Robotics – HW 1

Zahra Hashemi – 97243072

Dorreen Rostami – 97243034

Niloufar Moradijam – 97243063

Q1) **zita_dot_robot_frame():**

The inputs of this function: phi1_dot (motor1 angular velocity), phi2_dot (motor2 angular velocity), r (wheel radius), two_l (2l = distance between two wheels)

Output: x_dot (linear velocity of the robot in the direction of X_R), y_dot (linear velocity of the robot in the direction of Y_R which is always 0), teta_dot (angular velocity of the robot)

```
def zita_dot_robot_frame(phi1_dot, phi2_dot, r, two_l): # phi1_dot and phi2_dot are i
n degree/s
  x_dot_R = (r/2) * (phi1_dot + phi2_dot) * Deg2Rad
  y_dot_R = 0
  teta_dot_R = (r/two_l)* (phi1_dot - phi2_dot) * Deg2Rad
  return np.array([x_dot_R, y_dot_R, teta_dot_R])  # result in radian/s
```

change_frame():

If boolean of inertial frame is true, zita_dot needs to be multiplied by R(-teta) otherwise it's multiplies by R(teta)

```
def change_frame(zita_dot, teta, inertialFrame):
  if inertialFrame:             # changing the frame from robot to inertial
    inertialFrame = -1          # R(-teta)
  else:                         # changing the frame from inertial to robot
    inertialFrame = 1           # R(teta)
  teta *= Deg2Rad
  # rotational matrix
  R_matrix = np.array([
    [math.cos(inertialFrame * teta), math.sin(inertialFrame * teta), 0],
    [-1 * math.sin(inertialFrame * teta), math.cos(inertialFrame * teta), 0 ],
    [0, 0, 1]
  ])
  return np.dot(R_matrix, zita_dot)
```

forward_kinematic():

first, finds zita_dot in the robot's frame and converts it into agular velocity in the inertial frame.

```
# Forward kinematics
def forward_kinematic(phi1_dot, phi2_dot, teta, r, two_l):
  return change_frame(zita_dot_robot_frame(phi1_dot, phi2_dot وr, two_l), teta, True)
  # teta is in degree
# Results (x_dot, y_dot, teta_dot) are in radian/s
```

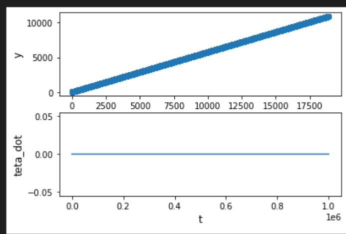
move_forward() and forward_plot():

Is used to trace the robot's movements. It find's the robot's placement for t=1 to t_range by using forward_kinematic. We pass the x and y and angular_v produced to forward_plot, which plots the x-y and angular_v-t plots.

```
def move_forward(phi1, phi2, teta, r, two_l):
  t_range = 1000000
  x0 = 5
  y0 = 5

  # calculating robot's velocity
  v = forward_kinematic(phi1, phi2, teta, r, two_l)
  v_linear = []
  v_linear.append(v[0])
  v_angular = []
  v_angular.append(v[2])

  # robot's coordinate
  x = []
  x.append(x0)
  y = []
  y.append(y0)

  # robot's velocity in robot frame
  v_local = change_frame(v, teta, False)
  v_linear = [v_local[0]] * t_range
  v_angular = [v_local[2]] * t_range

  # calculating robot's coordinate
  for i in range(1, t_range):
    x.append(x[i-1]+(DeltaT*v[0]))
    y.append(y[i-1]+(DeltaT*v[1]))
    teta += v[2]*DeltaT
    v = forward_kinematic(phi1, phi2, teta, r, two_l)

  forward_plot(x, y, v_angular, t_range)


def forward_plot(x, y, angular_v, t_range):
  # evenly sampled time at 1sec intervals
  t = np.arange(0., t_range, 1)
  fig, (ax1) = plt.subplots(1, 1)
  # x - y
  ax1.plot(x, y)
  ax1.set_xlabel('x', fontsize=12)
  ax1.set_ylabel('y', fontsize=12)
  fig, (ax2) = plt.subplots(1, 1)
  # teta_dot - t
  ax2.plot(t, angular_v)
  ax2.set_xlabel('t', fontsize=12)
  ax2.set_ylabel('teta_dot', fontsize=12)
```

