**Eng. 100: Music Signal Processing** W24

©J. Fessler
January 5, 2024 13:53

# Lab 1: `Julia` and sampling

## 1 Abstract

These lab descriptions are essentially the textbook for this course. Study them carefully!

This lab introduces some basic tools that you will use throughout the course. The goals of this lab are as follows.

(1) To learn to apply basic `Julia` language features that will be used in Engin 100.
(2) To learn the basics of sinusoidal signals.
(3) To learn the basics of sampling continuous-time signals.

Projects later in the course will involve simple `Julia` programs that synthesize music and that process sampled continuous-time signals (music) to determine their sinusoidal components.

## 2 Background

### 2.1 Embedded audio and links

Throughout the PDF files for the digital signal processing (DSP) lecture notes and labs and projects, there are audio examples that you can hear by using the (free) Adobe Acrobat Reader application and clicking the play buttons. Other PDF readers may not be able to play the embedded audio. Here is an example: $\boxed{\text{play}}$ . You should hear a 2 second long 440 Hz tone when you click that play button, assuming you have a suitable PDF reader, a working sound card and speaker or earphones, the system volume high enough, etc. On my iMac I had tell Acrobat Reader to "trust" this document, and then I had to close the file and reopen it and click again to hear it. And it works for me only Acrobat Reader, not Acrobat or Preview.

Also, throughout the PDF files are embedded URLs (uniform resource locator, aka web address), annotated using cyan boxed words. There are embedded links to the **`Julia` manual** and to wikipedia resources. Clicking on any of them should open an Internet browser window related to the topic for more information. Try clicking on "Adobe Acrobat Reader" above and your browser should open to a corresponding Wikipedia page.

### 2.2 `Julia`

`Julia` is a free and open-source programming language that was developed specifically for scientific and numerical computing. It combines the convenience of interactive programming (like Python and `Matlab`) with the speed of compiled languages (like C/C++). Its syntax incorporates the best features of Python and `Matlab`.

Section 3.2 describes how to download and install the free `Julia` application from `https://julialang.org`. You may use it with any code editor; we recommend using the free VSCode editor with its `Julia` extension for convenient debugging. You also can use Jupyter notebooks (the Ju in Jupyter is for `Julia`). It will be most convenient for you to install `Julia` on your own computer, but you can also install it locally on a CAEN workstation if needed.

`Julia` provides a convenient interactive environment for scientific computing. It excels at manipulating vectors and matrices, *i.e.*, arrays of numbers, and we will see that (digital) music signals are simply arrays of numbers. A `Julia` program is a list of `Julia` statements executed in succession. This lab introduces some `Julia` basics.

## 2.3 Sinusoids

Sinusoids have a central role in Engin 100. They are the building blocks of musical sounds.
A sinusoid or sinusoidal signal or sine wave is a function or signal of the form

$$x(t) = A\cos(2\pi f t + \theta), \ -\infty < t < \infty. \tag{1}$$

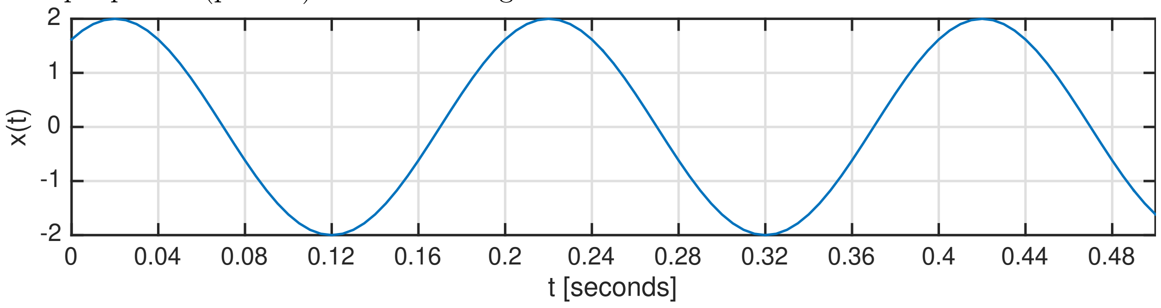
This sinusoid is a function of time $t$, typically measured in seconds. The signal $x(t)$ is a function of $t$; for each value of $t$, there is an associated signal value $x(t)$. Each sinusoidal signal is defined in terms of the following three constants:

- $A$ = amplitude. The amplitude $A$ and the signal $x(t)$ have the same units (*e.g.*, "volts").
  $A \geq 0$ always. In music, if $A$ increases then the tone sounds louder.
- $f$ = frequency in Hertz = $\frac{\text{CYCLES}}{\text{SECOND}}$.
  $x(t)$ has period $T = 1/f$ seconds, so $x(t) = x(t+T) = x(t+1/f)$ for all $t$.
  In music, this frequency is also called the pitch.
- $\theta$ = phase or phase shift, in radians or degrees. Phase = $\theta$ is equivalent to time delay

  $$\tau = -\frac{\theta}{2\pi f} \text{ seconds.}$$

  Another (less common) way of writing the sinusoid in terms of a time delay $\tau$ is $x(t) = A\cos(2\pi f(t-\tau))$.
  Note that $\sin(\alpha) = \cos(90° - \alpha) = \cos(\alpha - 90°)$, so a sine is a cosine with phase $\theta = -90°$.

