A touch of R in robotics

Eric and Ian

Table of contents

Preface

Hujambo and welcome to the accompanying guide for our rstudio::conf(2022) talk: *A Touch of R in Robotics*.

In these series of notebooks, we try as much as possible to provide detailed explanations of the methodologies used along with the corresponding code.

We hope you enjoy it as much as we did .

> **Note**
>
> `Hujambo` is a *Swahili* word used in the context of `Hello` or `How are you?`. One typically replies with `Sijambo` meaning `I am fine`

1 Speech to text

Ian Muchiri and Eric Wanjau

This notebook contains a step by step guide on how to record audio in R using the audio package and convert the recording to text using Microsoft Azure Cognitive Services, specifically the speech-to-text service.

I recently found myself working on a chess play automation project. The user would issue a voice command describing the location of the piece they want to move and their desired destination location, the data would then processed and a chess move made. This was so cool and being the R-nista that I am (i don't think this is a word, i have only come across Pythonistas) ,I thought to myself, maybe this can also be done in R and voila, this article was born. For any feedback feel free to reach out at Ian

1.1 Recording Audio in R using the audio Package

We begin with the input which we plan to transcribe. In order to record audio,we need to install and load the audio package

```
#install the audio package
install.packages("audio")
```

We then proceed to setup the audio driver which we will use to record as shown below:

```
#Load the library
library(audio)
#Check which audio driver is set
current.audio.driver()
```

```
#if there is no driver set, view what drivers are available
audio.drivers()
```

```
#from the list provided, set which driver to use (default driver is always best)
set.audio.driver(NULL)# sets the default audio driver
#option 2
set.audio.driver (insert_name_here ) #sets the audio driver to a driver other than the def
```

```
#Checks and verifies that indeed the audio driver is set as per the command above
current.audio.driver()
```

1.1.1 Recording Audio

Some of the key parameters in audio processing which we are going to encounter during this exercise include:

Sample rate: This is the number of times per second a sound is sampled and recorded.Therefore, if we use a sampling rate of say 8000 Hertz, a recording with a duration of 5 seconds will contain 40,000 samples (8000 * 5). The industry standard sampling rate commonly used is 44100 Hertz.

Mono vs Stereo: If you are a sound enthusiast, you've probably come across these terms. Simply put, Mono sound is recorded and played back using only *one audio channel* e.g. a guitarist recording using one mic to pick up sound of the guitar and Stereo sound is recorded and played through *more than one channel.*

In our case we will use one audio channel(mono) and a sampling rate of 44100Hz. That being said let's start our recording

```
#Set our recording time
rec_time <- 5

#Recording
Samples<-rep(NA_real_, 44100 * rec_time) #Defining number of samples recorded
print("Start speaking")
audio_obj <-record(Samples, 44100, 1) #Create an audio instance with sample rate 44100 and
wait(6)
rec <- audio_obj$data # get the recorded data from the audio object

#Save the recorded audio
file.create("sample.wav")#gets created in your current working directory
save.wave(rec,"Insert_path_to_sample.wav_here")
```

On recording and saving the audio file to `sample.wav` , we have to clear the audio instance object `audio_obj` before proceeding to make the next recording. From the audio package

documentation, this is achieved by using the `close(con,…)` method, where `con` is the audioInstance object .However, using this method proved to be cumbersome as it causes the console to freeze anytime you want to record audio for more than one time. After doing some research, I discovered that restarting the console clears the audio instance object therefore allowing for one to record audio multiple times with no issue. I know this is not an elegant solution (more like tying duct tape around a leaking pipe)and i am actively looking for a better solution to fix this issue. For now, we go by the saying: if it works, don't touch it and implement this step to clear the audio instance object.

```
.rs.restartR() # clear the audio instance object(looking for a more elegant solution)
wait(3)
```

Play the recording we just created just to confirm that indeed we recorded something.

```
play(rec)
```

That's it for our input. We now proceed to process this and transcribe the audio we just recorded

