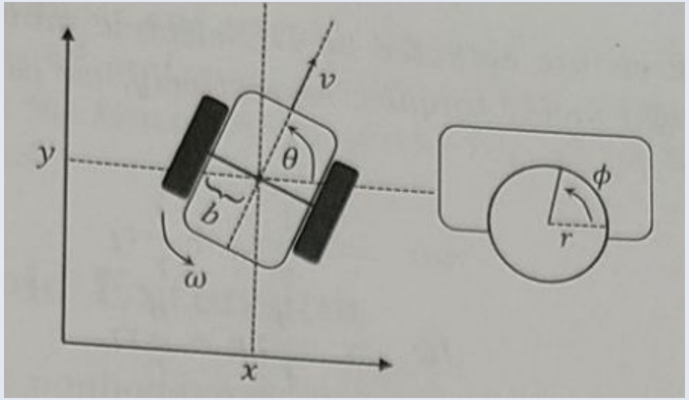
Differential Drive Mobile Robot Final Project

By: Audrey Warrene and Leo Doak

Outline

Differential Drive Robot Review

- A DDR is a mobile robot system that controls its movement through the differential speed and direction of its two drive wheels.
- Known for its simplicity and maneuverability.

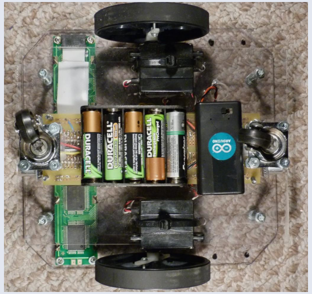


- Our project originally focused on seeing how altering the acceleration of each individual wheel on our robot affects the velocity, heading and position of our system.

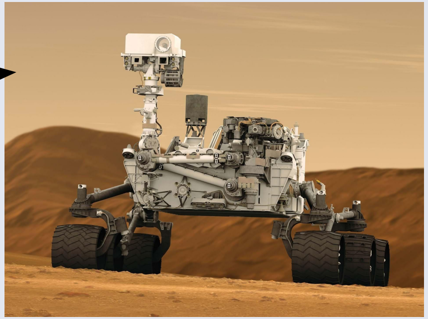
Real-Life Scenarios

Can be used in many fields:

- Warehouse Automation → Helps deliver goods to/from different sections
- Agricultural Robots → For crop monitoring and maintenance
- Space Exploration
- Hazardous environment exploration
- Educational Purposes
- AND MORE!



[6]



[8]

[7]

[13] and [14]

What each variable is:

- x, y: the horizontal and vertical positions of the center of the robot
- Θ: the angular position (heading) of the robot
- v: the forward velocity
- w: the angular velocity
- r: radius of the wheel
- b: distance of each wheel to the center of the robot
- theta(l) and theta(r): Angular accelerations of left and right wheel, also the system inputs

Formulas Given:

$$\dot{\mathbf{x}} = \cos(\theta)v$$

$$\dot{\mathbf{y}} = \sin(\theta)v$$

$$\dot{\theta} = w$$

$$\dot{\mathbf{v}} = \frac{r}{2}(\ddot{\emptyset}_R + \ddot{\emptyset}_L)$$

$$\dot{\mathbf{w}} = \frac{r}{2b}(\ddot{\emptyset}_R - \ddot{\emptyset}_L)$$

Predetermined Values:

$\rightarrow r$: .02 meters

$\rightarrow b$: .05 meters

Discrete Time Format

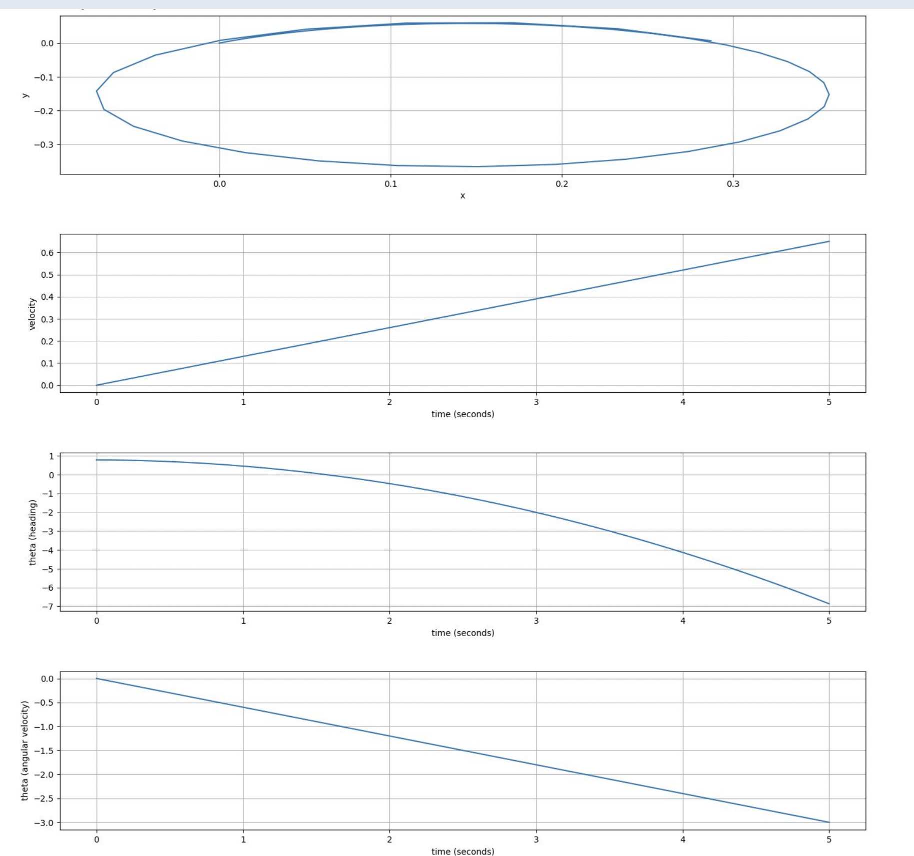
The discrete-time format is:

$$
\begin{aligned}
x_{k+1} &= \Delta t(cos(\theta_k)v_k) + x_k \\
y_{k+1} &= \Delta t(sin(\theta_k)v_k) + y_k \\
\theta_{k+1} &= \Delta t(w_k) + \theta_k \\
v_{k+1} &= \Delta t(\tfrac{r}{2}(\ddot{\emptyset}_R + \ddot{\emptyset}_L)) + v_k \\
w_{k+1} &= \Delta t(\tfrac{r}{2b}(\ddot{\emptyset}_R - \ddot{\emptyset}_L)) + w_k
\end{aligned}
$$