Here is an example plot of (part of) a sinusoidal signal:



Let us "reverse engineer" this plot to express the sinusoid mathematically.

- Amplitude: $A = 2$. This is the maximum value of the signal because the maximum value of cos is 1.
  (In contrast, the peak-to-peak amplitude = 4).
- Period: examine the time difference between two peaks: $T = 0.22 - 0.02 = 0.42 - 0.22 = 0.2$ seconds.
  So the frequency is $f = 1/T = 5$ Hertz.
- Time delay: normally the first peak of cos is at 0, but here it is at $\tau = 0.02$ seconds. That is the time delay.
  The phase is thus $\theta = -2\pi 5\tau = -2\pi 5\,(0.02) = -\pi/5$ radians.
- So two mathematical formulas for this sinusoidal signal are:

  $$x(t) = 2\cos(2\pi 5(t - 0.02)) = 2\cos(10\pi t - \pi/5).$$

Note that "$x$" is on the vertical axis here; get used to this!

### 2.4 Reading questions

You must read the description of each lab and project before the start of the corresponding lab section. To help ensure that you do, each lab/project writeup includes a few easy "reading questions" that you must answer on Canvas **no later than the start of the earliest lab session**.
*Because our class begins on Wednesday in W24, and labs begin on Thursday, for Lab1 you must complete the RQs no later than the start of the earliest lab session in the* second *week.*

Here are the first two such questions. There are more later in the document. Submit your answers to Canvas after you finish reading this lab.

These reading questions are *individual* work. You must do them by yourself.

RQ Lab1.1. A sinusoidal signal has amplitude=8, frequency=100Hz, and phase $\theta = \pi/3$.
What is the value of the signal at $t = 0.02$ seconds?

Because this is a warm-up problem, here is the answer. The problem is asking for the value $x(0.02)$, where the sinusoidal signal is $x(t) = 8\cos(2\pi 100t + \pi/3)$ so we simply substitute $t = 0.02$ into the formula: $x(0.02) = 8\cos(2\pi(100)0.02 + \pi/3) = 8\cos(4\pi + \pi/3) = 8\cos(\pi/3) = 8/2 = 4$.
Even though the answer is given, you still must enter it into Canvas to verify you read the lab in advance!

RQ Lab1.2. A sinusoidal signal has frequency=50Hz and phase $\theta = 2\pi/3$. The value of the signal at $t_1 = 0.01$ is $x(t_1) = 7$. What is the amplitude of the sinusoid?

### 2.5 Sampling

Unless you have an analog computer, continuous-time signals such as music must be sampled or discretized into an array of numbers that can be processed with a (digital) computer. The numbers themselves must also be quantized or digitized into a finite number of bits (finite precision). Engin 100 will not explicitly deal with quantization much, although we will see some of its effects in the next lab[1]. In signal processing, the term "sampling" does *not* mean incorporating a short segment of one recording into another, as is often done in certain music styles today.

Sampling is performed by analog-to-digital (A/D) converters. These are very sophisticated electronic components present in any digital audio system. Fortunately, all we need here is a simple mathematical model. The usual way to sample a continuous-time signal $x(t)$ is to look at $t = n\Delta$ for integers $n$ for a desired sampling interval $= \Delta$ seconds. The corresponding sampling rate is $S = \frac{1}{\Delta}\frac{\text{Sample}}{\text{Second}}$. The units of the sampling rate $S$ is Hertz or $\frac{\text{Sample}}{\text{Second}}$. When the continuous-time signal $x(t)$ is the input to an A/D converter, the output is the following array of numbers that can be stored digitally:

$$\begin{aligned}&(x(0),\ x(\Delta),\ x(2\Delta),\ x(3\Delta),\ \ldots) = \\ &(x[0],\ x[1],\ x[2],\ x[3],\ \ldots),\end{aligned}$$

where we relate the digital signal $x[n]$ and the analog signal $x(t)$ as follows:

$$x[n] = x(n\Delta) = x(t)\Big|_{t=n\Delta} = x\left(\frac{n}{S}\right) = x(t)\Big|_{t=n/S}. \qquad (2)$$

If $x(t)$ has finite duration (length), *i.e.*, it starts at some time and stops at another, as all music does (*e.g.*, musical chairs), then $x[n]$ is a finite list of numbers (an array). This makes $x[n]$ suitable for computer processing.

