will need a computer with both MATLAB and sound. If you use a computer in the ECE department, you may need a set of headphones.

This project is a rewritten version of a project given on the CD accompanying the book:

J. H. McClellan, R. W. Schafer, and M. A. Yoder, *Signal Processing First*, Prentice Hall, 2003.

Each student will submit his/her own work. You may discuss this project with other students. But your submissions must be the fruit of your own labor.

Description of Project

The file **fugue.mat** contains a structure array called **voice** with three elements. (In musical terms, this fugue contains three voices.) These elements are:

voice(m), where **m = 1**, **2**, or **3**

Each of these elements is a structure. Each of the elements represents a sequence of notes that is to be played. Each structure contains three fields:

voice(m).pitch

voice(m).start

voice(m).duration

Each of these fields is a vector. For a given structure − for example, **voice(2)** − the three vectors − **pitch**, **start**, **duration** − are of equal length. These three vectors together define a sequence of notes. The vector **start** defines the start time of each note. The vector **duration** defines the duration of each note. The vector **pitch** defines the pitch of each note.

For example, the first number in the vector **voice(2).start**

voice(2).start(1)

is the start time in seconds of the first note in the sequence of notes defined by **`voice(2)`**. The first number in the vector **`voice(2).duration`**

`voice(2).duration(1)`

is the duration in seconds of this first note. The first number in the vector **`voice(2).pitch`**

`voice(2).pitch(1)`

is the pitch number of this first note.

Using the symbol *p* to represent the pitch number **`voice(m).pitch(n)`**, the pitch number for the **`n`**-th note of the **`m`**-th voice, the frequency of this note is given by

$440 \times 2^{(p-49)/12}$ Hz

A couple of examples might be helpful. When $p = 49$, the frequency is 440 Hz. When $p = 37$, the frequency is 220 Hz.

The vector **`start`** tells you when each note should start, and the vector **`duration`** tells you how long each note should last. Both of these vectors measure time in seconds.

In musical terms, each of the pitch numbers (numbers in the vector **`pitch`**) represents a key on the piano keyboard. This scheme accounts for both black and white keys. The pitch number increases as your finger moves from left to right on the keyboard. Middle C has been assigned the pitch number 40. When you move your finger on the keyboard to the right by 12 keys, the frequency doubles. For example, your finger moves 12 keys to the right in shifting from the key associated with pitch number 37 to the key associated with pitch number 49, and the corresponding frequencies are 220 Hz and 440 Hz, respectively. A set of 12 consecutive keys (including both black and white keys) is called an *octave*, corresponding to a doubling of the frequency.

Based on the above comments, the following idea should be forming in your mind. The file **`fugue.mat`** contains the same information normally found on sheet music. The program that you write will essentially replace three instruments. Your program will take the information from **`fugue.mat`** and use this information and the sound capability of the computer to play this fugue. The actual playing of the fugue is accomplished by passing a signal vector, which your program constructs, to the function **`sound`**. The signal vector will also be recorded.

Specific Instructions

You will first copy the file **`fugue.mat`** from Canvas into the folder that you will use as the current folder while working on this project.

You will write a script m-file and a function m-file. The function will be called from the script. Within the script, you will load the file **`fugue.mat`** into the MATLAB workspace using **`load`**. The script will also define a variable representing the sampling frequency and assign this variable the value 11025 (hertz or, equivalently, samples per second). This sampling frequency is only one-fourth the sampling frequency (44.1 kHz) used on a CD, but 11025 Hz is adequate for the purpose of this project.

Your function will produce a signal vector representing one note. There will be three input arguments to this function. The first of these is the pitch number. The next argument is the duration in seconds. The final argument is the sampling frequency in hertz. This function will not need the start time because the signal vector produced by this function will later be shifted to start at the correct time. (That shifting will occur in the script.) There will be two outputs from this function: one signal vector and one scalar. The scalar is the number of samples in the signal vector.

A single pure note having frequency **`f`** (Hz) and duration **`T`** (seconds) could be synthesized like this:

```
t = 0:Ts:T;

x = cos(2*pi*f*t);
```

In the above sample code, **`Ts`** is the sample period (the time between adjacent sample values) in seconds. The sample period equals the reciprocal of the sampling frequency. (Don't confuse the sampling frequency with the frequency **`f`** of the note.) The frequency **`f`** is related to the pitch number as explained in the section *Description of Project*. If you were to synthesize the notes as shown above, however, there would be audible clicks in your synthesized music. These clicks are due to the sudden arrival and departure of the notes.

You should eliminate the clicks by having each note fade in and fade out. You can accomplish this by multiplying each pure note (the **`x`** defined above) by a mask. The mask is a vector having the same length as **`x`**. The first **`k`** samples of the mask vector should increase linearly from **`0`** to **`1`**. The last **`k`** samples of the mask should decrease linearly from **`1`** to **`0`**. All other samples in the mask should equal **`1`**. You should use **`k = 100`**; this is a suitable choice for the sampling frequency of 11025 Hz that you will be using in this project. In other words, with **`k = 100`**, the fade-in of each note will last about 9 ms as will the fade-out of each note. The duration of the note will be the time from the beginning of the fade-in to the end of the fade-out. So the mask for each note will equal **`1`** for a period of time that is less than the duration of the note by about 18 ms. Since different notes will, in general, have different durations, the mask will be different for different notes. (However, the duration of every note will be greater than 20 ms, so the fade-out will never overlap the fade-in.)