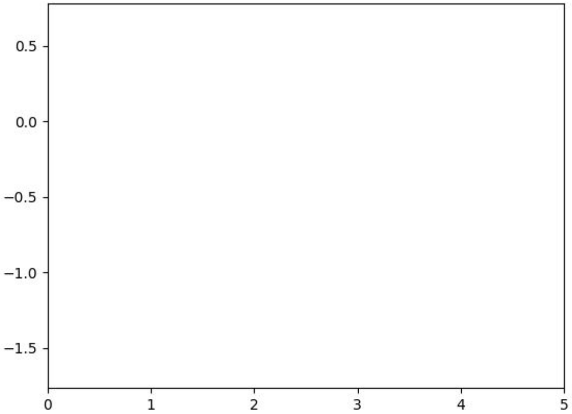
Original Implementation – **Example**

Hypothesis –

left wheel having higher value than right wheel

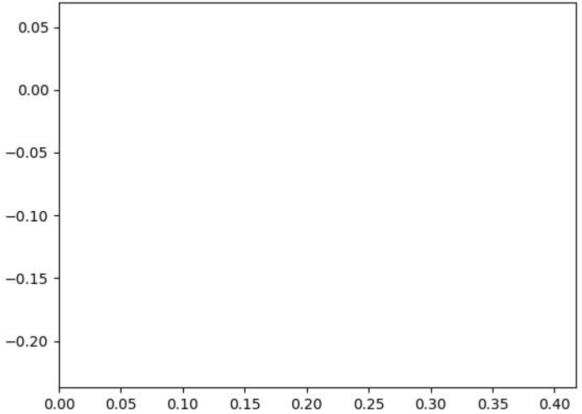


Animations

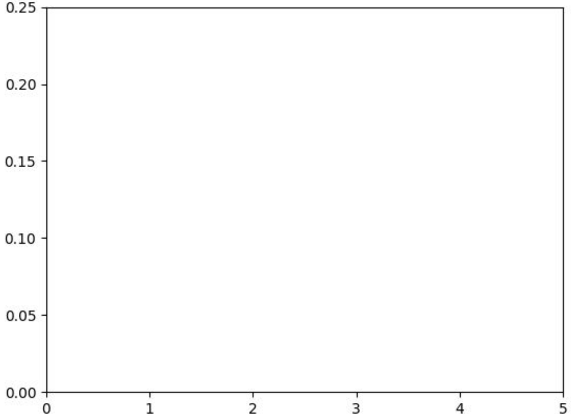


Time List VS. Heading List

X_List VS. Y_List

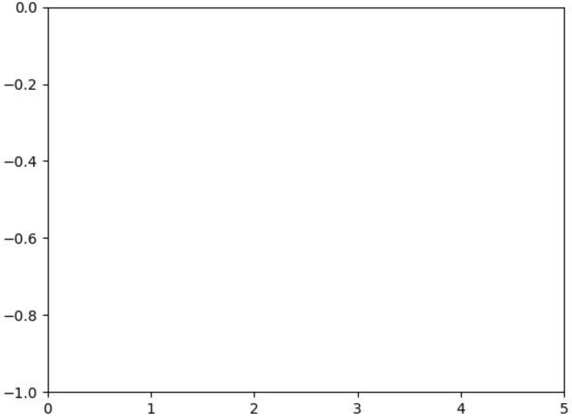


Right Acceleration = 2
Left acceleration = 3



Time List VS. V_List

Time List VS. Ang_Vel List



Implementation of PID and MPC controllers

PID Controllers Implementation

Solving the Matrix:

$$A = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{2b} & \frac{-r}{2b} \end{bmatrix}$$

$$B = \begin{bmatrix} v \\ w \end{bmatrix}$$

$$X = \begin{bmatrix} \theta_R \\ \theta_L \end{bmatrix}$$

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{2b} & \frac{-r}{2b} \end{bmatrix} * \begin{bmatrix} \theta_R \\ \theta_L \end{bmatrix}$$

$$A^{-1}B = X$$

```
#Function purpose: To solve the matrix to solve for what acceleration values would equate to a certain angular and linear velocity

import numpy as np
r = 0.02  #radius of wheel
b = 0.05  #


def solve_matrix(v,w):
    B=np.array([[v],[w]])

    A = np.array([[r/2, r/2],
                  [r/(2*b),-1 * r/(2*b)]])
    A_inv=np.linalg.inv(A)
    X = np.matmul(A_inv, B)
    print(X)
print("Matrix solved where the linear velocity is 1 and the angular velocity is 0, the right and left wheel acceleration is:\n")
solve_matrix(1,0)
print("Matrix solved where the linear velocity is 0 and the angular velocity is 1, the right and left wheel acceleration is:\n")
solve_matrix(0,1)
print("Matrix solved where the linear velocity is 1 and the angular velocity is 1, the right and left wheel acceleration is:\n")
solve_matrix(1,1)
```

```
Matrix solved where the linear velocity is 1 and the angular velocity is 0, the right and left wheel acceleration is:

[[50.]
 [50.]]
Matrix solved where the linear velocity is 0 and the angular velocity is 1, the right and left wheel acceleration is:

[[ 2.5]
 [-2.5]]
Matrix solved where the linear velocity is 1 and the angular velocity is 1, the right and left wheel acceleration is:

[[52.5]
 [47.5]]
```

Logic behind the PIDs

$$V = 1\ ,\ B = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ X = \begin{bmatrix} 50 \\ 50 \end{bmatrix}$$

$$W = 1\ ,\ B = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ X = \begin{bmatrix} 2.5 \\ -2.5 \end{bmatrix}$$

$$W = 1, V = 1\ ,\ B = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ X = \begin{bmatrix} 52.5 \\ 47.5 \end{bmatrix}$$

$$\begin{bmatrix} 50 \\ 50 \end{bmatrix} + \begin{bmatrix} 2.5 \\ -2.5 \end{bmatrix} = \begin{bmatrix} 52.5 \\ 47.5 \end{bmatrix} \text{ therefore, } \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

PID Code (*not including Plotting*):

```
ref_values = []
left_acceleration_values = []
right_acceleration_values = []

t_f = 5 #end time
#setting up the numerical solution parameters
v_0 = 0 #meters/second
dt = 0.1 # delta t
max_acc_l = 50 #max acc used for saturation
max_acc_r = 50


acc_r = 0  #set to 0, since PID will control this
acc_l = 0  #set to 0, since PID will control this

e_prev_r = 0 #for pid controls
e_int_r = 0 #for pid controls

e_prev_l = 0 #for pid controls
e_int_l = 0 #for pid controls

#PID #1 Left wheel
K_p1 = 400  #past
K_d1 = 20   #future
K_i1 = 10 #present
#PID #2 Right wheel
K_p2 = 400  #past
K_d2 = 20   #future
K_i2 = 10  #present

v_d = 1  #desired linear velocity
v_k = v_0
w_k = 0 #set the angular velocity to zero

A = np.array([[r/2, r/2],  #matrix that holds the formula for velocity and angular velocity
            [r/(2*b),-1 * r/(2*b)]])

B = np.array([[acc_l],[acc_r]]) #matrix that holds the left and right acceleration

v_w = np.array([[v_k],[w_k]])  #matrix that holds the values of the velocity and the angular velocity
```