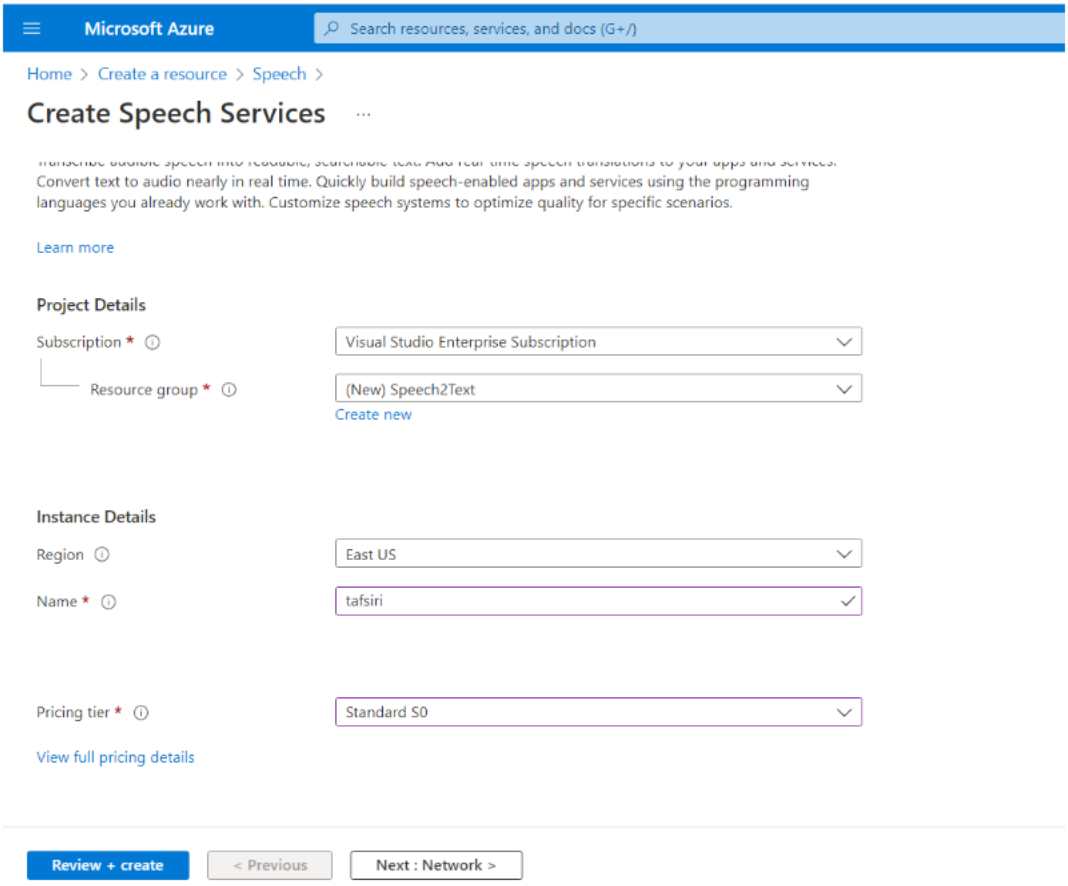
1.2 House Keeping on Azure

First we begin with some light housekeeping, i.e setting up a Cognitive Services resource in our Azure subscription. You can create a free Azure account here and if you have an account already, you can skip this step.

We then proceed to create a cognitive resource group by following these steps:
1. Open the Azure portal `https://portal.azure.com` and sign in using your Microsoft account.
2. Click **+ Create a resource** and on the search bar, type **Speech**, click **create** and create a Cognitive service resource with the following settings:

- Subscription: *Enter your Azure subscription*
- Resource group: *Create one with a unique name or an existing one*
- Region: *Choose any*
- Name: *Enter a unique name*
- Pricing tier: *Standard S0*
- Select I have read and understood the notices
- Click on **Review + create** and we are good to go.

As a guide, these are the settings I used however, you can go ahead and get creative with the names.



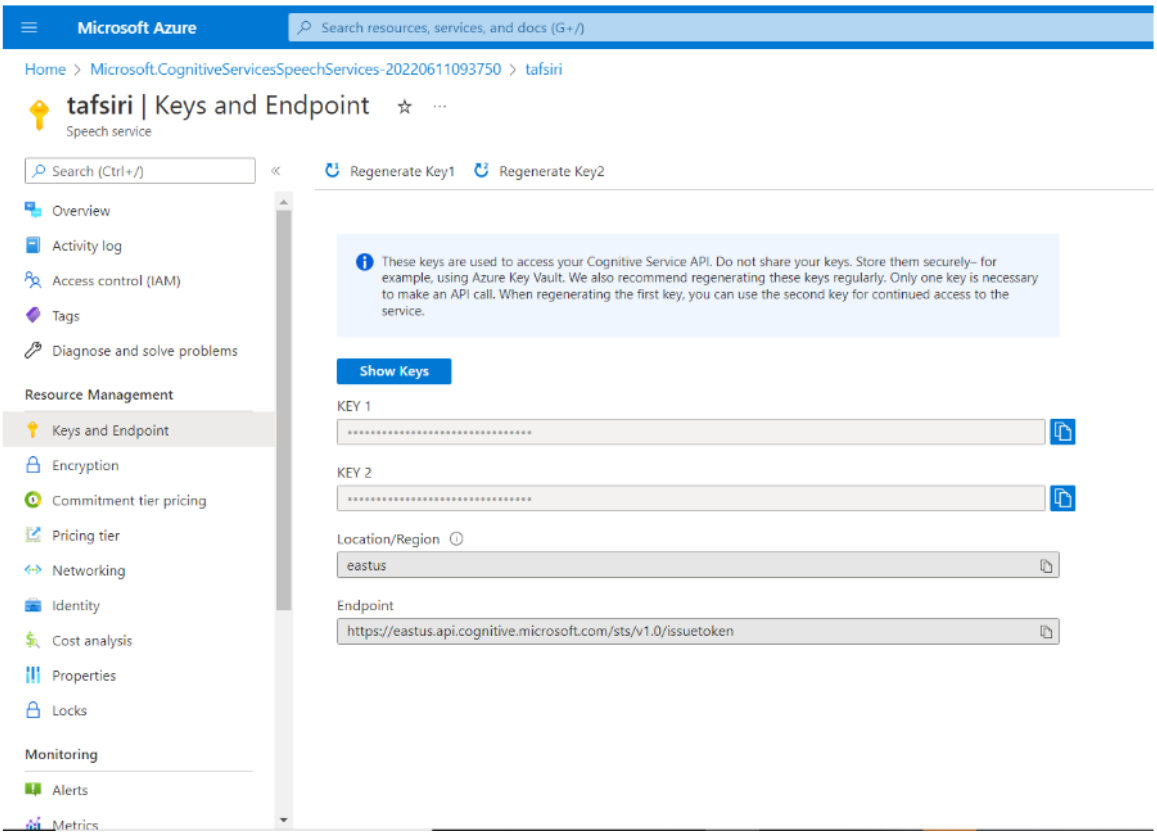
1.3 Speech-to-text API call

Since now we have our audio sample input `sample.wav` ,we can now go ahead to the transcribing bit. It is noteworthy that the speech to text rest API for short audio has several limitations which are:

- Requests cannot contain more than 60 seconds of audio. For batch transcriptions and custom speech use Speech to Text API v3.0
- The API does not provide partial results.
- It does not support Speech translation. For translation, you can checkout the Speech SDK

Now that we are done with that, we can then go ahead and get coding.