For a sinusoidal analog signal of the form (1) that is sampled at $S$ "Hertz" $= S\frac{\text{Sample}}{\text{Second}}$, (*i.e.*, every $\Delta = \frac{1}{S}$ seconds), the resulting sampled signal is

$$x[n] = x(n\Delta) = x(t)\Big|_{t=n\Delta} = A\cos\left(2\pi f \underbrace{\Delta n}_{\text{t}} + \theta\right) = A\cos\left(2\pi\frac{f}{S}n + \theta\right),\ n = 0, 1, 2, \ldots. \qquad (3)$$

Note that $f/S$, $f\Delta$ and $n$ are all dimensionless. It is usually desirable to put equations in dimensionless form.

It might seem that we will lose information about $x(t)$ by storing only its samples $x[n] = x(n\Delta)$. How can we recover the original analog signal $x(t)$ from it samples $x(n\Delta)$, *e.g.*, for audio playback? Amazingly, we will see later in Engin 100 that we can reconstruct $x(t)$ *exactly* from its samples $x[n] = x(n\Delta) = x(n/S)$ as long as $S > 2f_{\max}$, where $f_{\max}$ denotes the maximum frequency of $x(t)$. This fact was discovered by Claude Shannon, a UM alumnus. His bust is outside the EECS building, and in Gaylord, MI, where he grew up.

To illustrate these concepts, consider a standard music CD that holds 64 minutes of music. The original analog musical signal is sampled at $S = 44100\frac{\text{Sample}}{\text{Second}}$ and quantized using 16 bits per sample (providing $2^{16} = 65536$ possible different signal values per sample). There are two channel components because it is stereo (although early Beatles recordings were monophonic, among others). So we can determine how many bytes are stored on the CD as follows (watching the units):

$$\left[\frac{1\,\text{byte}}{8\,\text{bits}}\right]\left[16\frac{\text{bits}}{\text{channel}}\right]\left[2\frac{\text{channel}}{\text{sample}}\right]\left[44100\frac{\text{sample}}{\text{second}}\right]\left[60\frac{\text{second}}{\text{minute}}\right][64\,\text{minutes}] = 677\,\text{MBytes}.$$

[1]If you buy a music CD (does anyone still do that?) and import it to another digital format such as MP3, then the import software (*e.g.*, iTunes) will likely give you options to set the encoding quality to high or low; this choice influences quantization.

RQ Lab1.3. A sinusoidal signal has amplitude=6, frequency=2000 Hz, and phase $\theta = 0$. The signal is sampled at rate $S = 8000 \, \frac{\text{Sample}}{\text{Second}}$. What is the value of the sampled signal $x[n]$ when $n = 4$?

Now we introduce `Julia` so that we can plot signals like sinusoids, manipulate them, and listen to them. But first we start with the basics.

---

## 3 Getting started with `Julia`

### 3.1 About `Julia`

All code examples in class and labs and projects will use the Julia programming language. Why?

- It is free and open source.
- It is a real programming language, developed for numerical computing [1].
- Used by Freshmen in Robotics 101 at UM, in ENGR108 at Stanford, and in a similar UCLA course.
- It is interactive (like Python and `Matlab`), yet has fast execution because it is compiled.
- Much of its syntax is similar to `Matlab` and Python.
  Here is a **list of noteworthy differences from `Matlab`**
- DSP / data-science / machine learning are all done with many software languages...
- Jupyter notebooks (based on IPython) are educational, integrating math with documented code and figures.

In parallel to working on Lab1, you should work on HW0 which is an interactive `Julia` tutorial. The HW0 deadline is a bit later to give you time to work through it, but the sooner you learn it, the easier the labs will be!

Some documentation / books:

- Online documentation: (stable version) **(latest version)**
- Wikibook Intro to `Julia`: `https://en.wikibooks.org/wiki/Introducing_Julia`

## 3.2 Installing and running `Julia`

During your first lab session, your lab instructor will help you with running `Julia` (and with installation if you get stuck), but it will be very helpful for you to get as far with set up as you can *before* lab.
We recommend (but do not require) that you use the `Julia` extension with VS Code:
`https://www.julia-vscode.org`, because of its excellent integration with the Julia Debugger.
Alternatively, you may use your own favorite editor (though you may find debugging more challenging).