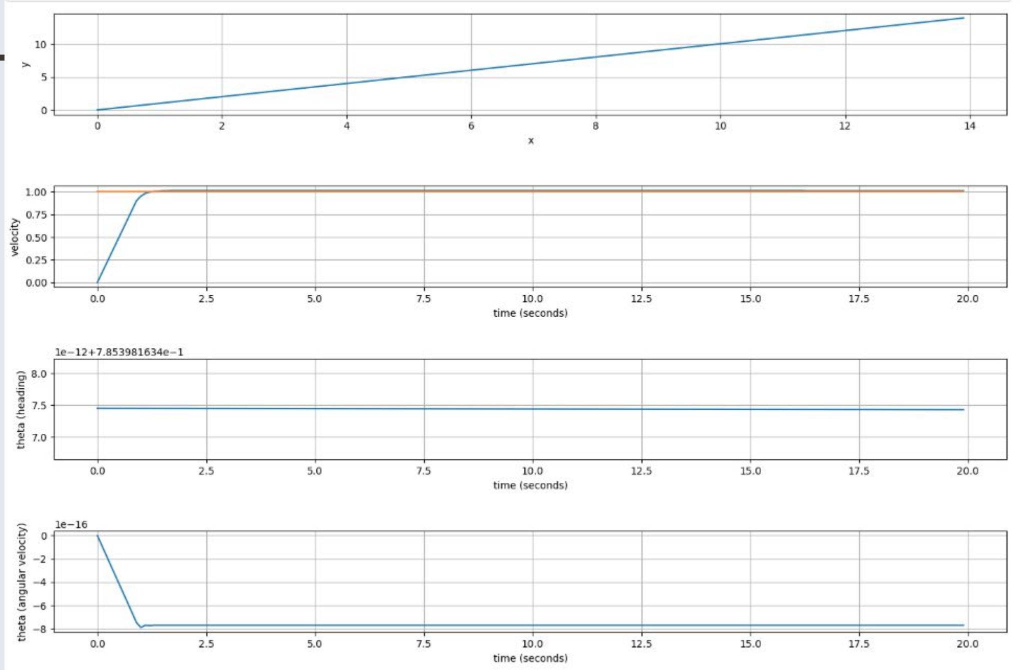
```
while t < t_f:  #the discrete time format in code
    ref_values.append(v_d)

    #PID#1 Left Wheel
    e_k_l = v_d - v_k #calculates error, desired acceleration minus current left_acceleration
    e_dot_l = (e_k_l - e_prev_l) /dt
    e_prev_l = e_k_l
    e_int_l += e_k_l* dt
    acceleration_left = K_p1 * e_k_l + K_i1 * e_int_l + K_d1 * e_dot_l
    acc_l = saturate(acceleration_left, max_acc_l)
    left_acceleration_values.append(acc_l)

    #PID#2 Right Wheel
    e_k_r = v_d - v_k #calculates error, reference minus the calculated velocity
    e_dot_r = (e_k_r - e_prev_r) /dt
    e_prev_r = e_k_r
    e_int_r += e_k_r* dt
    acceleration_right = K_p2 * e_k_r + K_i2 * e_int_r + K_d2 * e_dot_r
    acc_r = saturate(acceleration_right, max_acc_r)
    right_acceleration_values.append(acc_r)

    B[0] = acc_r #updating matrix with updated acceleration
    B[1] = acc_l #updating matrix with updated acceleration
    v_k = float((v_w[0]))  #takes the velocity from the matrix (float because had issue with it appending arrays instead of numeric values)
    w_k = float((v_w[1]))  #takes the angular velocity from the matrix and holds it as the angular velocity
    v_list.append(v_k)
    x_list.append(x_k)
    y_list.append(y_k)
    time_list.append(t)
    heading.append(theta_k)
    ang_vel_list.append(w_k)

    v_w1 = (A.dot(B) * dt) + v_w  #calculates the matrix for the time step
    v_w = v_w1 #sets up the matrix for the next time step
    v_k = float(v_w[0]) #translates the velocity to a numeric value to be used for the x,y,and theta calculations
    w_k = float(v_w[1]) #translates the angular velocity to a numeric value to be used for the x,y,and theta calculations
    theta_k1 = dt*(w_k) + theta_k  #calculates next heading
    theta_k = theta_k1  #sets up heading for the next time step
    x_k1 = dt*(math.cos(theta_k)*v_k)+x_k  #calculates next x-value
    x_k = x_k1  #sets up x for the next time step
    y_k1 = dt*(math.sin(theta_k)*v_k)+y_k  #calculates next y
    y_k = y_k1 #sets up y for the next time step
    t += dt  #increments time
```

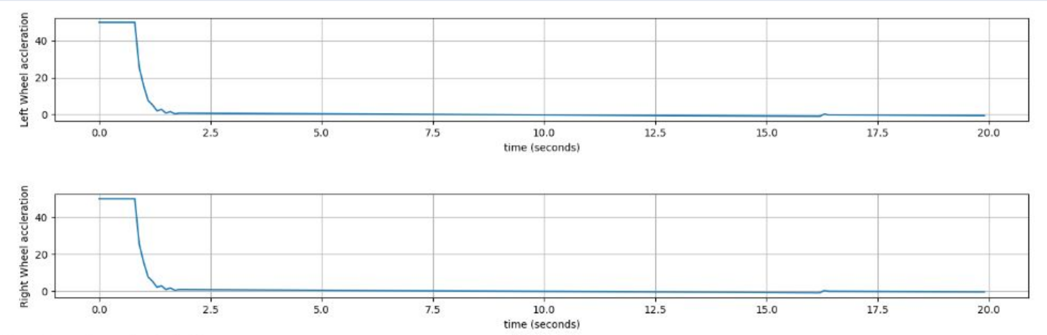
PID Examples:

5 Examples:
*(where V is the **desired velocity** and W is the **desired angular velocity**)*

1. V = 1
2. W = 1 (Using predicted values from solving the matrix)
3. W = 1 (Left Acceleration is 0 but right acceleration is set)
4. W = 1, V =1 (Using values from #2)
5. W = 1, V = 1 (Using values from #3)

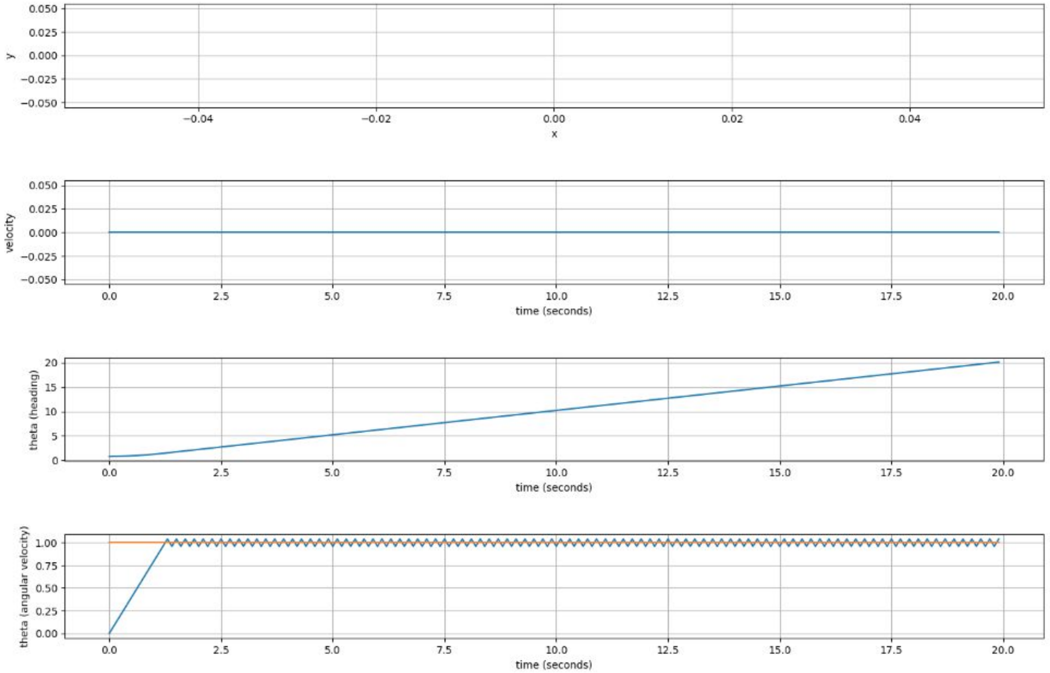
EX #1 :
PID for a linear velocity of 1:

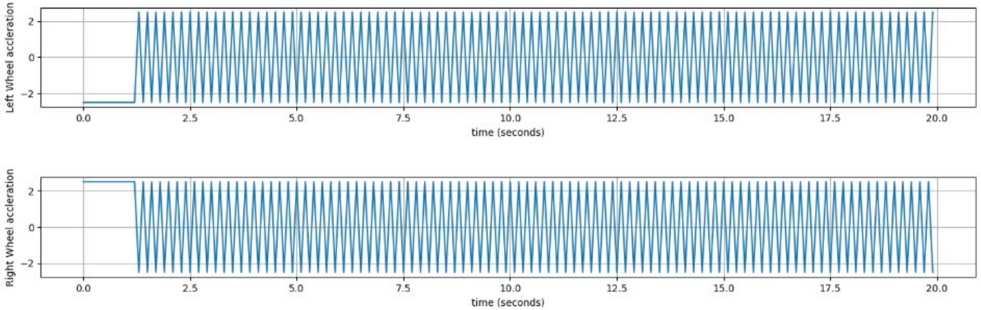





```
Percentage of overshoot: 1.20%
Ess (m): -0.010000000000000009
Peak time: 1.7000000000000002 seconds
```

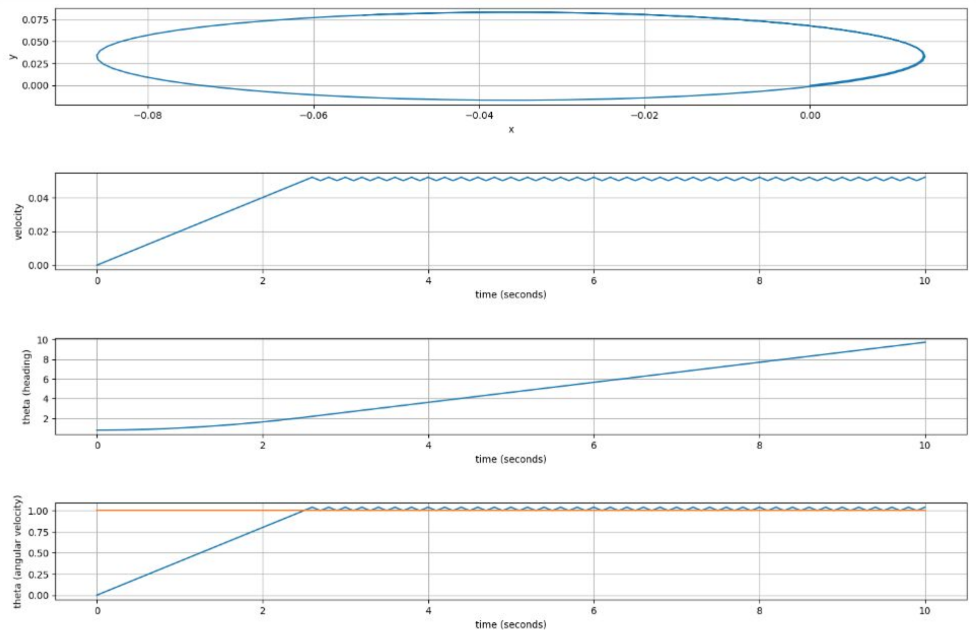
EX #2
PID for W = 1 (Using values from solving the matrix)

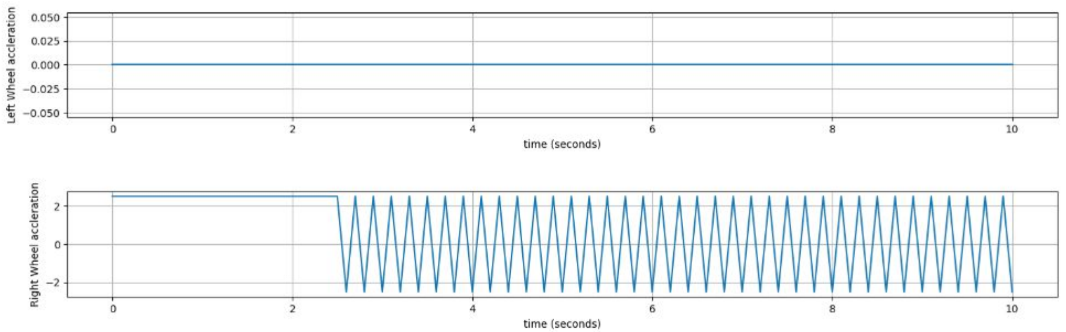





```
Percentage of overshoot: 4.00%
Ess (m): 0.04000000000000015
Peak time: 1.3 seconds
```

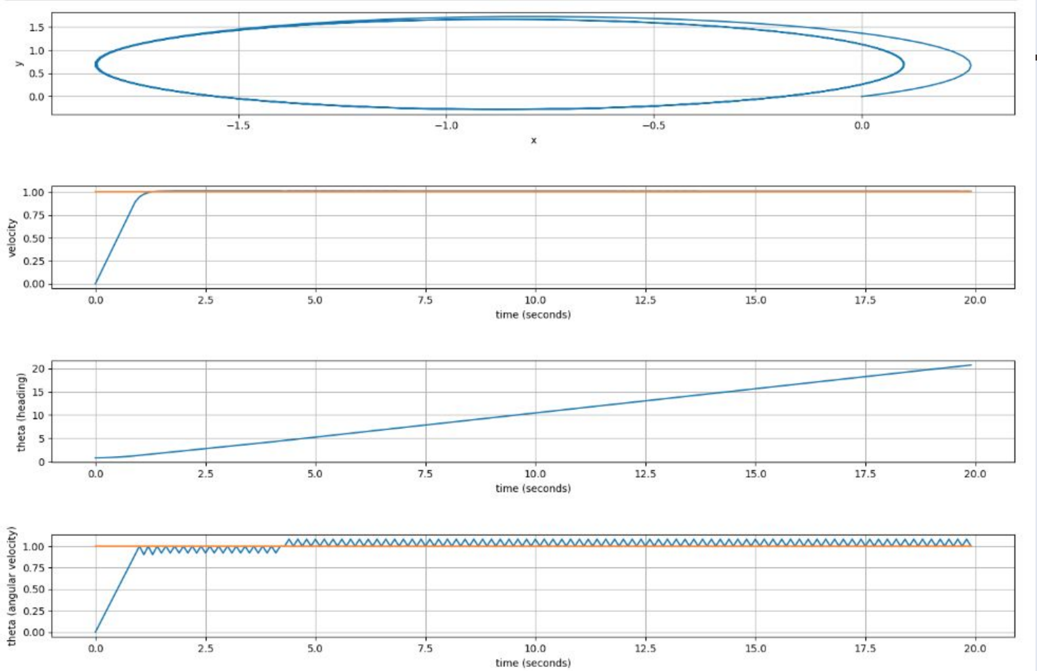
EX #3
PID for W = 1 (Using 0 as the max for left acceleration and 2.5 as the max of the right acceleration)