`Note:` To access the speech service we just created from R, we need the **URL of the endpoint**, **location/region details** and **KEY 1 & 2** parameters. These can be found in the azure portal under **Keys and Endpoint** page of your cognitive service resource as illustrated below:



Copy these details and save them since we are going to need them when make our API call. Note: you can use either **KEY 1** or **KEY 2** as your subscription key

To begin our transcription, install the packages we are going to need if you don't have them already installed:

```
#Enables us to work with HTTP in R
install.packages("httr")
install.packages("jsonlite")
```

Note You will have to modify the subscription key, language and data path parameters to match the ones on the Cognitive Services resource you created earlier. Also modify the URL `'https://eastus.stt.speech.microsoft.com/speech/recognition/conversation/cognitiveservices/v` by inserting the aforementioned location details as shown in the modified URL :

'https://_ENTER_LOCATION_HERE_.stt.speech.microsoft.com/speech/recognition/conversation/cogni
. At the end you should have something like this:

```
library(httr)
library(jsonlite)
#Documentation on the two packages can be found by running ?httr and ?jsonlite commands re

#Define headers containing subscription key and content type
headers = c(
  `Ocp-Apim-Subscription-Key` = '_ENTER_YOUR_SUBSCRIPTION_KEY_HERE',  #Key 1 or 2
  `Content-Type` = 'audio/wav' #Since we are transcribing a WAV file
)

#Create a parameters list, in this case we specify the languange parameter
params = list(
  `language` = 'en-US'
)

#Enter path to the audio file we just recorded and saved
data = upload_file ('Insert_path_to_sample.wav_here')

#Make the API call and save the response recived
response <- httr::POST(url = 'https://eastus.stt.speech.microsoft.com/speech/recognition/c
httr::add_headers(.headers=headers), query = params, body = data)

#Convert response received to a dataframe
result <- fromJSON(content(response, as  = 'text'))
txt_output <- as.data.frame(result)

#Extract transcribed text
txt_input <- txt_output[1,2]
txt_input
```

And with that, we have successfully recorded audio and transcribed it with the aid of Microsoft Azure Cognitive Services. The applications of the cognitive services are wide and this is just but a glimpse of what one can accomplish using the Speech to Text service. I'll leave it at that. Thank you foR youR time. Cheers.

2 Manipulator kinematics

Eric Wanjau and Ian Muchiri

This Notebook illustrates how the manipulator will arrive at the desired location as instructed by the speech input.

Kinematics is the science of motion that treats the subject without regard to the forces that cause it.

2.0.1 Forward Kinematics

Forward kinematics addresses the problem of computing the **position and orientation of the end effector** relative to the user's workstation given the joint angles of the manipulator.

The forward kinematics were performed in accordance with the Denavit—Hartenberg (D-H) convention. According to the D-H convention, any robot can be described kinematically by giving the values of four quantities, typically known as the D-H parameters, for each link. The link length a and the link twist αquantities, describe the link itself and the remaining two; link offset d and the joint angle θ describe the link's connection to a neighboring link.

To perform the manipulator kinematics, link frames were attached to the manipulator as shown in Figure 2.2. In summary, link frames are laid out as follows:

1. The $z-axis$ is in the direction of the joint axis.
2. The $x-axis$ is parallel to the common normal.
3. The $y-axis$ follows from the x-axis and z-axis by choosing it to be a right-handed coordinate system.

Once the link frames have been laid, the D-H parameters can be easily defined as:

d: offset along the previous z to the common normal.

θ: angle about previous z, from old x to new x.

a: length of the common normal.

α: angle about common normal from old z $axis$ to new z $axis$.

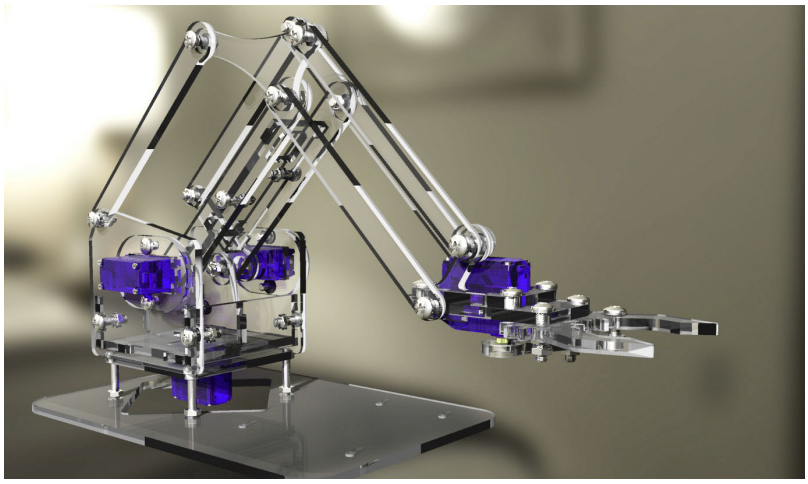


Figure 2.1: meArm parallel-link manipulator

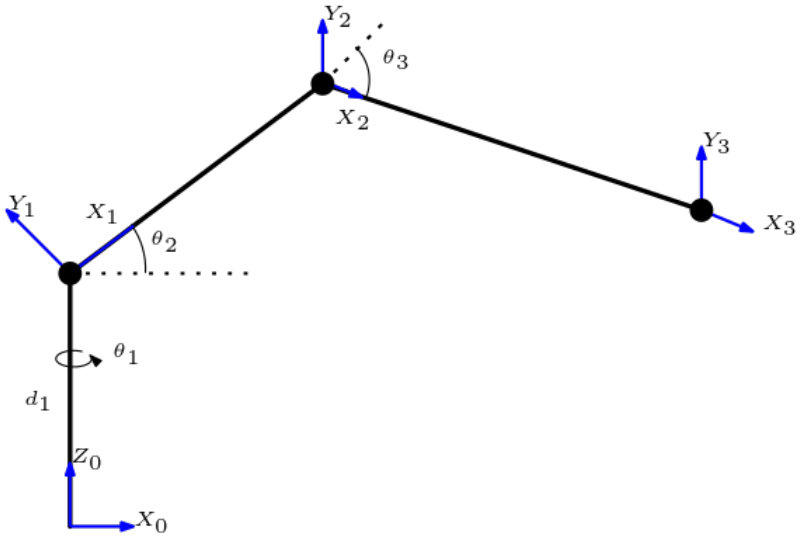


Figure 2.2:

The D-H parameters for the meArm were evaluated as below:

Table 2.1: D-H parameters for meArm

Link	θ	$a(mm)$	α	$d(mm)$
1	θ_1	0	90	55
2	θ_2	80	0	0
3	θ_3	120	0	0

In this convention, each homogeneous transformation A_i is represented as a product of the four basic transformations (Spong, Hutchinson, and Vidyasagar 2005), which evaluates to a 4×4 matrix that is used to transform a point from frame n to $n\ -1$.

