COMP0128 - Group Coursework 1

Chiang Hou Nam 15055142
Tim Andersson 20167996
Habib Sarosh 20096908

Question 1

The left and right wheels having different speeds can be attributed to a number of parameters in the LTI system:

- The resistance in the circuit of each wheel may differ, which will affect the angular velocity for said wheel.
- In reality, the relative smoothness of each wheel may differ, which affects the friction constant b, which in turn affects the angular velocity of the wheel.
- The induction L, in Henries, may differ between the two LTI systems, which in turn would affect the angular velocities of the wheels.
- The rotational inertia for each wheel, J, would also be a parameter that would affect each wheel. In a ‘real life’ case, this would be a constant found out through investigation of the robotic system in question. However, in the case of a simulation, a reasonable value is assigned for each wheel. If these were to differ, they would affect the speed of the individual wheels differently.
- K_e, the emf constant, is another physical property defined by a real LTI dc motor system. This, in a real application, would likely be provided by the robot manufacturer or would be something that would have to be derived. This also would affect the performance of each respective wheel should it differ between the two systems.

Question 2

Please see below code for ***init_question_2.m***:

```
Kp = 5;
```

Please see below code for ***controller_question_2.m***:

```
function u = controller_question_2(y) % returns input (u) from odometry (y)
       init_question_2;
       error_left = y(2) - y(1);
       error_right = y(1) - y(2);
       u = [(4 + error_left*Kp); (4 + error_right*Kp)];
end
```

As seen above, the only parameter initialised in order to complete this task was K_P. K_P is a proportional gain, used to adjust the input voltage at each sample step according to the difference between the wheel odometry measurements (y), in order to keep the robot moving in a straight line. You can see this in the declaration of u, where the input voltages are four volts plus the error times K_P with respect to the wheel in question.

Question 3

Please see below code for ***init_question_3.m***:

```
Kp = 5;
lw = 0.238;
epsilon = 0.05;
```

In this instance, K_P is initialised to 3. l_w is defined as the width of the robot between the wheels. This was derived through the following experimentation:

- Set wheel input voltage to be 0 for one wheel, and any arbitrary positive number for the other
- Using the plot from the provided simulation, analyse the odometry measurements when the robot has completed a full circle.
- Using the odometry measurements, derive the amount of ticks of the wheel with the input voltage not set to 0 for the full circle.
- Using the implementation for question 2, derive the distance per tick based on the odometry measurements and the plot from the provided simulation.
- Use the above to calculate the circumference of the circle the robot just made, and then from the circumference, derive the radius of the circle, which is equal to l_w.

The parameter ϵ is a margin of error in metres to tell the robot whether it has reached its end point to a certain level of accuracy based on hypotenuse distance.

Please see below code for ***controller_question_3.m*** complete with annotations describing the process:

```
function [u, d, co] = controller_question_3(y, co_ref, co, i_th, d)
      init_question_3;
       % This function returns input voltage, distance travelled at this sample and
the
       % current coordinate for the robot position.
      % Here the distance travelled by each wheel is derived based on odometry
       % measurements and the wheel circumference (0.4204)
      d_l = y(1) * (0.4204/64);
      d_r = y(2) * (0.4204/64);

      % alpha is change of angle from origin position.
      alpha = (1 / lw) * (d_r - d_l);

      % Here the speed of each wheel is derived, based on the difference between the
       % distance of each wheel travelled at the current sample and the distance
       % of each wheel travelled at the last sample.
       % s_m is the speed of the robot parallel to its orientation.
```

```
        s_l = (d_l - d(1));
        s_r = (d_r - d(2));
        s_m = (1/2) * (s_l + s_r);

        % x_dot and y_dot respectively are the change of x and y position of the robot
        % after the current sample.
        x_dot = s_m * cos(alpha);
        y_dot = s_m * sin(alpha);

        % here the estimated coordinates are updated using the forward kinematics.
        co(1) = co(1) + x_dot;
        co(2) = co(2) + y_dot;

        % The angle between the robot and the desired reference point is derived,
         % based on the arctangent with regard to the current robot position, minus the
         % robot orientation, alpha.
        phi = atan2(co_ref(2) - co(2), co_ref(1) - co(1)) - alpha;

        % Here the current distance is stored in order to use at the next sample step.
        d = [d_l; d_r];

        % Stopping condition created using hypotenuse distance to the reference point.
        x_diff = ((co(1) - co_ref(1))^2);
        y_diff = ((co(2) - co_ref(2))^2);
        hyp_diff = sqrt(x_diff + y_diff);
         correction = phi*Kp;
        if (hyp_diff < epsilon)
             u = [0; 0];
        else
            u_min = -6; % set variables to ensure the input voltage never goes above
            u_max = 6; % or below the min and max of -6v and 6v respectively.

            u = [4-correction; 4+correction];

            % The code here simply regulates the input voltages.
            if (4-correction > u_max)
                u(1) = u_max;
            elseif (4-correction < u_min)
                u(2) = u_min;
            end
            if (4+correction > u_max)
                u(2) = u_max;
            elseif (4+correction < u_min)
                u(2) = u_min;
            end
        end
        % simple function to plot a cross for the robot position every 20 samples.
        if (mod(i_th, 20) == 0)
               plot(co(1), co(2), 'g*', 'markersize', 1);
        end
end
```