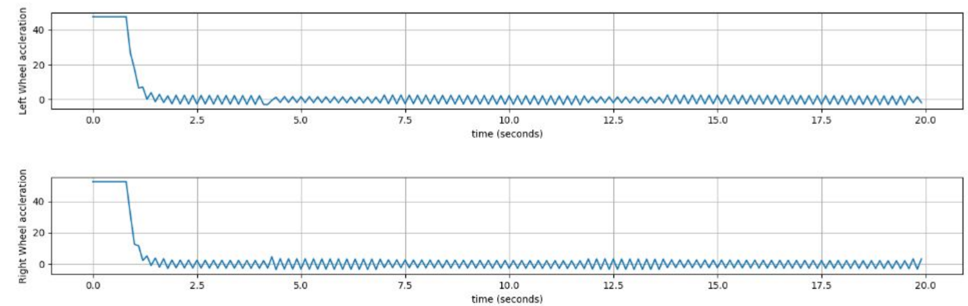





```
Percentage of overshoot: 4.00%
Ess (m): 0.95
Peak time: 2.6 seconds
```

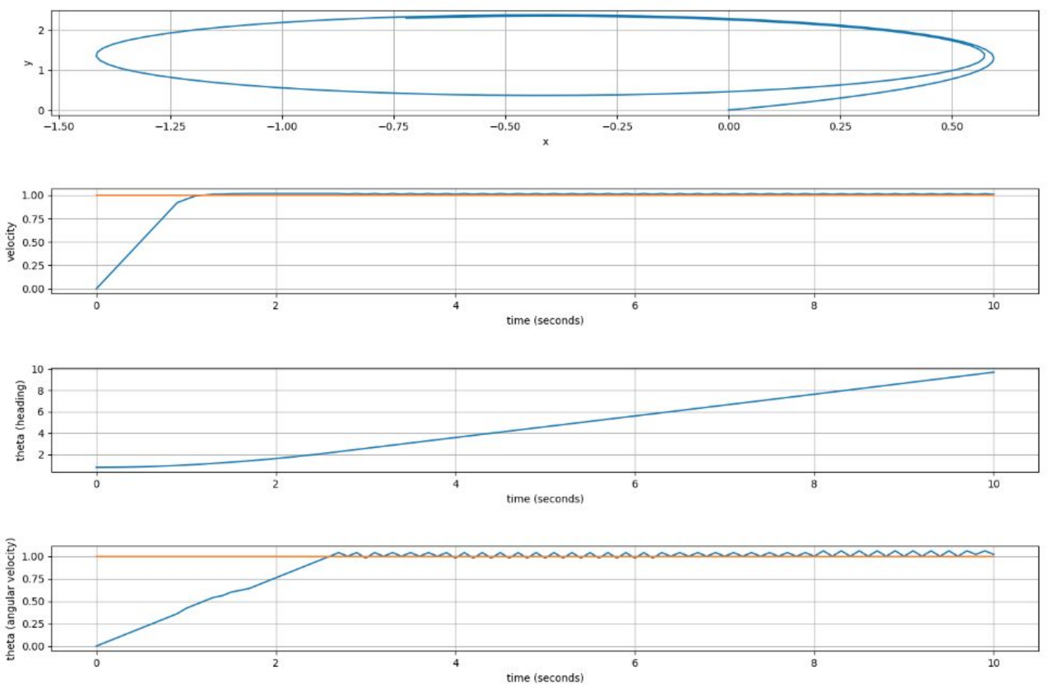
EX#4
PID for V=1 and W = 1 (Using values from solving the matrix for max acceleration values)

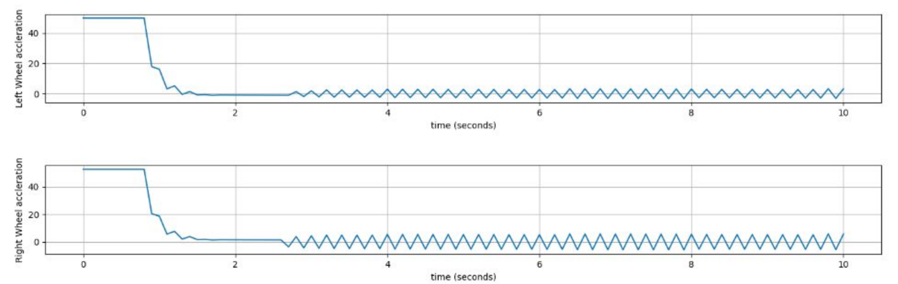





```
Percentage of overshoot (linear velocity): 1.40%
Ess (m) (linear velocity): -0.009999999999999787
Peak time (linear velocity): 1.8 seconds
Percentage of overshoot (angular linear velocity): 8.00%
Ess (m (Angular linear velocity)): -0.009999999999999787
Peak time (Angular linear velocity): 4.4 seconds
```

EX #5
PID for V=1 and W = 1 (Using 2.5 for max right acceleration and 0 for max left acceleration)






```
Percentage of overshoot (linear velocity): 1.60%
Ess (m) (linear velocity): -0.014999999999999458
Peak time (linear velocity): 1.7000000000000002 seconds
Percentage of overshoot (angular linear velocity): 6.00%
Ess (m (Angular linear velocity)): -0.014999999999999458
Peak time (Angular linear velocity): 8.1 seconds
```