$$A_i = Rot_{z,\theta_i}\ Trans_{x,a_i}\ Rot_{x,\alpha_i}$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}c_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Considering the three links of the meArm, the total homogeneous transformation will be a product of the transformations of the three links given as:

$$A_T = A_1 \times A_2 \times A_3 \tag{2.1}$$

The final total homogeneous transformation was derived to be:

$$A_t = \begin{bmatrix} cos(\theta_2+\theta_3)cos(\theta_1) & -sin(\theta_2+\theta_3)cos(\theta_1) & sin(\theta_1) & 4\sigma_1 cos(\theta_1) \\ cos(\theta_2+\theta_3)sin(\theta_1) & -sin(\theta_2+\theta_3)sin(\theta_1) & -cos(\theta_1) & 4\sigma_1 sin(\theta_1) \\ sin(\theta_2+\theta_3) & cos(\theta_2+\theta_3) & 0 & 12sin(\theta_2+\theta_3)+8sin(\theta_2)-\frac{11}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

where

$$\sigma_1 = 3\ cos(\theta_2 + \theta_3) + 2\ cos(\theta_2)$$

The position and orientation of the end effector x, y, z of the meArm was consequently obtained from the total homogeneous $A_t$transformation using the upper right 3x1 matrix as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4\cos(\theta_1)\ (3\ cos(\theta_2 + \theta_3) + 2\ cos(\theta_2)) \\ 4\sin(\theta_1)\ (3\ cos(\theta_2 + \theta_3) + 2\ cos(\theta_2)) \\ 12sin(\theta_2 + \theta_3) + 8sin(\theta_2) - \frac{11}{2} \end{bmatrix} \tag{2.3}$$

> **Tip**
>
> See this paper for a thorough derivation of the above.

That said, let's put this into code:

```
# Function that calculates forward kinematics
fkin <- function(motor_angles){

  # Convert to radians
  angles = motor_angles * pi/180

  # Extract angles
  theta1 = angles[1]
  theta2 = angles[2]
  theta3 = pi - angles[3]

  # Calculate x, y, z
  x <- 4 * cos(theta1) *( (3*cos(theta2 + theta3)) + (2*cos(theta2)))

  y <- 4 * sin(theta1) *( (3*cos(theta2 + theta3)) + (2*cos(theta2)))

  z <- (12 * sin(theta2 + theta3)) + (8*sin(theta2)) - (11/2)


  # Return a tibble
  fkin <- tibble(

    orientation = c("x", "y", "z"),

    # Multiply by -1 to re-orient y and z
    # mistake made during finding DH
```

```
    position = round(c(x, y * -1, z * -1))

  )

  return(fkin)



}
```

What would be the x, y, z coordinates of the end effector when rotation of motors $1, 2, 3$ are $90, 113, 78$ degrees respectively?

```
# Calculate forward kinematics
fkin(motor_angles = c(90, 113, 78))
```

```
# A tibble: 3 x 2
  orientation position
  <chr>          <dbl>
1 x                  0
2 y                 13
3 z                  5
```

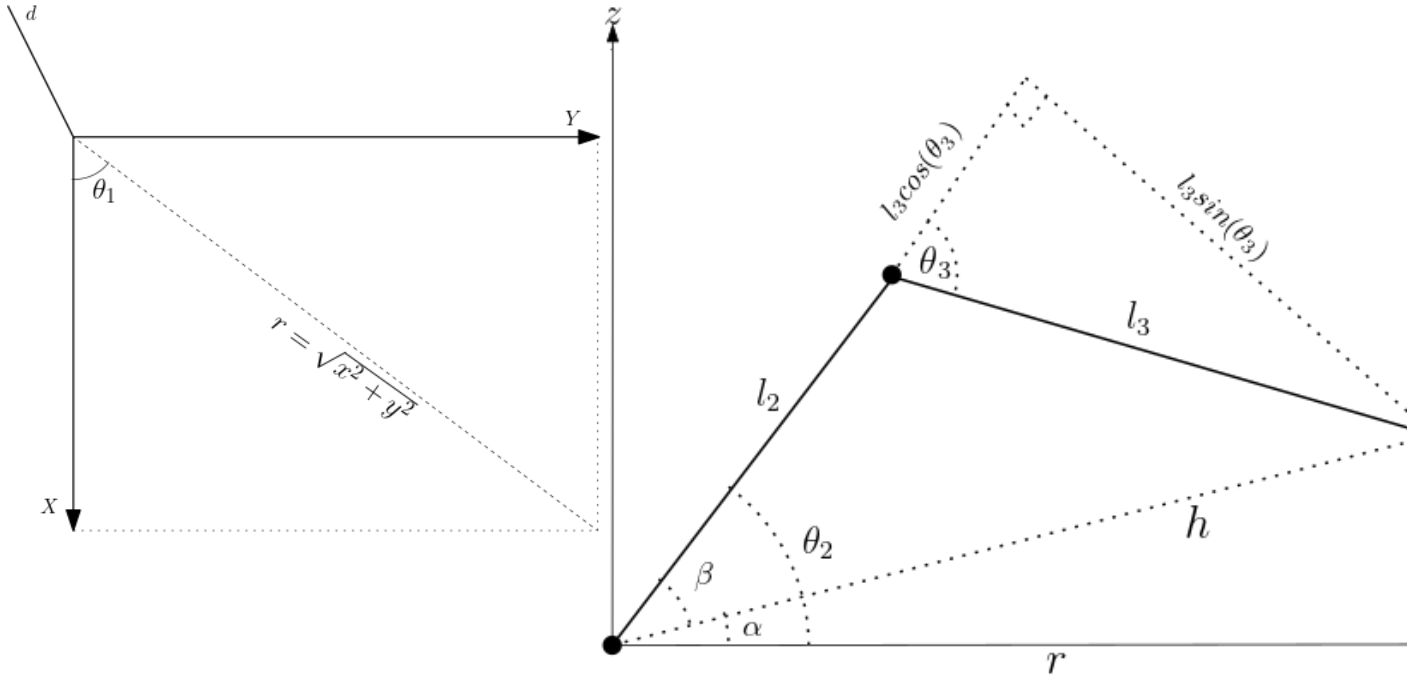
```
fkin(motor_angles =c(42, 110, 114))
```

```
# A tibble: 3 x 2
  orientation position
  <chr>          <dbl>
1 x                -11
2 y                 10
3 z                 -3
```

2.0.2 Inverse kinematics

Inverse kinematics addresses the more difficult converse problem of computing the **set of joint angles** that will place the end effector at a desired position and orientation. It is the computation of the manipulator joint angles given the position and orientation of the end effector.