The system uses the odometry measurements at each sample to derive forward kinematics and estimate the robot position at the next sample. It does this using the circumference of the wheels, the robot width l_w, and the cumulative amount of ticks over samples from the geometry measurements to derive the distance travelled by each wheel. The difference in ticks between subsequent samples is used to derive the speed for each wheel of the robot respectively, by comparing the distance travelled at the current sample with the distance travelled at the last sample.

Using the change of robot orientation (α) from the origin and the speed of the robot parallel to its orientation (S_M), the estimated change of x and y position of the robot from the current sample to the next are estimated, completing the forward kinematics. The current coordinates are then updated.

$$S_M = \frac{1}{2}(S_L + S_R)$$

$$\alpha = \frac{1}{l_w}(d_R - d_L)$$

$$\dot{p_x} = S_M cos\alpha, \dot{p_y} = S_M sin\alpha$$

φ, the angle between the robot at the estimated next position and the reference point is then calculated using the ***atan2*** built in MATLAB function. The robot orientation, α, is then subtracted from this value to give the angle between the robot, given position and orientation, and the reference point ***co_ref***.

$$\varphi = tan^{-1}\left(\frac{r_y - p_y}{r_x - p_x}\right) - \alpha$$

After this, a simple if statement is used to define a stopping condition based on the hypotenuse distance between the robot and the reference point.

Subsequently, numerous if statements are used to regulate the input voltage to the robot so that it does not exceed 6V or -6V respectively, which is defined as the minimum and maximum input voltage for the system. The last few lines of code simply plot the robot position based on the forward kinematics every 20 samples.

Question 4

Similar to Question 3, moving the robot to the desired position is achieved by odometry conversion, forward kinematics, calculating PID correction from angle difference to the desired location and then moving the DC motors with the corresponding corrections. To detect if the robot has arrived at the desired position, after forward kinematics at every time step, the distance differences of the current robot x- and y-coordinates (***p_x*** and ***p_y*** respectively) to the x- and y- coordinates of the desired position are calculated and checked if they are both smaller than a certain error *epsilon*, for example 0.01 (as shown in the code block below, the other parts of the code are more or less the same as Question 3, for more details, please check the corresponding .m files).

```
% Acceptable error from the destination
epsilon = 0.01
% Check whether the robot's current position is at the desired location with a
slight error epsilon
if abs(p_x - co_ref(1)) < epsilon && abs(p_y - co_ref(2)) < epsilon
        % Tell the robot to return if it has arrived at the first destination
        is_returning = true;
        % Tell the robot to head back to the origin
        co_ref = [0; 0];
end
```

If true, the robot's new desired position to move to is set to the origin (0,0) and then it is moved accordingly using the same exact control logic as before but with a different desired position. This checking step is only done when the robot has yet to reach the first desired destination by wrapping the above checking procedure into an outer if-statement and checking with the boolean ***is_returning*** as shown below. The ***is_returning*** is set to 'false' in the beginning of the simulation so that this checking step is run in every time step to detect whether the robot has reached the first destination. After it has reached it, this boolean is set to 'true' to avoid going through this checking step again and again.

```
% Acceptable error from the destination
epsilon = 0.01
% Check if the robot has to return to the origin yet
if is_returning == false
    % Check whether the robot's current position is at the desired location with a
    slight error epsilon
    if abs(p_x - co_ref(1)) < epsilon && abs(p_y - co_ref(2)) < epsilon
        % Tell the robot to return if it has arrived at the first destination
        is_returning = true;
        % Tell the robot to head back to the origin
        co_ref = [0; 0];
    end
end
```

Question 5

To move the robot along a desired line, we first have the user to set the values of the slope (m) and y-intercept (b) of the line equation $y = mx + b$. After that, we employed a similar logic to moving to a desired location as in Question 3 but with some minor changes. The odometry conversion and forward kinematics are first calculated in the same way as before to get the newest position and orientation of the robot. For each time step, a ***p_x_target*** is calculated by adding to the current x-coordinate obtained from the forward kinematics a ***x_step_interval*** (e.g. 0.1) to extrapolate the next x-coordinate position to get to. The desired y-coordinate ***p_y_target*** to move to is calculated by inputting the ***p_x_target*** into the line equation:

$$p_{x_{target}} = p_{x_{next}} + x_{step\ interval}$$
$$p_{y_{target}} = m(p_{x_{next}} + x_{step\ interval}) + b$$

This allows the next desired position along the line to move to be extrapolated in order to control the robot to follow this line equation.