MPC Controllers Implementation

MPC Code (*not including Plotting*):

```
#Description: This MPC aims to model the DDR to reach a linear velocity of 1 through finding the correct left and right wheel accelerations
#Source: This code takes code from the csc-340-MPC.ipynb, but is adapted to our model
#Source: The code for the overshoot, steady state error, and peak time come from HW5
r,b,t_0, x_k, y_k,w_k, v_k,y_list, x_list,time_list,v_list, heading, ang_vel_list ,t, t_f, dt, theta_k = set_variables_back()

left_acceleration_values = []
right_acceleration_values = []
ref_values = []

v_d = 1
accel_r_max = 50
accel_l_max = 50

dt = 0.1 # delta t = 0.1 second

x_0 =  0
v_0 =  0


r = 0.02  #radius of the wheel in meters
b = 0.05  #length of half of axle, in meters
t_0 = 0  #starting time
t_f = 20  #ending time
dt = 0.1 #time increment
x_k =0   #original x value, 0
y_k =0  #original y value, 0

w_k = 0  #angular velocity
v_k = 0  #velocity

acc_l = 0  #acceleration of the left wheel
acc_r = 0  #acceleration of the right wheel
theta_k = math.pi/4  #heading of the robot, set to pi/4 becuase of diagram wanted it to be easy to visualize
y_list = []  #initializing y list
x_list = []  #initializing x list
time_list = []  #initializing time list
v_list=[]  #intializing velocity list
heading = []  #intialiing heading
ang_vel_list = []  #intializing angular velocity list
t = t_0 #setting t to the start poistion

A = np.array([[r/2, r/2],
            [r/(2*b),-1 * r/(2*b)]])

B = np.array([[acc_l],[acc_r]])

v_w = np.array([[v_k],[w_k]])
```

```
############ MPC setup
# setting up the MPC solver
model_type = 'discrete' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
# define the states
v_lin = model.set_variable(var_type='_x', var_name='v_lin', shape=(1,1))
v_ang = model.set_variable(var_type='_x', var_name='v_ang', shape=(1,1))

# define the inputs
accel_l = model.set_variable(var_type='_u', var_name='accel_l')
accel_r = model.set_variable(var_type='_u', var_name='accel_r')
# setup the control goals
# avoid large forces
model.set_expression( expr_name="lagrange_term", expr= 0.00001* accel_l **2+ 0.00001* accel_r **2 )

# reach the final destinatioin (x_d) and stop
model.set_expression(
    expr_name="meyer_term", expr= 1000*(v_d - v_lin) ** 2+ 1* v_lin**2
)
# include the dynamics in the model
model.set_rhs("v_lin", ((((accel_l * r)/2) + ((accel_r * r)/2))*dt + v_lin))
model.set_rhs("v_ang", ((((accel_l * r)/2) + ((accel_r * r)/2))*dt + v_ang))

# finish the setup
model.setup()
print('Model defined!')

# set up the MPC solver
mpc = do_mpc.controller.MPC(model)
mpc.settings.n_horizon = 10 # predict the next N steps
mpc.settings.t_step = dt

lterm = model.aux["lagrange_term"]
mterm = model.aux["meyer_term"]

mpc.set_objective(lterm=lterm, mterm=mterm)
mpc.set_rterm(accel_l=0)
mpc.set_rterm(accel_r=0)

mpc.scaling['_x', 'v_lin'] = 1
mpc.scaling['_x', 'v_ang'] = 1

mpc.bounds["lower", "_u", "accel_l"] = -accel_l_max
mpc.bounds["upper", "_u", "accel_l"] = accel_l_max

mpc.bounds["lower", "_u", "accel_r"] = -accel_r_max
mpc.bounds["upper", "_u", "accel_r"] = accel_r_max

surpress_ipopt = {'ipopt.print_level':0, 'ipopt.sb': 'yes', 'print_time':0}
mpc.set_param(nlpsol_opts = surpress_ipopt)
mpc.setup()
print('MPC solver defined!')
```

Code Continued…

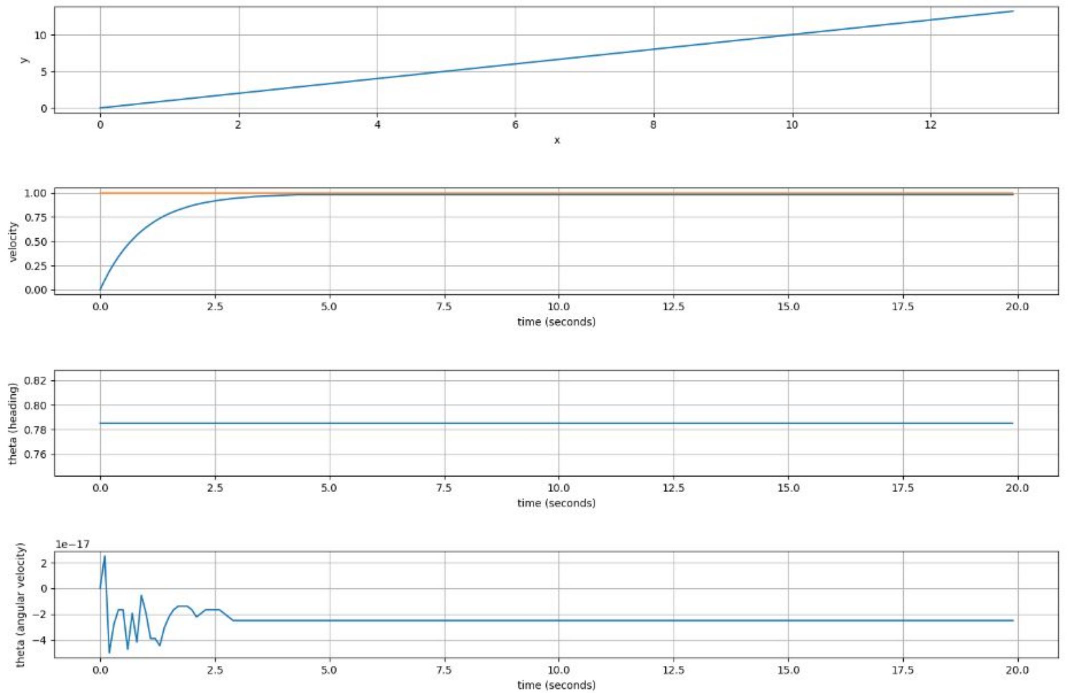
```
# simulate the system
while t < t_f:
    ref_values.append(v_d)
    # set the solver for the current iteration
    mpc.x0 = v_w
    mpc.set_initial_guess()
    u_opt = mpc.make_step(v_w) # find the optimal input

    B[0] = u_opt[0]
    B[1] = u_opt[1]

    right_acceleration_values.append(float(B[0]))
    left_acceleration_values.append(float(B[1]))

    v_list.append(v_k)
    x_list.append(x_k)
    y_list.append(y_k)
    time_list.append(t)
    heading.append(theta_k)
    ang_vel_list.append(w_k)

    v_w1 = (A.dot(B) * dt) + v_w
    v_w = v_w1
    v_k = float(v_w[0])
    w_k = float(v_w[1])

    theta_k1 = dt*(w_k) + theta_k
    theta_k = theta_k1
    x_k1 = dt*(math.cos(theta_k)*v_k)+x_k
    x_k = x_k1
    y_k1 = dt*(math.sin(theta_k)*v_k)+y_k
    y_k = y_k1
    t += dt
```

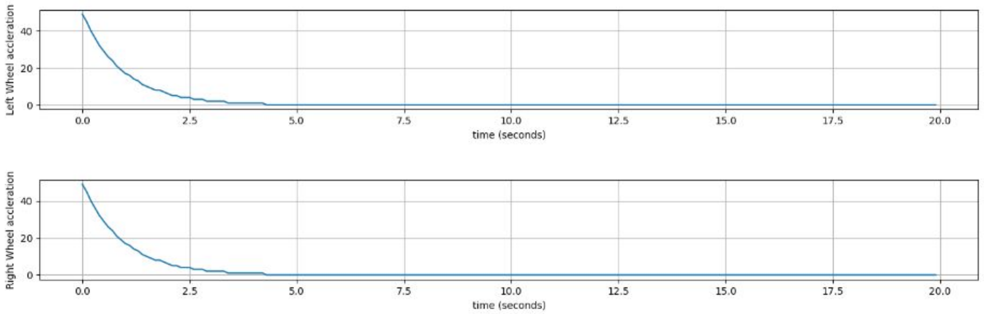
MPC examples:

4 Examples:
*(where V is the **desired velocity** and W is the **desired angular velocity**)*

1. V = 1
2. W = 1 (Using 0 for right acceleration and a value for left acceleration)
3. V = -1
4. W = -1 (Using a value for left acceleration and 0 for right acceleration)