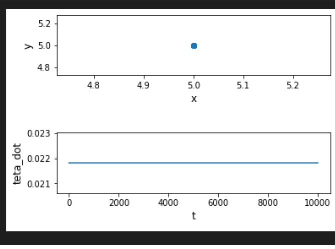
[56]

```python
# phi1_dot = phi2_dot = const
move_forward(5, 5, 30, 2.5, 20)
```



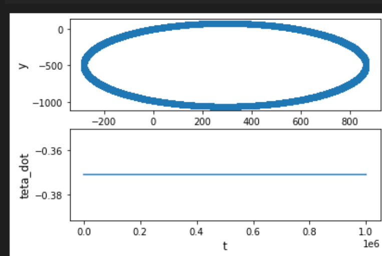
[45]

```python
# phi1_dot = const, phi2_dot = - const
move_forward(5, -5, 30, 2.5, 20)
```



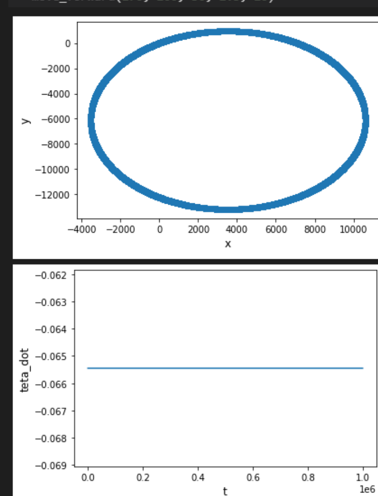
[58]

```python
# phi1_dot = 0, phi2_dot = const
move_forward(0, 170, 30, 2.5, 20)
```



[63]

```python
# phi1_dot < phi2_dot
move_forward(170, 200, 30, 2.5, 20)
```



Q2) **wheels_velocity():**

Uses zita_dot_r to find phi1_dot and phi2_dot

$$\dot{\xi}_R = \begin{bmatrix} \dot{x}_R \\ 0 \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r\dot{\varphi}_1}{2} + \frac{r\dot{\varphi}_2}{2} \\ 0 \\ \frac{r\dot{\varphi}_1}{2l} + \frac{-r\dot{\varphi}_2}{2l} \end{bmatrix}$$

```
def wheels_velocity(matrix, r, two_l):
  phi1_dot, phi2_dot = symbols('phi1_dot phi2_dot')
  eq1 = Eq(matrix[0], (r/2)*(phi1_dot+phi2_dot))
  eq2 = Eq(matrix[2]*Deg2Rad, (r/two_l)*(phi1_dot-phi2_dot))
  sol = solve((eq1, eq2),(phi1_dot, phi2_dot))
  return sol # phi1_dot and phi2_dot are in radian/s
```

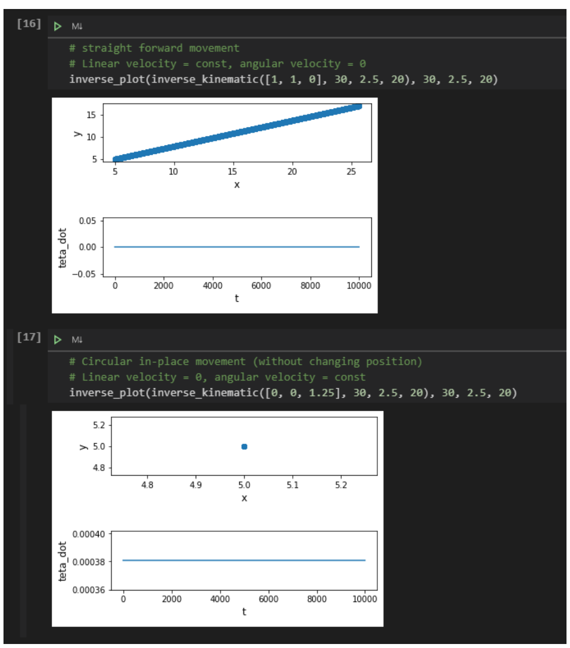
$$\begin{cases} \dot{x}_R = \dfrac{r\dot{\phi}_1}{2} + \dfrac{r\dot{\phi}_2}{2} \\ \dot{\theta} = \dfrac{r\dot{\phi}_1}{2l} - \dfrac{r\dot{\phi}_2}{2l} \end{cases}$$

$$\dot{\xi}_I = R(\theta)^{-1} \begin{bmatrix} \frac{r\dot{\varphi}_1}{2} + \frac{r\dot{\varphi}_2}{2} \\ 0 \\ \frac{r\dot{\varphi}_1}{2l} + \frac{-r\dot{\varphi}_2}{2l} \end{bmatrix}$$