Follow installation instructions at `https://www.julia-vscode.org/docs/dev/gettingstarted`

- Download and install `Julia`, at least version v1.10.0, from the "manual download" section here:
  `https://julialang.org/downloads/#official_binaries_for_manual_download`.
  We recommend trying the binary installer (windows) or the .dmg file (mac) first,
  rather than `winget` or `curl` or `juliaup`.
  You can use any computer that has a sound card for audio.
- Download and install VS Code from `https://code.visualstudio.com`
- Launch VS Code, typically by clicking its icon in your Application folder.
- Install the `Julia` VS Code extension ("Julia Language Support"), following the instructions.
- Restart VS Code.
- Optional power-user tip: set up your shell to make sure the `julia` executable is in your path.
  On my Mac, I made a link in `/usr/local/bin` that points to the executable:
  `[path-to-julia]/Contents/Resources/julia/bin/julia`
- Peruse VS Code tutorials: `https://code.visualstudio.com/docs`
- Customize VS Code (if you want), *e.g.*, by installing your favorite keymap. (I use vim – retro...)
- Follow the Hello World instructions to run your first `Julia` program.
- Either within VS Code, or separately, launch a `Julia` REPL with prompt: `julia>`
- Type `1+2` to verify it works.
- Press the `?` key and then type a function name like `range` to get documentation about it.
- Experiment with some basic statements like `x=3` and `x+2` and `x^2`
- Use `Julia`'s **package manager**, Pkg, to add some `Julia` packages that you will use in this course:
  ```
  using Pkg
  Pkg.add(["FileIO", "FFTW", "HTTP", "MAT", "Plots", "Polynomials", "Sound", "WAV"])
  ```
  If any of these fail to install, then try adding them one at a time, like `Pkg.add("FileIO")`
  If you want to work with Jupyter notebooks, then also `Pkg.add("IJulia")`
- Here is an alternative way to install packages.
  At the REPL, press the `]` key to enter the **package manager** that has a prompt like `(v1.10) pkg>`
  Add some useful packages using the package manager:
  - `add Plots`
  - `add Polynomials`
  - `add FFTW`
  - ...
  - Press the delete key to exit the package manager and return to `Julia`'s main REPL.
- If you have problems later, return to the package manager and run `update` to get the latest package versions.
- For documentation of the `Plots.jl` plotting package, see:
  `http://docs.juliaplots.org/latest/`
  `https://github.com/sswatson/cheatsheets/blob/master/plotsjl-cheatsheet.pdf`

The next section really begins using `Julia`.

## 3.3 Vectors and arrays

Because `Julia` is designed for scientific computing, when using `Julia` you often think in terms of vectors (1D arrays), matrices (2D arrays) and possibly higher dimensional arrays. Examples of a (column) vector, a matrix, and the transpose of that matrix in mathematical notation are, respectively:

$$\boldsymbol{x} = \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix} \qquad \boldsymbol{Y} = \begin{bmatrix} 3 & 4 & 7 \\ 1 & 5 & 9 \end{bmatrix} \qquad \boldsymbol{Z} = \boldsymbol{Y}' = \begin{bmatrix} 3 & 1 \\ 4 & 5 \\ 7 & 9 \end{bmatrix}.$$

In `Julia` syntax you enter such simple arrays by typing (you can use cut-and-paste from the lab pdf) the following statements at the REPL:

`x = [3, 1, 4]`

`Y = [3 4 7; 1 5 9]`

`Z = Y'` or `Z = transpose(Y)`

The transpose operation `Z = transpose(Y)` means convert rows to columns, and vice-versa. The conjugate transpose operation $\boldsymbol{Z} = \boldsymbol{Y}'$, also known as the Hermitian transpose, is the combination of transpose and complex conjugate of each matrix element. For a matrix of real numbers, the complex conjugate has no effect, so transpose and conjugate transpose are equivalent, so we usually just use type `Z = Y'` in `Julia`. Note the convenient similarity between the mathematical notation and the `Julia` syntax[2].

In both `Julia` and Python, a vector is a 1D array, and is best thought of as a column vector. In `Matlab`, there are no true vectors! There are $N \times 1$ arrays (often thought of as column vectors) and $1 \times N$ arrays (often thought of as row vectors). This sloppiness in `Matlab` often leads to confusion. In `Julia`, typing `size(Y)` returns `(2, 3)` because `Y` is a $2 \times 3$ matrix, *i.e.*, a matrix with 2 rows and 3 columns. Typing `size(x)` returns `(3,)` because `x` is a true 1D array (a vector) with 3 elements. It is *not* a "$3 \times 1$" array! We can transpose the vector $\boldsymbol{x}$ by typing `y = x'`, which produces a special type of "adjoint" vector of size `(1,3)`. (You can think of it as a row vector if you want.) If we transpose back by typing `z = y'`, then we again get a true 1D array that is the same as `x`. Try it in `Julia` to see. When in doubt, use the `size` function to check.

To make a 1D vector we can either use `vec([3 1 4])` or insert commas (or semicolons): `[3, 1, 4]`.