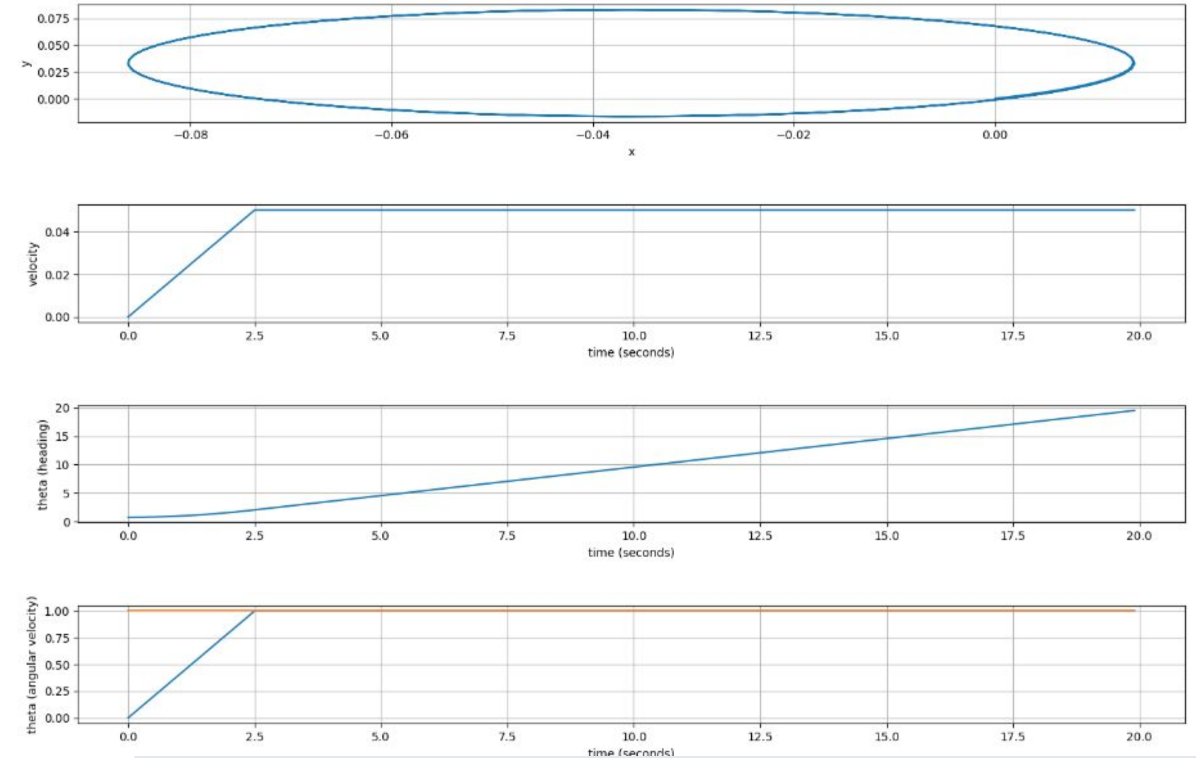
EX #1
MPC for V=1

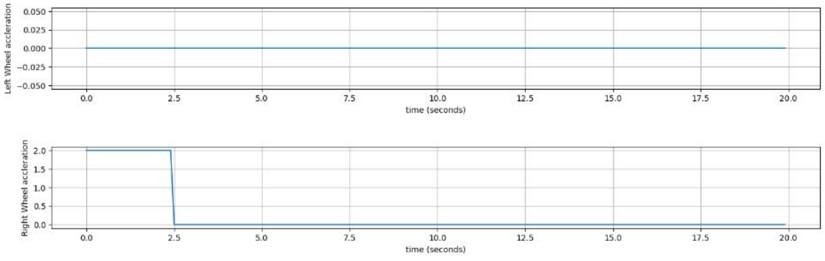





```
Percentage of overshoot (linear velocity): -2.00%
Ess (m) (linear velocity): 0.019999999999999574
Peak time (linear velocity): 4.3 seconds
```

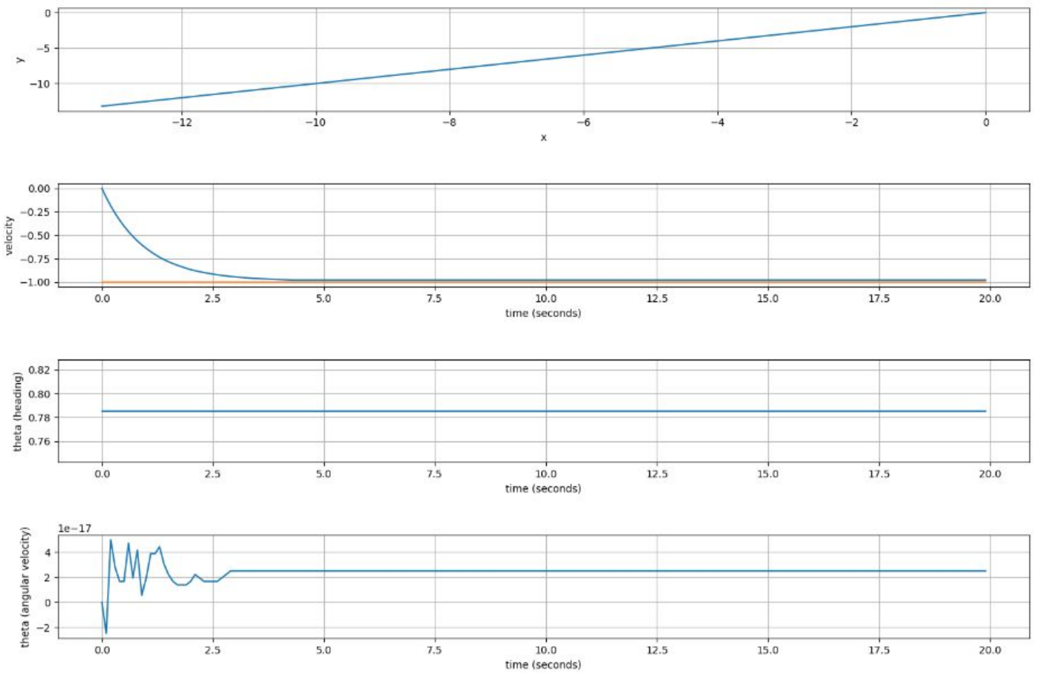
EX #2
MPC for W=1

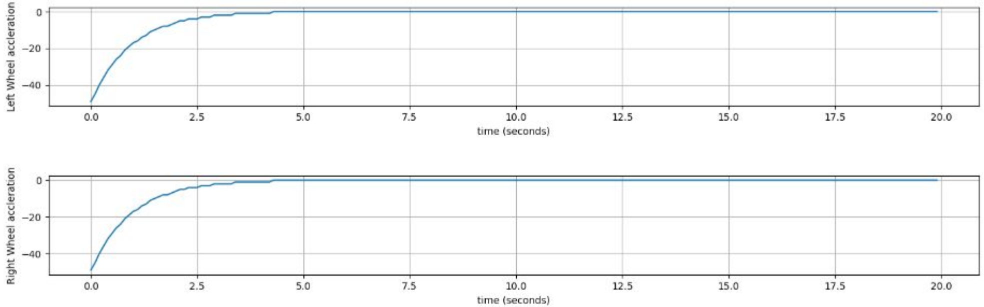





```
Percentage of overshoot (angular linear velocity): 0.00%
Ess (m (Angular linear velocity)): 0.95
Peak time (Angular linear velocity): 2.5 seconds
```