The vector returned by your function will represent one masked note. As mentioned above, the function will also return a scalar equal to the number of samples in the output vector.

Your script will call your function once for each note of each voice. The script will shift the signal for each note so that it starts at the proper time. The script will then compose a signal vector that incorporates all notes of all three voices. We will call this the composite-signal vector.

You should look up the actual values of **`voice(1).start(1)`**, **`voice(2).start(1)`** and **`voice(3).start(1)`**. You will find that for a time at the beginning of the fugue only the first voice is present. In other words, during this time only one note plays. After a while, the second voice begins, so that two sequences of notes are playing simultaneously. A short while after that, the third voice begins and three sequences of notes are playing simultaneously.

In the interest of efficiency, you should pre-allocate memory for the signal vector that will be passed to **`sound`**. As usual, this pre-allocation will be accomplished with the function **`zeros`**. But first, it will be necessary to determine how long this signal vector must be. This can be determined by doing some simple calculations on the start time and duration of the last note for each of the three voices. You should program your script to do these calculations. (You could hard-code the maximum length, but that is not good programming practice.) Be aware that your calculations will, by default, employ (double-precision) floating-pointing numbers. The length that you input to **`zeros`** cannot have a fractional part. You can use the function **`uint32`** to round a floating-point number to the nearest unsigned integer. (You need **`uint32`** because **`uint8`** and **`uint16`** can't represent a large enough number for this application.)

In the script, the masked signal vector for each note must be added to the composite-signal vector (that has been pre-allocated and that, after normalization, will eventually be passed to **`sound`**). For example, the masked signal vector **`w`** of a note can be added in to the long signal vector **`y`** like this:

```
y(a:b) = y(a:b) + w;
```

where **`a`** is the index of **`y`** corresponding to the start time of the note and **`b`** is the index corresponding to the end time of the note. These index values **`a`** and **`b`** may be computed from the start time of the note, the sampling frequency (11025 Hz) and the number of samples for the note's signal vector. Make sure you round-off your calculated index values using **`uint32`**. (If you use a floating-point number with a nonzero fractional part as an index in an array, MATLAB will complain.)

It is essential to scale properly the composite-signal vector. The function **`sound`**, which will be used to play the composite-signal vector, and the function **`audiowrite`**, which will be used to record the composite-signal vector, both expect that all sample values will be between -1 and $+1$. Any sample value that does not meet this criterion will be clipped. (A sample value of -1.27 will,

for example, be clipped to -1, and a sample value of $+2.41$ will be clipped to $+1$.) As you might imagine, clipping will distort the fugue. You should recognize that even if each component note meets this criterion, the composite-signal vector, obtained by summing notes together, may not. You can achieve the desired normalization by first finding the maximum of the absolute value of the (composite-signal) samples and then dividing all samples by this maximum. It is recommended that you additionally multiply the result by 0.9; this ensures that the new, normalized composite-signal vector does not even touch the limits of -1 and $+1$.

After the normalized, composite-signal vector has been constructed, the script will use **`sound`** in order to play the music. You should use three input arguments to **`sound`**: the (normalized, composite-signal) vector, a scalar equal to the sampling frequency, and a scalar equal to 16. (This last input is the number of bits to be used per sample when playing the vector. If you don't specify 16 bits, **`sound`** will use 8 bits by default.)

When the script is executed, the command **`sound`** should be executed only once. In this single invocation of **`sound`**, the script should send the normalized, composite-signal vector, which includes every note of all three voices. If your script invokes **`sound`** more than once, the fugue won't sound right. With multiple invocations of **`sound`**, computational delay between invocations will cause the timing of the notes to be wrong.

When played correctly, this fugue begins with just one voice, a second voice joins in after a few seconds, and the third voice joins in a few seconds after that. If you have done the masking (in the function m-file) correctly, there should be no clicking sounds.

Finally, you will record the normalized, composite-signal vector in a WAVE (waveform audio format) file (**`*.wav`**) using the **`audiowrite`** function. (A WAVE file does not employ compression and is much less memory efficient than MP3; but a WAVE file is acceptable for small sound snippets.) Use **`audiowrite`** with three inputs: a string for the filename, the (normalized, composite-signal) vector, and the sampling frequency. In the interest of conserving memory, **`audiowrite`** saves each sample using just 16 bits, rather than with the full 64 bits (that is, 8 bytes) of a **`double`**. After saving the signal vector to a WAVE file, use the function **`audioinfo`** to obtain some basic facts about the saved WAVE file. Then print out the sample rate and the total (number of) samples for the WAVE file (with information provided by the function **`audioinfo`**). A WAVE file can be played even outside of MATLAB.

What You Will Submit

You will submit three files to Canvas.

One file will be a document (preferably submitted as a PDF file) in which you discuss possibilities for the computer-generated synthesis of music. Your document should be approximately one page long.

The other two files will be your script m-file and your function m-file. You need *not* submit the WAVE file. You will ***not*** be publishing your script m-file.

Checklist:
1. Document (as a PDF file) containing your discussion of music synthesis
2. Script m-file
3. Function m-file

Grade

Your grade will reflect primarily the technical merit of your MATLAB solution. However, the artistic merit (including, for example, your discussion of music synthesis) will also be taken into account. The grader will be checking that you followed all instructions.