Try the following statements in `Julia` and note what they are doing (`reshape` will be *very* useful later):

- `A = 10*[1 2 3; 4 5 6]` (Create a $2 \times 3$ array and multiply every element by 10.)
- `A[2,3]` (Access the element of `A` in row 2 and column 3.
- `b = vec(A)` (Stack A by columns.)
- `reshape(b,3,2)` (Convert the column vector `b` into a $3 \times 2$ array.)
- `x = vec([3 1 4 1 5 9 2 6 5 3 5]); d = [x[n+1] - x[n] for n in 1:10]`
  This takes differences between the elements of `x`, using a useful `Julia` technique called **comprehensions**.
  What is the size of the vector `d`?
- `x[4], x[end]` (Access 4th and last element of `x`.)
- `x = vec([3 1 4 1 5 9 2 6 5 3 5]); c = x[2:11] - x[1:10]`
  This is another way to take differences of `x`.
  A more elegant way is: `c = x[2:end] - x[1:end-1]` because this works for any length vector `x`
  The simplest way of all uses the built-in `Julia` function `diff` as follows: `c = diff(x)`

[2]Another name for complex conjugate of a matrix is adjoint, and `Z = adjoint(Y)` is another way to perform conjugate transpose. You will never need to use `adjoint` in this class, but if you want to look at documentation for what `Z = Y'` is doing, you need to look at the **adjoint** function in the `Julia` manual because internally the syntax `Z = Y'` is translated into `Z = adjoint(Y)`.

- Try to predict what the result will be before you enter the following: `x = [10, 20, 30, 40]; s = sum(x)`
- `a = [1, 2, 3]; b = [5, 6, 7]; c = [a; b]`
  The operation `[a; b]` concatenates vectors `a` and `b`, appending `b` after `a` to make a longer vector `c`.
- The colon `:` is useful for generating vectors of values called **ranges**. Try these statements and think about the output. You will need to use such statements frequently:
  `x = 3:7`, `t = 2:1/10:3`, `y = 10:2:20`, `z = 40:-1:30`
  To see the full list of values, you can use the `collect` function:
  `collect(x), collect(t), collect(y), collect(z)`
  Only rarely does one need to use `collect`.
- At any point if you want to display the value of an array you can either just type the array name, *e.g.*, `x`, and press enter, or use the show function: `show(x)`
  When debugging functions you write, it is helpful to use the macro version: `@show x`
- Because `Julia` is a compiled language, unlike `Matlab`, it does not have any "clear" function. If you want to start over, you should exit `Julia` using `exit()` or ctrl-D.

### 3.4 Broadcast

Often we want to apply the same function, like cos, to every element of an array. In `Julia` this is called a **broadcast** operation. It is usually implemented by appending a dot `.` to the function name.

Try the following:
`t=0:10; x = cos.(2*pi*t/10)`

Here `t` is a vector having 11 elements. (You can verify that by typing `length(t)`.) The argument `2*pi*t/10` is also a vector with 11 elements.

The broadcast syntax `cos.( )` tells `Julia` to apply the `cos` function to every element of the 11 element vector argument.

If you try to type `cos(t)` directly, you will get an error message, because mathematically the cos function is defined for scalar arguments[3].

### 3.5 Plotting

A particularly common `Julia` task you will perform is plotting (samples of) functions.
Try running the following examples. (This is not yet part of the lab you will submit):

- You must start by telling `Julia` to use the `Plots` package.
  See `https://docs.juliaplots.org` for plotting package documentation.
  `using Plots`
- `t = 1:100; x = cos.(2*pi*t/30); plot(x)` (Plot a sinusoidal signal with period=30.)
- `t = 1:100; y = 2.0.^(t/20); plot(y, ylabel="2^(t/20)", label="")`
  (Plot an exponential function growing in time.)
  Here the `.^` is another broadcast meaning element-wise exponentiation.
- `t = 1:100; z = 2.0 .^(-t/20) .* cos.(2*pi*t/30); plot(z, title="decay")`
  (Plot a decaying sinusoid.)
  Here the `.*` is another broadcast meaning element-wise multiplication of two vectors.

[3] In `Matlab` you would not need the dot, but then it is harder to read `Matlab` code and understand what it is doing in general. In `Julia` the dot conveys to the reader that the operation is element-wise.

- Typing `t` and `x` will each display for you a vector of 100 numbers. You can display them as side-by-side columns by typing `[t x]` which forms a $100 \times 2$ matrix.
- To plot multiple things on one plot use `plot!` (note the colors):
  `plot(t, x, label="x"); plot!(t, z, label="z")`
- To plot x vs. y as a scatter plot:
  `scatter(x, y)`
  In such cases the number of elements in `t` and `x`, or `x` and `y`, must be the same.

## 3.6 Miscellaneous features

These simple examples illustrate several important points about `Julia`.