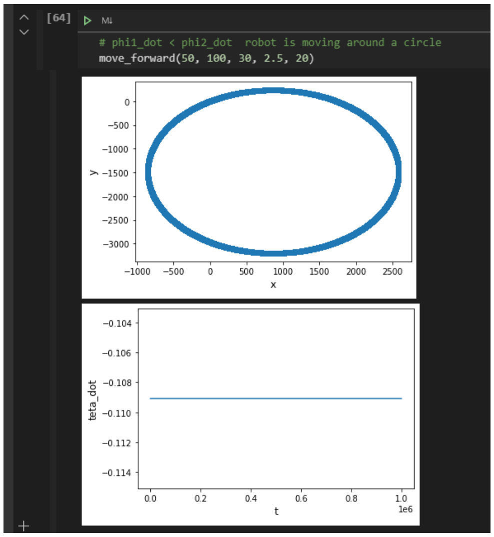
inverse_kinematic():

```
def inverse_kinematic(matrix, teta, r, two_l):
  return wheels_velocity(change_frame(matrix, teta, False), r, two_l)
```

gets zita_dot_i as input and after converting them into the velocities in the robot frame using change_frame, zita_dor_r is produced which will be the input of wheels_velocity.



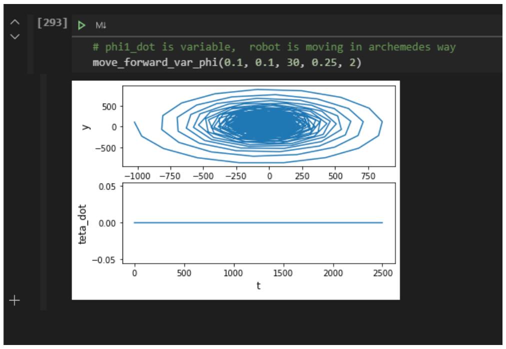
Q3) As we’ve seen before, to make a circle, one wheel needs have an angular velocity bigger than the other and in the same direction. We can do this using the move_forward function which already exists.



And to make the robot move in a spiral, one wheel’s angular velocity needs to increase by at (t = current time). To do this we use the following function, which is the move_forward function, but it changes phi1 before each time calculating the robot’s next location.

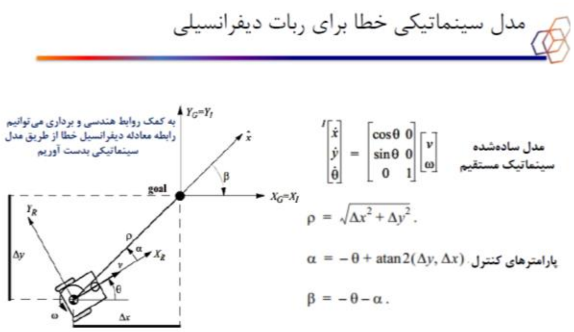
```
# Tracing robot's movement
def move_forward_var_phi(phi1, phi2, teta, r, two_l):
  t_range = 2500
  x0 = 5
  y0 = 5

  # calculating robot's velocity
  v = forward_kinematic(phi1, phi2, teta, r, two_l)
  v_linear = []
  v_linear.append(v[0])
  v_angular = []
  v_angular.append(v[2])

  # robot's coordinate
  x = []
  x.append(x0)
  y = []
  y.append(y0)

  # robot's velocity in robot frame
  v_local = change_frame(v, teta, False)
  v_linear = [v_local[0]] * t_range
  v_angular = [v_local[2]] * t_range


  # calculating robot's coordinate
  for i in range(1, t_range):
    x.append(x[i-1]+(DeltaT*v[0]))
    y.append(y[i-1]+(DeltaT*v[1]))
    teta += v[2]*DeltaT
```

```
    phi1 += 0.5 * i
    v = forward_kinematic(phi1, phi2, teta, r, two_l)

  forward_plot(x, y, v_linear, v_angular, t_range)
```



Q4) **velocity_calculation():**

This function takes the robot's initial position and destination and calculates its needed velocity to reach the destination. We can calculate alpha, beta, rho using the equations in the slide shown below. We have to choose values for k_rho, k_alpha and k_beta that satisfies the rules commented in front of them (the rules needed for stability).




```
def velocity_calculation(x_i, y_i, x_f, y_f, teta):
  teta *= Deg2Rad
  rho = math.sqrt(math.pow(x_f - x_i, 2) + math.pow(y_f - y_i, 2))
  alpha = -1 * teta + math.atan2((y_f - y_i), (x_f - x_i))
  beta = -1 * teta - alpha # beta

  k_rho = 0.5   # k_rho > 0
  k_alpha = 3000  # k_alpha > k_rho
  k_beta = -1000 # k_beta < 0

  # linear velocity
  v = k_rho * rho
  # angular velocity
  w = k_alpha * alpha + k_beta * beta

  return [v, 0, w]
```