In solving the inverse kinematics problem, the Geometric approach was used to decompose the spatial geometry into several plane-geometry problems based on the sine and the cosine rules. This was done by considering the trigonometric decomposition of various planes of the manipulator as graphically illustrated below:



$$\theta_1 = tan^{-1}\frac{y}{x} \tag{2.4}$$

With the hypotenuse r, connecting x and y obtained using the Pythagoras theorem as

$$r = \sqrt{x^2 + y^2}$$

The angles θ_2 θ_3 were obtained by considering the plane formed by the second and third links as illustrated:

$$\theta_2 = \alpha + \beta\theta_2 = tan^{-1}\frac{s}{r} + tan^{-1}\frac{l_3 sin(\theta_3)}{l_2 + l_3 cos(\theta_3)} \tag{2.5}$$

$$\theta_3 = cos^{-1}\frac{x^2 + y^2 + s^2 - l_2^2 - l_3^2}{2l_2 l_3} \tag{2.6}$$

Where s is the difference between the distance of the end effector from the base and the offset:
$$ s = z- d

$$

Again, let's put the above into code

```
ikin <- function(xyz_coordinates){

  # Extract xyz coordinates
  x = xyz_coordinates[1]
  y = xyz_coordinates[2]
  z = xyz_coordinates[3]

  # Account for manipulator moving right or left
  if (x >= 0){
    theta1 = atan(x/y) + pi/2
  } else {
    theta1 = atan(y/x) %>% abs()
  }

  # Calculate theta 3 since its needed in theta 2
  theta3 = acos((x^2 + y^2 + (z-5)^2 - 8^2 - 12^2) / (2*8*12))
    # 8 and 12 are the dimensions of manipulator arms


  # Calculate theta 2
  theta2 = atan((5.5 - z) / (sqrt(x^2 + y^2)) ) + atan((12 * sin(theta3)) / (8 + 12*cos(th

  if(theta2 > 0){
    theta2 = pi - abs(theta2)
  }

  tbl <- tibble(
    ef_position = c(x, y, z),
    motor_angles = (c(theta1, theta2, pi-theta3)*180/pi) %>% round()
  )

  return(tbl)

}
```

Theoretically, the results from inverse kinematics and forward kinematics should be in tandem for a given set of inputs. Let's see whether we can get our previous joint angles from the results of the forward kinematics.

```r
# Calculate inverse kinematics
ikin(xyz_coordinates = c(0, 13, 5))
```

```
# A tibble: 3 x 2
  ef_position motor_angles
        <dbl>        <dbl>
1           0           90
2          13          113
3           5           78
```

```r
ikin(xyz_coordinates = c(-11, 10, -3))
```

```
# A tibble: 3 x 2
  ef_position motor_angles
        <dbl>        <dbl>
1         -11           42
2          10          110
3          -3          114
```

2.1 Summary

From the above examples, the xyz coordinate values obtained from forward kinematics operation produced the same input angles when passed through the inverse kinematics equations. This will be essential for future operations such as validating whether motor angle rotations result in a desired xyz position of the end effector.

And with that, this section is done! Please do feel free to reach out in case of any questions, feedback and suggestions.

Happy LeaRning,

Eric.

3 Board mapping

Eric Wanjau and Ian Muchiri

This notebook illustrates how we assigned real world board coordinates (cm) to each square box on the chess board.

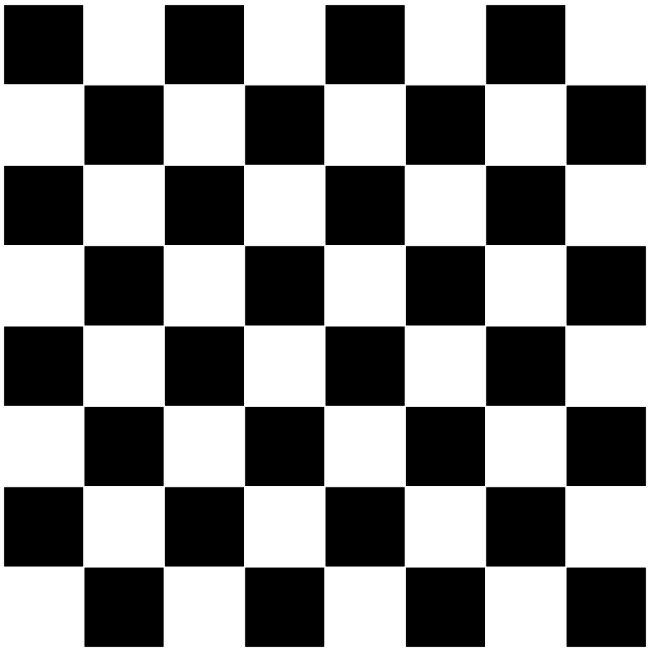
The approach was quite straightforward. We created a virtual board and then translated virtual coordinates to the manipulator's workspace.

```
library(here)
library(waffle)
library(patchwork)
library(magick)
library(tidyverse)

# Vector
x <- c(A= 1, B = 1, C = 1, D = 1, E = 1, F = 1, G = 1)
y <- c(A= 1, B = 1, C = 1, D = 1, E = 1, F = 1, G = 1)

# Create checker boxes
w1 = waffle(x, rows = 8, flip = TRUE, colors = c("black", "white", "black", "white", "blac
  theme(plot.margin = margin(0, 0, 0, 0))
w2 = waffle(y, rows = 8, flip = TRUE, colors = c("white", "black", "white", "black", "whit
  theme(plot.margin = margin(0, 0, 0, 0))

# Make checker board
checkerboard <- w1 / w2 / w1 / w2 / w1 / w2 / w1 / w2
checkerboard
```



```
ggsave("images/checkerboard.png", width = 7, height = 7)
```