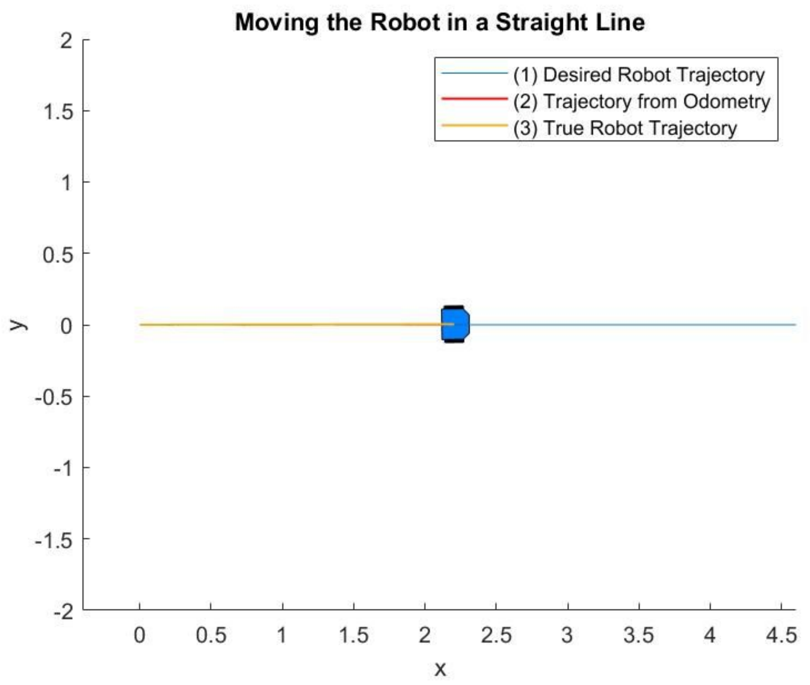
These extrapolated target points ***p_x_target*** and ***p_y_target*** are then used to calculate the angular error which is then fed into the PID controller. The angular correction is then used in order to move the robot to this new desired position at each time step similar to the procedures used before.

```
% Calculate angle difference
r_phi = atan2((p_y_target - p_y) , (p_x_target - p_x));
% r_phi refers to the angular difference between the original robot position and
the destination
error = r_phi - phi;

% Pass into PID Controller
correction = Kp * error;
```

At first, as the robot moves with increasing **p_x**, it tries to locate the y position of the line to follow to. After a certain while, it can perfectly fit along the line as the robot gets closer and closer to the y-position of the line equation.

Question 6

Plot for Question 2:



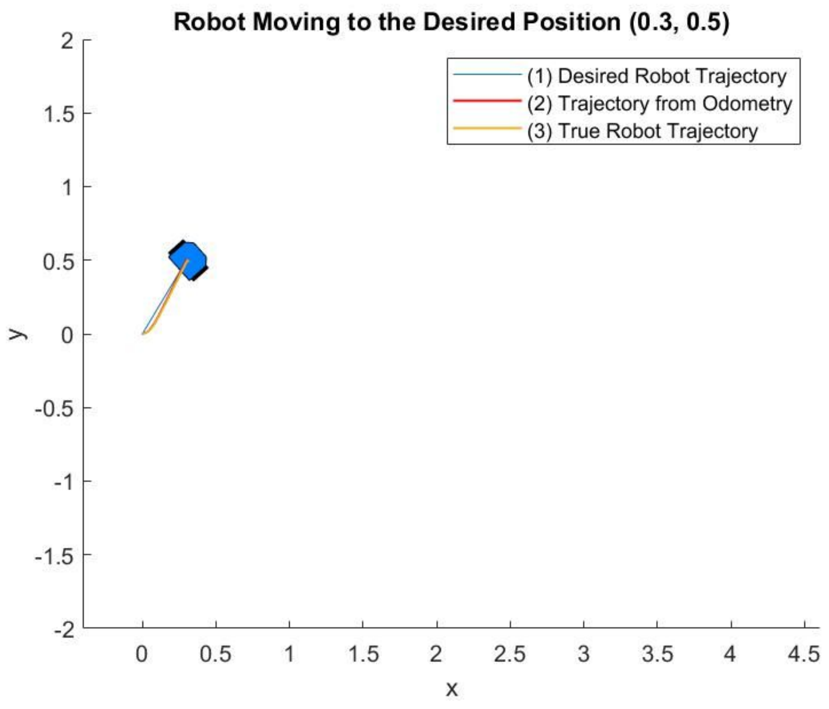
This code block from ***DifferentialDriveDCMotorRobot_2.m*** is run at the beginning of the simulation with ***p_x_all*** and ***p_y_all*** storing all the previous positions of the robot:

```
title('Moving the Robot in a Straight Line')
legend('(2) Trajectory from Odometry', '(1) Desired Robot Trajectory','(3) True
Robot Trajectory')
init_question_2;
% (1) Plot the desired robot trajectory
desiredTrajplot = plot([0 5], [0 0], 'DisplayName', '(1) Desired Robot
Trajectory');
% (2) Plot the robot trajectory estimated from the odometry measurements
p_x_all = [p_x];
p_y_all = [p_y];
odometryTrajplot = plot(p_x_all, p_y_all,'red', 'linewidth', 1, 'DisplayName', '(2)
Trajectory from Odometry');
% (3) Plot the true robot trajectory
trajplot = plot(csim.Log.Output(2,:),csim.Log.Output(3,:),'linewidth',1 ,
'DisplayName', '(3) True Robot Trajectory');
```