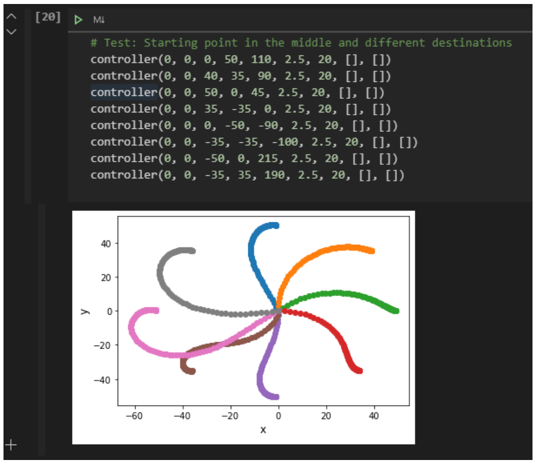
controller():

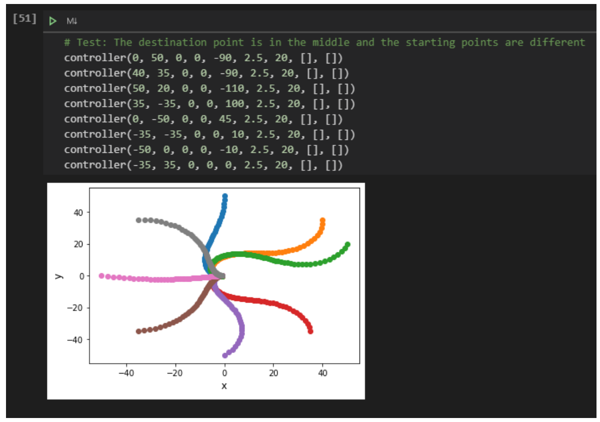
this is the main method for this part. It takes the robot's initial position, destination, the wheels' radii, and the length of the shaft between the wheels (2l), and calculates the robot's trajectory to the destination (its coordination at each point is stored in two arrays). It first calculates robot's velocity in the robot frame, and then calls inverse_kinematic to calculate the wheels' velocity. After these steps, it calculates the robot's next destination by using forward_kinematic.

This function is a recursive function that calls itself until the input initial coordination is a point close to the destination. To avoid errors, we check this using a threshold for the distance between current initial position and destination. Since it is a recursive function, it stores the coordination at each point in the said lists every time it is called.

```
def controller(x_i, y_i, x_f, y_f, teta, r, two_l, x_list, y_list):

  x_list.append(x_i)
  y_list.append(y_i)

  if (abs(x_i - x_f) < 1) and (abs(y_i - y_f) < 1):
    # the robot's movement path
    plt.plot(x_list, y_list, 'o')
    plt.xlabel('x', fontsize=12)
    plt.ylabel('y', fontsize=12)
    return

  # Calculating robot's velocity in robot frame
  matrix = change_frame(velocity_calculation(x_i, y_i, x_f, y_f, teta), teta, True)

  # Inverse kinematic to calculate wheels' velocity
  phi1_dot, phi2_dot = symbols('phi1_dot phi2_dot')
  # wheels' velocity in radian/s
  wheel_velocity = inverse_kinematic(matrix, teta, r, two_l)
  # wheels' velocity in degree/s
  phi1_dot = wheel_velocity[phi1_dot]/Deg2Rad
  phi2_dot = wheel_velocity[phi2_dot]/Deg2Rad

  # calculate robot's next destination
  zita_dot = forward_kinematic(phi1_dot, phi2_dot, teta, r, two_l) # x_dot, y_dot, te
ta_dot
  x_i += DeltaT * zita_dot[0]
  y_i += DeltaT * zita_dot[1]
  teta += DeltaT * zita_dot[2]

  controller(x_i, y_i, x_f, y_f, teta, r, two_l, x_list, y_list)
```

the output plots for this part (how the robot actually moves/the blue line mentioned in the question):





The desired path (red line) would be a straight line between the initial and final position (which we have not plotted)

As we see in the above plots (how the robot actually moves), as the robot gets closer to the destination, it moves with smaller steps. This phenomenon is the result of how k_rho, k_alpha and k_beta have values that satisfy the rules needed for stability.