- Using a broadcast operation like `t = 1:5; x = exp.(t)` to define `x` as a function of `t` produces a vector of `x` values corresponding to the vector of `t` values very concisely. Alternatively you could use `Julia`'s array comprehension:
  `x = [exp(t) for t in 1:5]`
  but the broadcast version is less typing and easier to read.
  Another alternative (that would be necessary in many other programming languages) is to write a loop:

```
t = 1:5
x = zeros(length(t)) # initialize
for i in 1:length(t)
    x[i] = exp(t[i])
end
```

  `Matlab` and Python users generally try to avoid loops because they are interpreted languages so loops are slow. `Julia` is compiled so using loops generally is fine from a speed perspective, but certainly the broadcast version is more readable and much less typing here.
- Putting a semicolon "`;`" at the end of a statement suppresses output; without it the `Julia` REPL will show the results of the computation on your screen. This is harmless, but can be annoying.
- The operation "`.*`" multiplies two vectors *element-by-element*, whereas "`*`" multiplies vectors as in linear algebra.
  Try these: `[1, 2] * [3, 4]'` and `[1, 2]' * [3, 4]` and `[1, 2] .* [3, 4]` and `[1, 2] * [3, 4]`
  (The last one gives an error message.)
- Compare `collect(0:.1:1)` (11 numbers spaced by 0.1) and
  `collect(range(0,1,10))` (10 numbers spaced by 0.111).
  This difference is sometimes called the fencepost error or picket fence error. If you have a 30 foot fence with pickets every 3 feet, how many pickets are there? Answer: 11 (not 10). Programmers often introduce bugs by disregarding this distinction.

## 3.7 Getting help for `Julia`

- To get help for a `Julia` function, say, `range`, type `?range` or `@doc range`.
- See `https://web.eecs.umich.edu/~fessler/course/100/misc/whywont-julia.htm` for ideas on what might be wrong with your program.

### 3.8 Programs (jl-files)

You can access previously-typed statements using the up-arrow and down-arrow on your keyboard. But typing gets old. Instead, you can write (and save) a *program* or script (a sequence of `Julia` statements) using a *jl-file*. In the VS Code editor, under the "File" menu select 'New File" and choose `Julia` as the language (or just name the file something with a `.jl` suffix). Use the editor to type in a sequence of `Julia` statements, then save. To execute that script, either push the "play" button in the VS Code editor (if you are actively editing that jl-file), or use an `include` statement at the `Julia` REPL, like `include("mycode.jl")`

### 3.9 Saving figures

As long as you have loaded the `Plots` package (by typing `using Plots`), you can save the current figure by typing `savefig("filename.png")` at the `Julia` REPL. Or, in VS Code there is a small "floppy disk" icon above the plot that you can click for saving. You could also capture the plot window using your operating system (typically saves it as a PNG file). Afterwards, you can insert the figure into a google doc to include it in a report. You will do this frequently in your lab reports and presentations.

### 3.10 Subplots

You can combine multiple plots into one figure by using the `plot` function. Try the following example.

```
using Plots   p1 = plot(cos.(2*pi*(1:100)/100), label="cos")
p2 = scatter(sin.(2*pi*(1:10)/10), label="sin")
plot(p1, p2)
```

The variables `p1` and `p2` contain plot descriptions. You can see their type by typing `typeof(p1)` which returns `Plots.Plot{Plots.GRBackend}`. `Julia` has numerous variable types, and `typeof` is very useful for learning about them.

### 3.11 Multiple dispatch

You might note above that the same function name `plot` is used into two very different ways. In the first use, the argument is a vector of cos values. In the second use, the arguments are plot descriptions. This feature in `Julia` is called multiple dispatch. It means that different methods are called based on the types of one or more of its arguments. This feature is closely related to function overloading, as used in C++ and Java, for example.

### 3.12 Exiting `Julia`

To exit `Julia`, type `exit()`.

RQ Lab1.4. What is the name of the `Julia` operation that applies a function to each element of its argument?

## 4 Lab 1: What you must do

### 4.0 Start a Google Doc for your lab results

After completing this lab you will submit a PDF file of your results to Gradescope for grading. You will generate the PDF file using Google docs. So before starting, point your browser to `http://drive.google.com` and login (with your umich account) and create a new document. Put your name, date, and section number at the top of the document. Put all of the requested figures and answers into this document. Ask your lab instructor for help if needed. If you have not used it already, Google Docs is worth learning and using for this report because it allows collaborative editing that will be useful later for the team projects. (This lab is an *individual* assignment.)

Section 3 suggested a lot of functions that you should try yourself at the `Julia` REPL *before* proceeding with the rest of this lab. For the rest of this lab, we recommend you enter the statements in one or more jl-files as described in Section 3.8. Your lab instructor can help show you how to do this.

### 4.1 Sinusoids

[5] Use `plot` to make a figure consisting of the four (sub)plots described below, following Section 3.10.
Start by typing `using Plots`.