At every time step, the following code is run to plot the curves:

```
     controller_question_2;
    % (2) Plot the robot trajectory estimated from the odometry
    % measurements at every time step
    p_x_all(end + 1) = p_x;
    p_y_all(end + 1) = p_y;
    set(odometryTrajplot, 'XData', p_x_all, 'DisplayName', '(2) Trajectory from
Odometry');
    set(odometryTrajplot, 'YData', p_y_all, 'DisplayName', '(2) Trajectory from
Odometry');
```

Plot for Question 3:



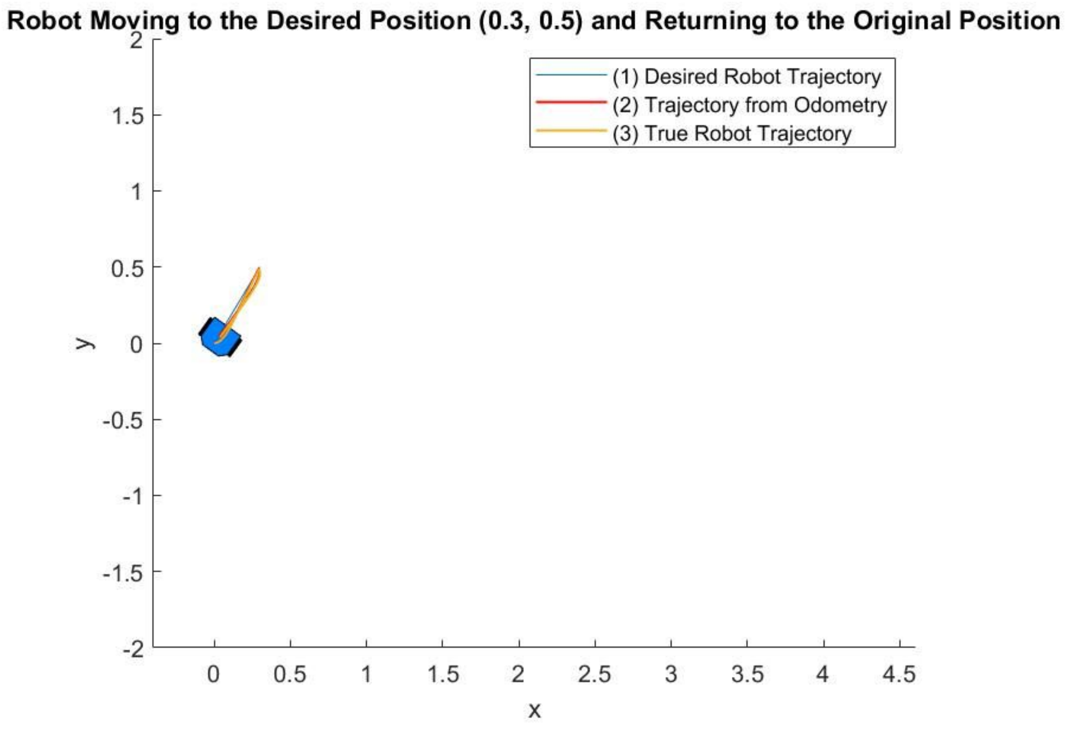
This code block from ***DifferentialDriveDCMotorRobot_3.m*** is run at the beginning of the simulation :

```
title('Robot Moving to the Desired Position (0.3, 0.5)')
legend('(2) Trajectory from Odometry', '(1) Desired Robot Trajectory','(3) True
Robot Trajectory')
init_question_3;
% (1) Plot the desired robot trajectory
desiredTrajplot = plot([0 ref_location(1)], [0 ref_location(2)], 'DisplayName',
'(1) Desired Robot Trajectory');
% (2) Plot the robot trajectory estimated from the odometry measurements
p_x_all = [p_x];
p_y_all = [p_y];
odometryTrajplot = plot(p_x_all, p_y_all,'red', 'linewidth', 1, 'DisplayName', '(2)
Trajectory from Odometry');
% (3) Plot the true robot trajectory
trajplot = plot(csim.Log.Output(2,:),csim.Log.Output(3,:),'linewidth',1 ,
'DisplayName', '(3) True Robot Trajectory');
```