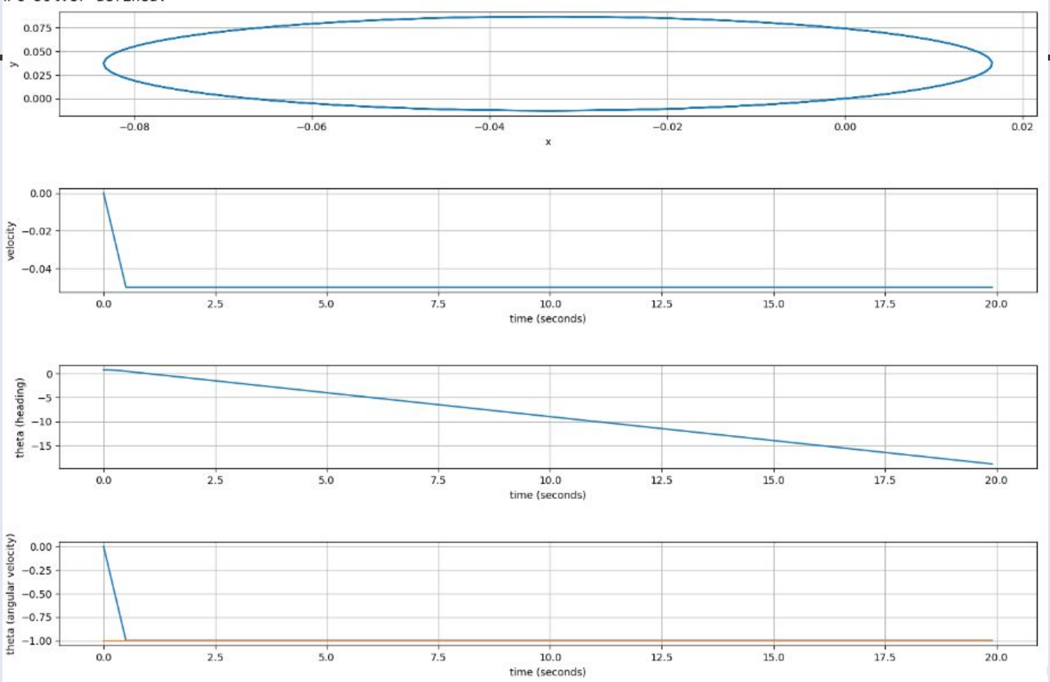
EX #3
MPC for V=-1

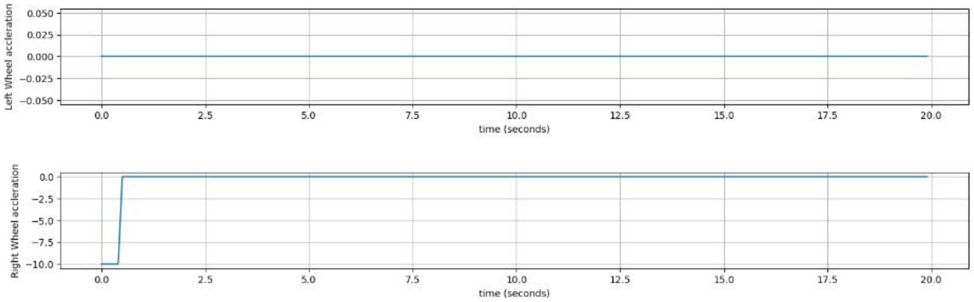





```
Percentage of overshoot for linear velocity: -2.00%
Ess (m) (linear velocity): -0.019999999999999574
Peak time (linear velocity): 4.3 seconds
```

EX #4
MPC for W=-1






```
Percentage of overshoot (angular linear velocity): -0.00%
Ess (m) (Angular linear velocity)): -1.1102230246251565e-16
Peak time (Angular linear velocity): 0.5 seconds
```

Inputs and Outputs

Actuation Devices:

- **Drive Motors**
- **Motor Controllers**
- **PID Controller**
- **MPC Controller** – *future behavior prediction*

Inputs:

- **Speed Values** – (Typically in *rpm* or *m/s*)
- **Velocity Values**

Sensors *(In coordination with our real-life examples)*

- Gyroscope/Accelerometer *(Part of IMU)*
- Proximity Sensors
- Inertial Measurement Units *(IMUs)*
- Temperature Sensors *(to ensure no overheating)*
- GPS Modules *(Especially DDRs operating outside)*
- *(More possible depending on devices needs)*

CONTROLLED Outputs:

- Velocity
- Position
- Heading
- Angular Velocity

[12] https://www.wevolver.com/article/sensors-in-robotics-the-common-types

Conclusion for Discrete Time Equation modeling – *project 1*

Limitations:

- Mass is not taken into account
- Bumps in terrain are not accounted for
- Differences in terrain (rocky, incline/decline, etc.)
- Friction of the wheels on the surface slowing the DDR down

Stable Behavior:

- Acceleration (due to it remaining the same throughout the experiment)

Unstable Behavior:

- Dampening on velocity is not included in our system, therefore; the velocity increases with no limit.

Conclusion for our PID and MPC controllers

- MPC appears to be a lot smoother and more accurate.
- An MPC is a lot easier to implement since there is no need to adjust the PID values *(Kp, Ki, Kd),*
- The model is extremely limited, considering once the model reaches the target linear or angular velocity it stays at that value.
- This makes it hard to model for a positional goal, since the DDR won't really slow down.
- Hard to make a model to reach a desired linear and angular velocity; PIDs might work against each other.
- Some improvements that could be made on this model is to have a dampening system on the linear and angular velocity so that the plots appear more realistic . This would allow the wheel accelerations to be constantly changed to a value in order to keep at the target.

Citations

1. *AquaNaut 200 - 2 - Wheel Drive, Pools up to 16’ x 32’*. (n.d.).https://hayward.com/aquanaut-200-2-wheel-drive-pools-up-to-16-x-32.html
2. Lydsto. (n.d.). *Lydsto R5 3000PA Self-Cleaning Robot Vacuum & MoP*. https://lydsto.com/products/lydsto-r5
3. Wikipedia contributors. (2023). Arduino. *Wikipedia*. https://en.wikipedia.org/wiki/Arduino
4. https://www.amazon.com/perseids-Chassis-Encoder-Wheels-Battery/dp/B07DNXBFQN?th=1
5. https://microdcmotors.com/product/36mm-12v-24v-dc-rs550-555-motor-high-speed-large-torque-used-for-power-tools
6. https://42bots.com/tutorials/differential-steering-with-continuous-rotation-servos-and-arduino/
7. https://www.electricwheelchairsusa.com/products/pride-jazzy-elite-14-front-wheel-drive-power-chair-elite14
8. https://www.teachengineering.org/lessons/view/cub_mars_lesson02
9. https://www.youtube.com/watch?v=XG4cODYVbJk
10. https://www.roboticsbook.org/S52_diffdrive_actions.html
11. https://blog.pal-robotics.com/omnidirectional-vs-differential-drive-robots/
12. https://www.wevolver.com/article/sensors-in-robotics-the-common-types
13. https://en.wikipedia.org/wiki/Differential_wheeled_robot
14. https://www.techtarget.com/iotagenda/definition/mobile-robot-mobile-robotics#:~:text=While%20the%20industrial%20use%20of,common%20uses%20of%20mobile%20robots.