(a) Type `t = 0:10; x = 3*cos.(t) + 4*sin.(t); p1 = plot(t, x, label="(a)")`
This should be a jagged-looking plot.
The reason is that it is only sampled at integers, and samples are connected by straight lines.

(b) Type `t = 0:0.1:10; x = 3*cos.(t) + 4*sin.(t); p2 = plot(t,x, label="(b)")`
This should be a smoother plot.
Are you surprised that the sum of a sin and a cos is a pure sinusoid? See the Appendix below for why.

(c) Prepare to use the Sound package, then listen to a sound by typing:
`using Sound`
`n = 1:4000; S = 8192; x = cos.(2*pi*440*n/S); sound(x, S)`
This tone should sound like note “A” (440 Hz) if your computer audio is working properly.

(d) Type `p3 = plot(x, label="(d)")`
This should be a blue smear! It is about 200 cycles squished together.

(e) Type `x[1:8]`
This extracts the first 8 elements of the vector `x` and displays them.
You will use frequently this way of extracting certain elements of a vector.

(f) Type `p4 = plot(x[1:100], label="(f)", marker=:square)`
This extracts the first 100 samples of `x` and then plots them so we can better see (part of) the sinusoid. The squares show the samples, and the plot just “connects the dots” between them.

(g) Type `sound(x[1:2:end], S)`
This extracts every other value of `x`, halving its length.
The sinusoid is squished together and effectively is sped up by a factor of two, so the tone is now “A” one octave higher (880 Hz).

(h) Combine all four plots into one following Section 3.10.

(i) Add a title to the combined plot by typing
`plot!(plot_title = "Lab 1 Figure 1", plot_titlefontcolor=:red)`
Save this four-subplot figure as a PNG file and import it into your Google Doc.
The plot title and font color are just two of numerous plot attributes that you can control.

### 4.2 Non-sinusoidal signals

[5] Use subplots with `plot` to make a figure consisting of the four plots described below. It will be best to combine all the code below into a single jl-file.

(a) Type:

```
using HTTP
using FileIO: load
using Sound: sound
using Plots
url = "https://web.eecs.umich.edu/~fessler/course/100/misc/train-whistle.wav"
(y, S, _, _) = load(HTTP.URI(url))
sound(y, S)
```

This signal should sound like a train whistle.

(b) Type `p1 = plot(y, label="train", xticks=[1,length(y)], ylim=(-1,1), yticks=-1:1)`
This will be a blue smear mainly showing the envelope. Then to zoom in type:
`n = 1501:1700; p2 = plot(n, y[n], marker=:circle, label="y zoom")`
Note that this signal is approximately periodic.
[2] Q1: How many samples are displayed in the second plot? (The answer is not 1700!)

(c) Type `sound(y[1:2:end], S)` Again this doubles the pitch and halves the length.

(d) Type `z = y .* cos.(2*pi*1000*(1:length(y))/S); sound(z, S)`
This operation is another way to alter pitch, called modulation.
Multiplying by this cosine shifts all of the frequencies of `y` both up and down by 1000 Hertz.

(e) Type `p3 = plot(z, label="z", xticks=[1,length(z)], ylim=(-1,1), yticks=-1:1)`
Again it is a similar looking blue smear; to zoom in:
`n = 1501:1700; p4 = plot(n, z[n], marker=:star, label="z zoom")`
Note how this signal `z` differs considerably from the original signal `y`.

(f) Combine all 4 plots into one using `plot`, add an appropriate title, save as PNG, and import to your report.

### 4.3 Sum of two sinusoids

[5] Make a figure consisting of the four (sub)plots described below.

(a) Type `S = 8192; t = (1:S)/S; z = cos.(2*pi*440*t) + cos.(2*pi*444*t); sound(z, S)`
Describe this sound. Does it sound like two tones close together in pitch? Or like a single tone with a time-varying amplitude?

(b) Type `p1 = plot(z, label="cos + cos")` and zoom in using
`n = 1501:1700; p2 = plot(n, z[n], marker=:hex, label="z zoom")`
You need both of these plots. Now can you *see* why this sounds the way it does? See Appendix below for why.

(c) Type `t = 0:.01:1; f=2; x = cos.(2*pi*f*t); p3 = plot(t, x, marker=:diamond, label="cos 2")`
This is just a sampled 2 Hz sinusoid.

(d) Type `t=0:.01:1; f=98; x = cos.(2*pi*f*t); p4 = plot(t, x, marker=:cross, label="cos 98")`
Just a sampled 98 Hz sinusoid.

(e) Notice something unsettling? Try this:
`t=0:.01:1; d = cos.(2*pi*98*t) - cos.(2*pi*2*t); plot(t, d)` (Do not print this plot.)
Sampling 2 Hz and 98 Hz sinusoids at $\Delta = 0.01$ seconds, *i.e.*, $S = 100$ Hz, yields the same result! This

phenomena is called aliasing. We will learn later that we must sample much faster than 100 Hz to distinguish a 98 Hz signal from lower frequency signals.