At every time step, the following code is run to plot the curves:

```
controller_question_3;
% (2) Plot the robot trajectory estimated from the odometry
% measurements at every time step
p_x_all(end + 1) = p_x;
p_y_all(end + 1) = p_y;
set(odometryTrajplot, 'XData', p_x_all, 'DisplayName', '(2) Trajectory from
Odometry');
set(odometryTrajplot, 'YData', p_y_all, 'DisplayName', '(2) Trajectory from
Odometry');
```

Plot for Question 4:



This code block from ***DifferentialDriveDCMotorRobot_4.m*** is run at the beginning of the simulation:

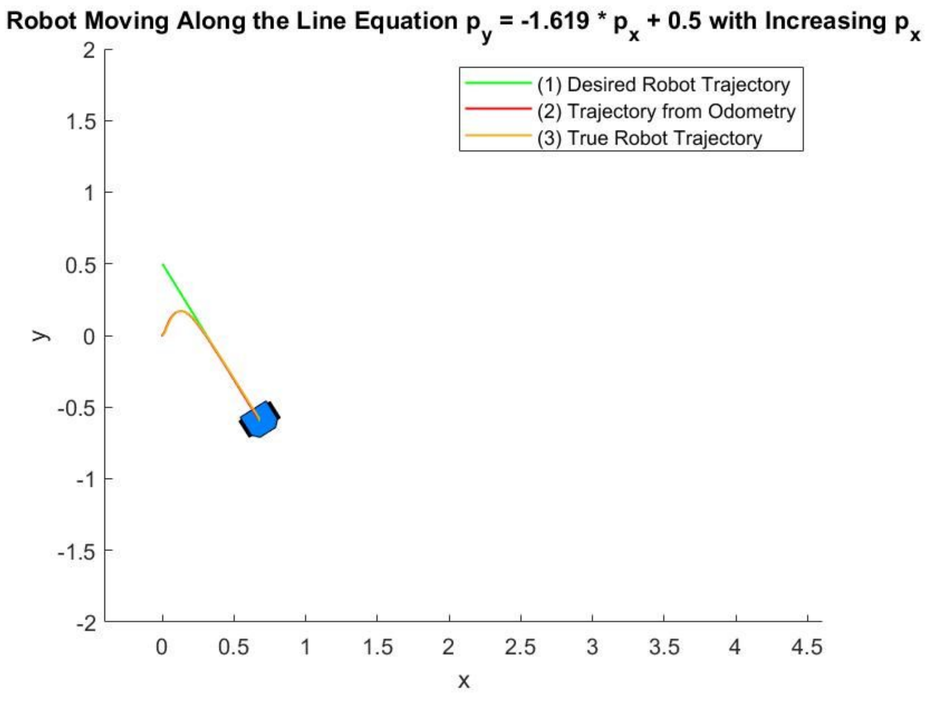
```
title('Robot Moving to the Desired Position (0.3, 0.5) and Returning to the
Original Position')
legend('(2) Trajectory from Odometry', '(1) Desired Robot Trajectory','(3) True
Robot Trajectory')
init_question_4;
% (1) Plot the desired robot trajectory
desiredTrajplot = plot([0 ref_location(1)], [0 ref_location(2)], 'DisplayName',
'(1) Desired Robot Trajectory');
% (2) Plot the robot trajectory estimated from the odometry measurements
```

```
p_x_all = [p_x];
p_y_all = [p_y];
odometryTrajplot = plot(p_x_all, p_y_all,'red', 'linewidth', 1, 'DisplayName', '(2)
Trajectory from Odometry');
% (3) Plot the true robot trajectory
trajplot = plot(csim.Log.Output(2,:),csim.Log.Output(3,:),'linewidth',1 ,
'DisplayName', '(3) True Robot Trajectory');
```

At every time step, the following code is run to plot the curves:

```
controller_question_4;
% (2) Plot the robot trajectory estimated from the odometry
% measurements at every time step
p_x_all(end + 1) = p_x;
p_y_all(end + 1) = p_y;
set(odometryTrajplot, 'XData', p_x_all, 'DisplayName', '(2) Trajectory from
Odometry');
set(odometryTrajplot, 'YData', p_y_all, 'DisplayName', '(2) Trajectory from
Odometry');
```

Plot for Question 5:



This code block from ***DifferentialDriveDCMotorRobot_5.m*** is run at the beginning of the simulation:

```
init_question_5;
desired_x = 0;
```

```
desired_y = m * desired_x + b;
% (1) Desired robot trajectory
desiredTrajplot = plot(desired_x, desired_y,'g','linewidth',1, 'DisplayName', '(1)
Desired Robot Trajectory');
% (2) The robot trajectory estimated from the odometry measurements
p_x_all = [p_x];
p_y_all = [p_y];
odometryTrajplot = plot(p_x_all, p_y_all,'red', 'linewidth', 1, 'DisplayName', '(2)
Trajectory from Odometry');
trajplot = plot(csim.Log.Output(2,:),csim.Log.Output(3,:),'linewidth',1 ,
'DisplayName', '(3) True Robot Trajectory');
```

At every time step, the following code is run to plot the curves:

```
controller_question_5;
% (1) Desired robot trajectory
desired_x = csim.Log.Output(2,:);
desired_y = m * desired_x + b;
set(desiredTrajplot,'XData', desired_x,'DisplayName', '(1) Desired Robot
Trajectory');
set(desiredTrajplot,'YData', desired_y,'DisplayName', '(1) Desired Robot
Trajectory');
% (2) The robot trajectory estimated from the odometry measurements
p_x_all(end + 1) = p_x;
p_y_all(end + 1) = p_y;
set(odometryTrajplot, 'XData', p_x_all, 'DisplayName', '(2) Trajectory from
Odometry');
set(odometryTrajplot, 'YData', p_y_all, 'DisplayName', '(2) Trajectory from
Odometry');
```

Question 7

The objective of question 7 is to implement our own version of a differential drive robot model with two DC motors using a state-space representation, and run our model with 3 different inputs:

$$V_R = V_L$$
$$V_R = 2V_L$$
$$V_R = V_L + V_L sin(t)$$

Where V_R is the right motor and V_L is the left motor, for a duration of 20 seconds with a sampling time of 0.02 seconds and plot their trajectory. This means we will have to create a for loop to loop for 20 seconds with an interval of 0.02 seconds.

To achieve this, we first figure out what control system we are dealing with. For a single DC motor, the state-space equation is:

$$\begin{cases} \dot{x}_{DC}(t) = \begin{bmatrix} \frac{di}{dt} \\ \theta \end{bmatrix} = Ax_{DC}(t) + Bu(t) \\ y_{DC}(t) = \dot{\theta} = Cx_{DC}(t) \end{cases}$$

$$\begin{cases} \dot{x}_{DC}(t) = \begin{bmatrix} -\frac{R}{L} & -\frac{K_e}{L} \\ \frac{K_m}{J} & -\frac{b}{J} \end{bmatrix} \begin{bmatrix} i(t) \\ \dot{\theta}(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V(t) \\ y_{DC}(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} x_{DC}(t) \end{cases}$$

Which, in discrete form, gives:

$$\begin{cases} x[t+1] = e^{A\Delta t}x_{DC}[t] + A^{-1}(e^{A\Delta t} - I)Bu[t] \\ y[t+1] = Cx_{DC}[t] \end{cases}$$

This means in each increment of time, we will get new values of current i and angular velocity $\dot{\theta}$ and we can take this $\dot{\theta}$ to calculate the linear velocity by:

$$v = r\dot{\theta}$$

For this part, the codes are as follows:

```
% DC Motor Model:
% Left:
[x_DC(1:2), theta_dot_L] = DCMotor_StateSpaceModel(SamplingTime, V_L, x_DC(1:2));
s_L = theta_dot_L*r; % Left wheel velocity

% Right:
[x_DC(3:4), theta_dot_R] = DCMotor_StateSpaceModel(SamplingTime, V_R, x_DC(3:4));
s_R = theta_dot_R*r; % Right wheel velocity
```

For the function **DCMotor_StateSpaceModel.m**:

```
% Description: State space model of a single DC Motor
% Input:  V
% Output: theta_dot (Motor angular velocity)
% State:  x_DC = [i; theta_dot]
function [x_DC_dot, theta_dot] = DCMotor_StateSpaceModel(SamplingTime, V, x_DC)
      % Constants:
      R   = 2.0;              % Resistance              (Ohms)
      L   = 0.5;              % Inductance              (Henrys)
      K_m = 0.1;              % Torque constant
      K_e = 0.1;              % Back emf constant
      b   = 0.2;              % Friction coefficient    (Nms)
      J   = 0.02;             % Moment of Inertia       (kg.m^2/s^2)

      A = [-R/L, -K_e/L; K_m/J, -b/J];
      B = [1/L; 0];

      % x[t + 1]  = exp(A*t)*x[delta_t] + Inverse(A)*(exp(A*delta_t) - I)*B*u[t]
      x_DC_dot    = expm(A.*SamplingTime)*x_DC + A\(expm(A.*SamplingTime) -
eye(size(A)))*B*V;

      % y[t + 1]  = [0, 1]*x[t]
      theta_dot   = [0, 1]*x_DC;

end
```