3.1 Convert image into a tibble

```
library(magick)
img <- image_read("images/checkerboard.png") %>%
  image_convert(type = "grayscale")
# Dimensions of virtual board
dim_x = 160 * 4
dim_y = 160 * 4
img <- image_resize(img, paste(dim_x, dim_y, sep = "x"))


# Create row and column identifiers
row_names <-  tibble(x = 1:dim_x, y = rep(LETTERS[1:8], each = dim_y/8), z = paste(x, "_",

col_names <-  tibble(x = 1:dim_x, y = rep(1:8, each = dim_x/8), z = paste(x, "_", y, sep =

# Create array and number rows and columns
```

```r
img_array <- drop(as.integer(pluck(img, 1)))
rownames(img_array) <- row_names
colnames(img_array) <- col_names


# Create data frame from array and rename column
img_dfx <- img_array %>%
  as_tibble() %>%
  mutate(y = row_names) %>%
  #rowid_to_column(var = "y") %>%
  pivot_longer(!y, names_to = "x", values_to = "pv") %>%
  mutate(pv = scales::rescale(pv, to = c(0, 1))) %>%
  # binarize image
  mutate(pv = case_when(
    pv > 0.5 ~ 1,
    TRUE ~ 0)) %>%
  separate(y, c("y", "pl")) %>%
  separate(x, c("x", "pn")) %>%
  mutate(pos = paste(pl, pn, sep = "")) %>%
  select(-c(pn, pl)) %>%
  mutate(across(c(y, x, pv), as.numeric)) %>%
  group_by(pos) %>%
  mutate(centroidx = round(mean(x)), centroidy = round(mean(y))) %>%
  ungroup()
```

Only centroid locations are of importance. So we narrow down to that:

```r
# Obtain location centroids
centroids <- img_dfx %>%
  ungroup() %>%
  distinct(pos, centroidx, centroidy)

# View centroids on the virtual board
centroids %>%
  slice_head(n = 10)
```

```
# A tibble: 10 x 3
   pos   centroidx centroidy
   <chr>     <dbl>     <dbl>
 1 A1           40        40
 2 A2          120        40
 3 A3          200        40
```

```
 4 A4          280        40
 5 A5          360        40
 6 A6          440        40
 7 A7          520        40
 8 A8          600        40
 9 B1           40       120
10 B2          120       120
```

3.2 Mapping coordinates to xy

Once the centroid pixel coordinates were successfully extracted, the next step involved translating them to their corresponding real-world x-y coordinates in cm. This scaling from pixel values to cm values was achieved by the mapping function or simply using `scales::rescale()`:

```r
# Mapping function
map_fun <- function(value, from_low, from_high, to_low, to_high){

  mapped_val = (value - from_low) * (to_high - to_low) / (from_high - from_low) + (to_low)

  return(mapped_val)
}


# Map virtual board coordinates to manipulator workspace
centroids = centroids %>%
    # mutate(x_mapped = map_fun(centroidx, from_low = 0, from_high = dim_x, to_low = 9, to
    #        y_mapped = map_fun(centroidy, from_low = 0, from_high = 192, to_low = 11, to_

    ## Using scales::rescale
    mutate(x_mapped = scales::rescale(centroidx, to = c(9, -9), from = c(0, 640)),
           y_mapped = scales::rescale(centroidy, to = c(11, 17), from = c(0, 192))) %>%
    mutate(across(where(is.numeric), round)) %>%
    # Rearrange board positions to match our physical chess board
    mutate(
      pl = rep(LETTERS[1:8], times = nrow(centroids)/8),
      pn = rep(1:8, each = nrow(centroids)/8),
      pos = paste(pl, pn, sep = "")) %>%
    select(-c(pl, pn))

# View centroids coordinates
centroids %>%
```

```
    slice_head(n = 10)
```

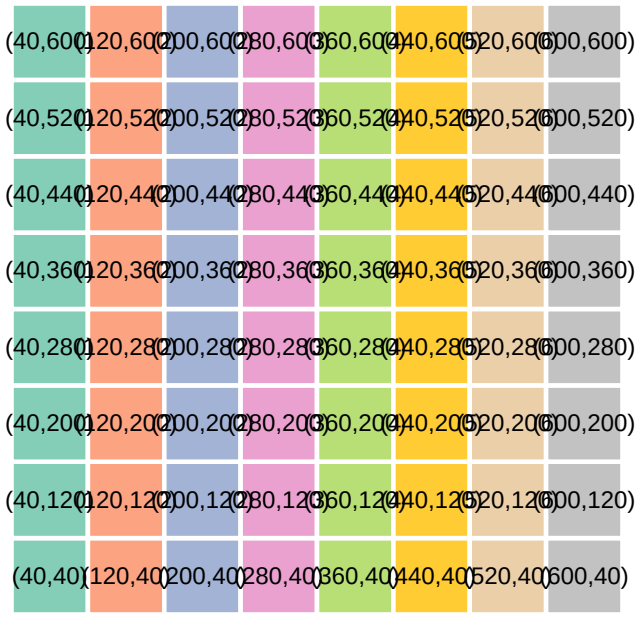
```
# A tibble: 10 x 5
   pos   centroidx centroidy x_mapped y_mapped
   <chr>     <dbl>     <dbl>    <dbl>    <dbl>
 1 A1           40        40        8       12
 2 B1          120        40        6       12
 3 C1          200        40        3       12
 4 D1          280        40        1       12
 5 E1          360        40       -1       12
 6 F1          440        40       -3       12
 7 G1          520        40       -6       12
 8 H1          600        40       -8       12
 9 A2           40       120        8       15
10 B2          120       120        6       15
```