(f) Combine plots, add an appropriate title, save as PNG, and import to your report document.

(g) Type `S = 8192; t=(1:S)/S; y = 0.6*cos.(2*pi*440*t) + 0.6*sin.(2*pi*440*t); sound(y, S)`
Does this sound like one tone or two? (See Appendix.)

## 4.4 Do not always believe numerical results from software

- Is software like `Julia` infallible? Type:

```
using Polynomials
p = fromroots(ones(11)) # defines a polynomial from its roots
r = roots(p) # computes roots of polynomial
```

This code forms the polynomial $(z-1)^{11}$ having 11 repeated roots at $z = 1$, and then computes the roots of that polynomial. Did you get 11 ones? No!

- To see what happened, type

```
scatter(real(r), imag(r), label="roots",
  xlims=(-1.2,1.2), ylims=(-1.2,1.2), aspect_ratio=1,
  xlabel="Real part", ylabel="Imaginary part")
```

Are computers infallible? All software has numerical problems for polynomials with multiple roots. (You need not print or save this plot.)

## 4.5 Questions

Answer these in your google doc, labeled clearly Q2 ... Q6.

[2] Q2. Determine how many elements are in the vector `x` produced by the statement `x = 3:0.01:5`

[2] Q3. Determine the spacing between the elements of the array `x = range(3,5,200)`

[3] Q4. Use `Julia`'s `sum` function to determine the sum of the values $300 + 301 + \cdots + 400$. Hint: use a colon. Report your code and the result.

[3] Q5. A sinusoidal signal with amplitude 6, frequency 10 Hz, and phase $2\pi/15$ is sampled every 2 msec using the ideal sampling formula (2). What is the value of the digital signal $x[n]$ when $n = 5$?
Hint: the numbers have been designed here so that you could do this without a calculator, though some students may need to review the unit circle.

[5] Q6. Plot the signal $x(t) = \sin(2\pi 100t)$ for samples of $t$ spaced by $1/8192$ seconds over the interval [3, 3.1] seconds. Label both axes.

### 4.6 Lab report

Add to your Google Doc report the answers to the 6 questions above. Then use the Google Doc menu `File` → `Download As` → `PDF Document` to export a PDF file. Upload that PDF file to Gradescope. If possible, show it to your lab instructor first.

If needed, you should keep seeking help from classmates and your lab instructor until you get all 30 points (and all the concepts). You should finish it all by the end of your lab section during the 2nd week of class.
Read Lab 2 thoroughly before coming to the next lab section.

## 5 Appendix: sum of sin and cos of same frequency

- Why is the sum of a sine and a cosine (of same frequency) in Section 4.3(g) a pure sinusoid rather than something more complicated?
- Why did the sum of two sinusoids at close frequencies in Section 4.3(a) appear to be one sinusoid with varying amplitude?

Answers to both questions follow from the trigonometric identity called the cosine angle difference formula:

$$\cos(\alpha - \beta) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta)\,.$$

- First, set $\alpha = 2\pi f t$ and $\beta = \theta$ and multiply by $C$ to get

  $$C\cos(2\pi f t - \theta) = C\cos(\theta)\cos(2\pi f t) + C\sin(\theta)\sin(2\pi f t) = A\cos(2\pi f t) + B\sin(2\pi f t)\,.$$

  Setting $t = 0$ yields the equality:
  $$A = C\cos(\theta)\,.$$
  Setting $t = \frac{1}{4f}$ yields the equality:
  $$B = C\sin(\theta)\,.$$
  Combining the formulas for $A$ and $B$ yields (think about polar coordinates):

  $$C = \sqrt{A^2 + B^2}, \quad \tan\theta = \frac{B}{A}.$$

  In other words, the sum of two sinusoids of the *same frequency* is just another sinusoid of that frequency but with a different amplitude and phase:

  $$A\cos(2\pi f t) + B\sin(2\pi f t) = \underbrace{\sqrt{A^2 + B^2}}_{\text{amplitude}} \cos\left(2\pi f t \underbrace{- \arctan(B/A)}_{\text{phase}}\right). \tag{4}$$

  Lab 3 uses this equation extensively.
- Second, set $\alpha = 2\pi 442 t$ and $\beta = \pm 2\pi 2 t$ and add and subtract the results to get:

  $$\cos(2\pi 440 t) + \cos(2\pi 444 t) = 2\cos(2\pi 2 t)\cos(2\pi 442 t)\,.$$

  So 440 Hz and 444 Hz sinusoids added together are identical to a 442 Hz sinusoid with a time-varying amplitude. Mathematically these two sides of the above equation are identical, but our ears "hear" the right-hand side. We will use this result (think tuning a piano) in Lab 2 to measure frequency.

## References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.