After getting the linear velocity for both the left and right wheel velocity (S_L, S_R), we can use these two values to do the forward kinematics (p_x, p_y). Since our goal is to calculate the next position of the robot (p_x, p_y, ϕ), we need to:

1. Calculate the robot rate of change of orientation $\dot{\phi}$ using:
$$\dot{\phi} = \frac{s_R - s_L}{l_w}$$
2. Calculate the robot centre s_M:
$$s_M = \frac{1}{2}(s_R + s_L)$$
3. Calculate forward kinematics for (p_x, p_y) at time $t+1$:
$$\begin{cases} p_x[t+1] = p_x[t] + \frac{s_M}{\dot{\phi}}[sin(\dot{\phi}\Delta t) - sin(\phi)] \\ p_y[t+1] = p_y[t] - \frac{s_M}{\dot{\phi}}[cos(\dot{\phi}\Delta t) - cos(\phi)] \end{cases}$$

For this part the codes are:

```
% Robot Orientation:
phi_dot = (s_R - s_L)/l_W;

% Robot Centre:
s_M    = 0.5*(s_L + s_R); % Robot centre velocity

% Forward Kinematics:
p_x_dot = s_M*cos(phi);
p_y_dot = s_M*sin(phi);

if phi_dot == 0
        % Handle singular case (phi_dot == 0 --> robot moving in straight line,
        % phi is constant, p_x, p_y is linear:
        p_x = p_x + p_x_dot*SamplingTime;
        p_y = p_y + p_y_dot*SamplingTime;
        phi = phi;
else
        % phi_dot != 0 --> robot turning:
        p_x = p_x + (s_M/phi_dot)*[sin(phi_dot*SamplingTime + phi) - sin(phi)];
        p_y = p_y - (s_M/phi_dot)*[cos(phi_dot*SamplingTime + phi) - cos(phi)];
        phi = phi + phi_dot*SamplingTime;
end
```

For the voltage, since we have three cases, it is handled by a function with 3 different modes:

```
% Description: To choose different modes of voltage supply
% Input:
% t:    Time
% mode: Voltage Input mode
% V_L:  Voltage supply of left wheel
% Output:
% [V_L, V_R]: Voltage supply for left wheel and right wheel
% Voltage input modes:
% mode = 1 --> V_R = V_L
% mode = 2 --> V_R = 2*V_L
% mode = 3 --> V_R = V_L + V_L*sin(t)
function [V_L, V_R] = Voltage(t, V_L, mode)
       if mode == 1
           V_R = V_L;

       elseif mode == 2
           V_R = 2*V_L;

       elseif mode == 3
           V_R = V_L + V_L*sin(t);

       end
end
```

where the mode can be modified at line 14 in **SelfVersionDifferentialDriveRobot_7.m**.

After this, we have to save the new (p_x, p_y) to an array:

```
% Append new position to memory arrays so the trajectory can be plotted
% later:
p_x_mem(end + 1) = p_x;
p_y_mem(end + 1) = p_y;
```

Then concluding the above, we have altogether 11 parameters to initialize, these are stored at **init_7.m**, their meanings are explained in the codes:

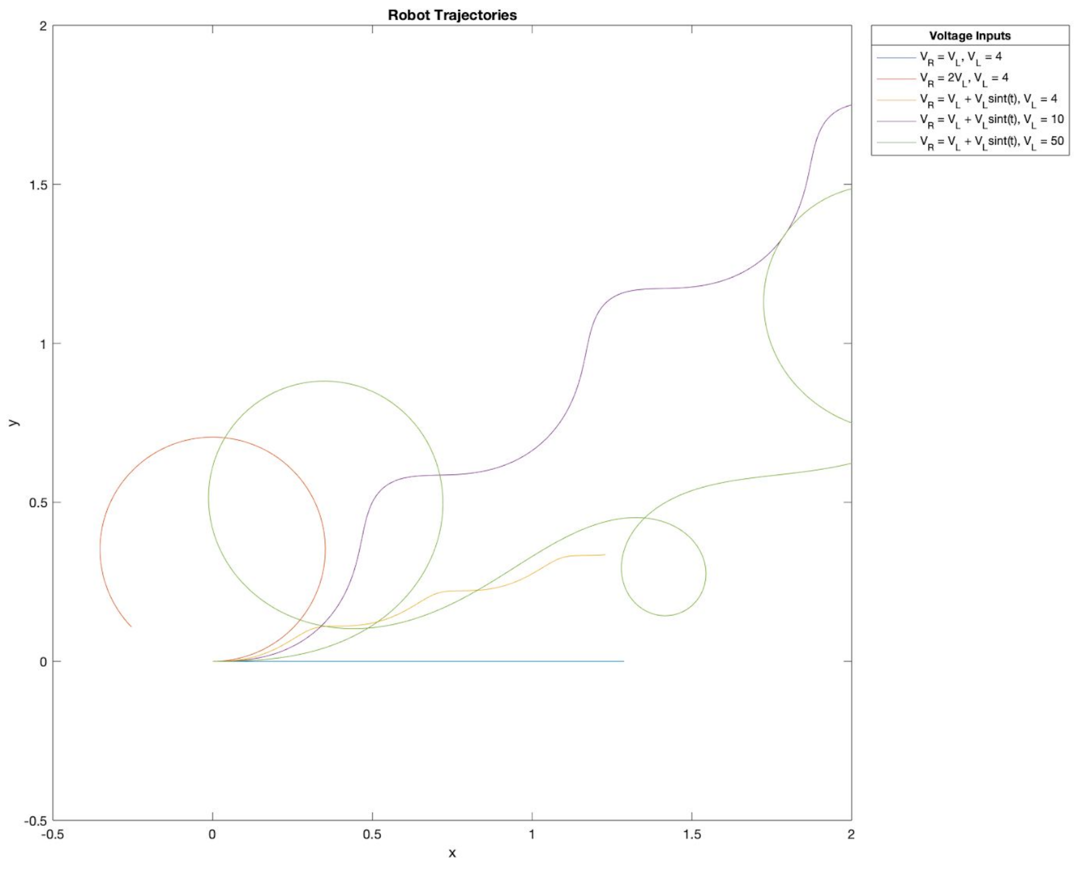
```
r             = 0.0671;    % Wheel radius
l_W           = 0.235;     % Distance between wheels
x_DC          = [0;0;0;0]; % State
phi           = 0;         % Robot orientation
p_x           = 0;         % x-position
p_y           = 0;         % y-position
p_x_mem       = [];        % Array storing the x-position
```

```
p_y_mem      = [];        % Array storing the y-position
V_L          = 50;        % DC Motor Voltage
TotalTime    = 20;        % Total time
SamplingTime = 0.02;      % Sampling time
```

Concluding the above, we composed our version of a differential drive robot model as **SelfVersionDifferentialDriveRobot_7.m** as follows:

```
% Initialize Workspace:
clc;
clear all;
close all;

% Init parameters:
init_7

for t = 0:SamplingTime:TotalTime
      % Voltage:
      % mode = 1 --> V_R = V_L
      % mode = 2 --> V_R = 2*V_L
      % mode = 3 --> V_R = V_L + V_L*sin(t)
      mode = 3;
      [V_L, V_R] = Voltage(t, V_L, mode);

      % DC Motor Model:
      % Left:
      [x_DC(1:2), theta_dot_L] = DCMotor_StateSpaceModel(SamplingTime, V_L,
x_DC(1:2));
      s_L = theta_dot_L*r; % Left wheel velocity

      % Right:
      [x_DC(3:4), theta_dot_R] = DCMotor_StateSpaceModel(SamplingTime, V_R,
x_DC(3:4));
      s_R = theta_dot_R*r; % Right wheel velocity

      % Robot Orientation:
      phi_dot = (s_R - s_L)/l_W;

      % Robot Centre:
      s_M    = 0.5*(s_L + s_R); % Robot centre velocity

      % Forward Kinematics:
      p_x_dot = s_M*cos(phi);
      p_y_dot = s_M*sin(phi);

      if phi_dot == 0
          % Handle singular case (phi_dot == 0 --> robot moving in straight line,
            % phi is constant, p_x, p_y is linear:
            p_x = p_x + p_x_dot*SamplingTime;
            p_y = p_y + p_y_dot*SamplingTime;
            phi = phi;

        else
            % phi_dot != 0 --> robot turning:
            p_x = p_x + (s_M/phi_dot)*[sin(phi_dot*SamplingTime + phi) - sin(phi)];
            p_y = p_y - (s_M/phi_dot)*[cos(phi_dot*SamplingTime + phi) - cos(phi)];
            phi = phi + phi_dot*SamplingTime;

        end

        % Append new position to memory arrays so the trajectory can be plotted
        % later:
        p_x_mem(end + 1) = p_x;
        p_y_mem(end + 1) = p_y;

end

% Plot trajectory:
plot(p_x_mem, p_y_mem)
xlabel('x')
ylabel('y')
```

Finally, The plotted results are as follows:



All in all, in question 7, there are 4 executing files in total. To start the model, run **SelfVersionDifferentialDriveRobot_7.m** and it will use **init_7.m**, **Voltage.m** and **DCMotor_StateSpaceModel.m.**