Now let's visualize our handy work:

```
theme_set(theme_void())

# Create virtual board
centroids %>%
  ggplot(mapping = aes(x = centroidx*2, y = centroidy*2)) +
  geom_tile(aes(fill = str_extract(pos, "[:alpha:]")), color = "white", alpha = 0.8, size
  geom_text(aes(x = centroidx*2, y = centroidy*2 + 2, label = pos), color = "black", size
  coord_equal() +
  #paletteer::scale_fill_paletteer_d("RColorBrewer::Set2") +
  scale_fill_manual(values = paletteer::paletteer_d("RColorBrewer::Set2") %>% paste() %>%
```


```
# Virtual coordinates on board
centroids %>%
  ggplot(mapping = aes(x = centroidx*2, y = centroidy*2)) +
  geom_tile(aes(fill = str_extract(pos, "[:alpha:]")), color = "white", alpha = 0.8, size
  geom_text(aes(x = centroidx*2, y = centroidy*2 + 2, label = paste("(", centroidx, ",", c
  coord_equal() +
  #paletteer::scale_fill_paletteer_d("RColorBrewer::Set2") +
  scale_fill_manual(values = paletteer::paletteer_d("RColorBrewer::Set2") %>% paste() %>%
```




```
# Real world coordinates on board
centroids %>%
  ggplot(mapping = aes(x = centroidx*2, y = centroidy*2)) +
  geom_tile(aes(fill = str_extract(pos, "[:alpha:]")), color = "white", alpha = 0.8, size
  geom_text(aes(x = centroidx*2, y = centroidy*2 + 2, label = paste("(", x_mapped, ",", y_
  coord_equal() +
  #paletteer::scale_fill_paletteer_d("RColorBrewer::Set2") +
  scale_fill_manual(values = paletteer::paletteer_d("RColorBrewer::Set2") %>% paste() %>%
```

(8,30)	(6,30)	(3,30)	(1,30)	(−1,30)	(−3,30)	(−6,30)	(−8,30)
(8,27)	(6,27)	(3,27)	(1,27)	(−1,27)	(−3,27)	(−6,27)	(−8,27)
(8,25)	(6,25)	(3,25)	(1,25)	(−1,25)	(−3,25)	(−6,25)	(−8,25)
(8,22)	(6,22)	(3,22)	(1,22)	(−1,22)	(−3,22)	(−6,22)	(−8,22)
(8,20)	(6,20)	(3,20)	(1,20)	(−1,20)	(−3,20)	(−6,20)	(−8,20)
(8,17)	(6,17)	(3,17)	(1,17)	(−1,17)	(−3,17)	(−6,17)	(−8,17)
(8,15)	(6,15)	(3,15)	(1,15)	(−1,15)	(−3,15)	(−6,15)	(−8,15)
(8,12)	(6,12)	(3,12)	(1,12)	(−1,12)	(−3,12)	(−6,12)	(−8,12)

Here's an example of how we would extract the real word coordinates from the tibble:

```
# Function to return mapped values from tibble
get_centroid <- function(position){
  x = centroids %>%
    filter(str_detect(pos, position)) %>%
    pull(x_mapped)

  y = centroids %>%
    filter(str_detect(pos, position)) %>%
    pull(y_mapped)

  return(c(x, y))
}

# Define initial and final positions
initial_pos <- "B2"
final_pos <- "G2"
get_centroid(position = initial_pos)
```

```
[1]  6 15
```

```
get_centroid(position = final_pos)
```

```
[1] -6 15
```

3.3 Summary

This is the approach we took at this step. We created a virtual board and then translated virtual coordinates to the manipulator's workspace.

And with that, this section is done! Please do feel free to reach out in case of any questions, feedback and suggestions.

Happy Learning,

Eric.

4 R and Arduino

Eric Wanjau and Ian Muchiri

This notebook briefly describes how we set up a communication interface between R and a microcontroller (Arduino). Most of it is based on a blog post we wrote a while back: *What we R about when we R about R and Arduino*.

`Arduino` is an open-source electronics platform based on easy-to-use hardware (`Arduino Board`) and software (`Arduino IDE`). One can tell the board what to do if one has the correct form of data and a set of instructions for processing the data and performing subsequent operations. The Arduino's microcontroller is responsible for holding all your compiled code and executing the commands you specify. The Arduino Software on the other hand is the board's IDE where one writes the set of instructions governing the board. The getting started guide would be a good place to start learning about the Arduino ecosystem.

Switching over to R, we couldn't have found better words to summarize what `R` is than with these words found in the book Advanced R by `Hadley Wickham`: Despite its sometimes frustrating quirks, R is, at its heart, an elegant and beautiful language, well tailored for data science .

With all this said, a fine convergence can be struck between the two: `data`. Consider this very simple example. We want the Arduino board to turn an LED (Light Emitting Diode) ON once it receives a `1` and OFF once it receives a `0`. If one can get a way of sending some data (1 or 0) to the board's microcontroller, then, the set objective will be achieved sooner or later.

4.0.1 Setting up a serial connection between R and Arduino

Serial communication is the communication protocol that will be used between R and the Arduino software similar to what is used in the Arduino serial monitor. This communication interface will facilitate the transmission of data between the two interfaces.

The serial package (Seilmayer 2020) will be used to set up this comunication interface.

Let's begin by loading the required packages.

```
library(tidyverse)
library(serial)
library(here)
```

Next, we'll create a serial port object called `arduino`, which represents a serial client for communication with the USB serial port where our board is connected.

```
# See the ports available
listPorts()
```

Create an Arduino object and set up the interface parameters.

This is achieved using the `serial::serialConnection` function. The interface parameters are such that the baud rate (specifies the number of bits being transferred per second) is set to `9600`, which is the same value in the Arduino script. Also, we have specified that the transmission ends with a new line and that the transmission is complete if the end of line symbol is the `carriage return cr`.

```
arduino <-  serialConnection(name = "aRduino",
                             port = "COM5",
                             mode = "9600,n,8,1" ,
                             buffering = "none",
                             newline = TRUE,
                             eof = "",
                             translation = "cr",
                             handshake = "none",
                             buffersize = 8096

                             )
```

Now that the serial interface is in place, the next step is initialising the interface and keeping it open for later usage such as writing and reading data from it. Once `serial::isOpen` initialises the interface, the Arduino board blinks. This is because the board resets once a serial port is opened to allow the bootloader to receive a new sketch.

`serial::isOpen` tests whether the connection is open or not.

```
open(arduino)

# testing whether the connection is open or not
isOpen(arduino)
```

4.0.2

4.0.3 Writing data from RStudio to the serial interface

At this point, we are all set to write some data to the serial interface.

Let's prepare some data to send to the serial interface.

```
## Create dummy data
n = 42
arduino_input <- tibble(
  c = sample(10:100, size = n, replace = T) %>%
                      paste('C', sep = ''))
```

The chunk below uses `serial::write.serialConnection()` to write the LED values to the serial port row by row.

```
close(arduino)
open(arduino)
Sys.sleep(3)
for (r in 1:nrow(arduino_input)){
  Sys.sleep(0.3)
  write.serialConnection(arduino, paste(arduino_input[r,], collapse = ''))
}
Sys.sleep(2)
```

4.0.4 Driving the manipulator's servo motors

The Arduino board itself is a programmable platform. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language and the Arduino Software (IDE). Here are sample instructions that were uploaded to the microcontroller:

```
if(Serial.available()){ // checks data in serial

  static int t=0;

    char mychar=Serial.read(); // reads serial data

    switch(mychar){
```

```
    case '0'...'9':
      t=t*10 + mychar - '0'; // parse integers
      break;
    case 'A':
      {
          servoA.write(t,50,1); // write value to motor
          Serial.println(t); // print data to serial
      }
      t=0;
      break;
      ...
  }
```

On a very high level, the microcontroller:

- Checks whether data is available on the serial interface.
- Reads the data one byte at a time
- Uses a series of switch case commands to
  - Parse integers
  - Write motor angles (send an electrical signal) to each respective motor based on the motor's tag e.g `A`, `B` or `C`

So how do the servo motors actually rotate? Servos are controlled using adjustable pulse widths on the signal line. This is achieved using a technique called `Pulse Width Modulation`. PWM is a modulation technique that generates variable-width pulses to represent the amplitude of an analog input signal.

For a standard servo, sending a `1 ms 5V` pulse turns the motor to `0` degrees, and sending a `2 ms 5V` pulse turns the motor to `180` degrees, with pulse lengths in the middle scaling linearly. A `1.5 ms` pulse, for example, turns the motor to 90 degrees. Once a pulse has been sent, the servo

turns to that position and stays there until another pulse instruction is received. However, if you want a servo to "hold" its position (resist being pushed on and try to maintain the exact position), you just resend the command once every 20 ms. The Arduino servo commands e.g `servo.write` takes care of all this for you. To better understand how servo control works, please see the timing diagram:

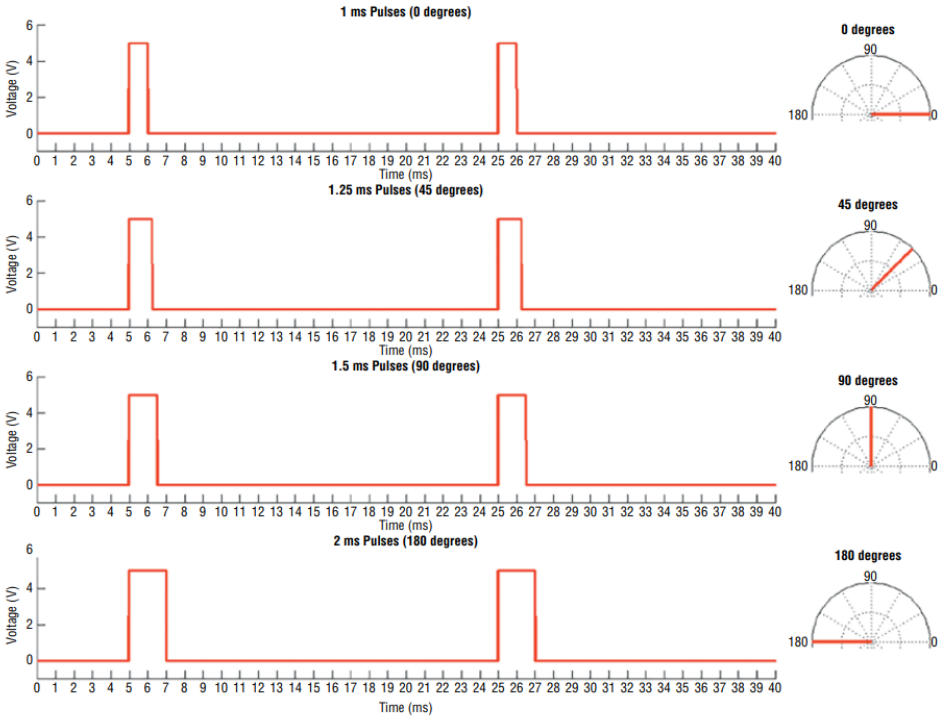


Figure 4.1: Servo motor timing diagram: Jeremy Blum - Exploring Arduino

A great place to get started with Arduino and some hobby electronics projects would be Blum (2013).

4.0.5 Reading data sent from Arduino board

We can read the values sent to the serial port connection by Arduino script using `read.serialConnection()`

```
data_frm_arduino <- tibble(capture.output(cat(read.serialConnection(arduino)))) %>%
  filter(if_any(where(is.character), ~ .x != ""))
```

```
data_frm_arduino
```

4.0.6 Summary

There we go! In this section, we leveraged Arduino's capability to be programmed via a serial interface to send and receive data from R to the Arduino board.

Please do feel free to reach out in case of any questions, feedback and suggestions.

Happy Learning,

Eric.

5 Summary

Awesome! That's how R played a role at each step of our project. You can find how we combined the code chunks from these four chapters and the user defined functions we created along the way to make our manipulator move: R

The instructions on the microcontroller (described in chapter 4) that send electrical pulses to rotate our motors can be found in the file robot_arm_control.ino.

And that's it! There goes a touch of R in robotics.

Please do feel free to reach out in case of any questions, feedback and suggestions,

Eric and Ian.

References

Blum, J. 2013. *Exploring Arduino: Tools and Techniques for Engineering Wizardry*. EEET 2046 : Electronics. Wiley. https://books.google.co.uk/books?id=iXsQAAAAQBAJ.

Seilmayer, Martin. 2020. "Serial: The Serial Interface Package." https://CRAN.R-project.org/package=serial.

Spong, M. W., S. Hutchinson, and M. Vidyasagar. 2005. *Robot Modeling and Control*. Wiley.