==
Delta Parallel Robot Documentation

Edition: #1
Date: April-18-2023
Author: Arvin Mohammadi

Log:
[Delta Robot class]

==

TABLE OF CONTENT

File name: delta_robot.py... 3
File name: path_planning.py.. 7

File name: delta_robot.py

This file contains [1 class(es), 3 function(s)]:

- DeltaRobot

> FUNCTION: tand

```
def tand(theta):
      return tan(theta*pi/180)
```

Input: angle in degrees θ
Output: $\tan(\theta)$

> FUNCTION: sind

```
def sind(theta):
      return sin(theta*pi/180)
```

Input: angle in degrees θ
Output: $\sin(\theta)$

> FUNCTION: cosd

```
def cosd(theta):
      return cos(theta*pi/180)
```

Input: angle in degrees θ
Output: $\cos(\theta)$

> Class: DeltaRobot

```
class DeltaRobot:
      def __init__(self, rod_b, rod_ee, r_b, r_ee):
            # configs the robot

            self.rod_b = rod_b
            self.rod_ee = rod_ee
            self.r_b = r_b
            self.r_ee = r_ee
            self.alpha = np.array([0, 120, 240])
```

```python
def forward_kin(self, theta):
        # calculate FK, takes theta(deg)

        rod_b = self.rod_b
        rod_ee = self.rod_ee

        theta = np.array(theta)

        theta1 = theta[0]
        theta2 = theta[1]
        theta3 = theta[2]

        side_ee       = 2/tand(30)*self.r_ee
        side_b        = 2/tand(30)*self.r_b

        t = (side_b - side_ee)*tand(30)/2

        y1 = -(t + rod_b*cosd(theta1))
        z1 = -rod_b*sind(theta1)

        y2 = (t + rod_b*cosd(theta2))*sind(30)
        x2 = y2*tand(60)
        z2 = -rod_b*sind(theta2)

        y3 = (t + rod_b*cosd(theta3))*sind(30)
        x3 = -y3*tand(60)
        z3 = -rod_b*sind(theta3)

        dnm = (y2 - y1)*x3 - (y3 - y1)*x2

        w1 = y1**2 + z1**2
        w2 = x2**2 + y2**2 + z2**2
        w3 = x3**2 + y3**2 + z3**2

        a1 = (z2-z1)*(y3-y1) - (z3-z1)*(y2-y1)
        b1 = -((w2-w1)*(y3-y1) - (w3-w1)*(y2-y1))/2

        a2 = -(z2-z1)*x3 + (z3-z1)*x2
        b2 = ((w2-w1)*x3 - (w3-w1)*x2)/2

        a = a1**2 + a2**2 + dnm**2
        b = 2*(a1*b1 + a2*(b2-y1*dnm) - z1*dnm**2)
        c = (b2 - y1*dnm)**2 + b1**2 + dnm**2*(z1**2 - rod_ee**2)

        d = b**2 - 4*a*c
```

```
        if d < 0:
            return -1

        z0 = -0.5*(b + d**0.5)/a
        x0 = (a1*z0 + b1)/dnm
        y0 = (a2*z0 + b2)/dnm

        return np.array([x0, y0, z0])

    def inverse_kin(self, _3d_pose):
        # calculates IK, returns theta(deg)
        [x0, y0, z0] = _3d_pose

        rod_ee = self.rod_ee
        rod_b = self.rod_b
        r_ee = self.r_ee
        r_b = self.r_b
        alpha = self.alpha

        F1_pos = ([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        J1_pos = ([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        theta = [0, 0, 0]

        for i in [0, 1, 2]:

            x = x0*cosd(alpha[i]) + y0*sind(alpha[i])
            y = -x0*sind(alpha[i]) + y0*cosd(alpha[i])
            z = z0

            ee_pos = np.array([x, y, z])

            E1_pos = ee_pos + np.array([0, -r_ee, 0])
            E1_prime_pos = np.array([0, E1_pos[1], E1_pos[2]])
            F1_pos[i] = np.array([0, -r_b, 0])

            _x0 = E1_pos[0]
            _y0 = E1_pos[1]
            _z0 = E1_pos[2]
            _yf = F1_pos[i][1]

            c1 = (_x0**2 + _y0**2 + _z0**2 + rod_b**2 - rod_ee**2 -
_yf**2)/(2*_z0)
            c2 = (_yf - _y0)/_z0
            c3 = -(c1 + c2*_yf)**2 + (c2**2+ 1)*rod_b**2

            if c3 < 0:
                print("non existing point")
```

```
                return -1

            J1_y = (_yf - c1*c2 - c3**0.5)/(c2**2 + 1)
            J1_z = c1 + c2*J1_y
            F1_y = -r_b

            theta[i] = math.atan(-J1_z/(F1_y - J1_y))*180/pi

        return np.array(theta)
```

>> *Method: init*

Input: rod_b, rod_ee, r_b, r_ee
Output: -

This method initializes the `DeltaRobot` class. The inputs are:

- `rod_b`: the length of the active arm that is attached to the base
- `rode_ee`: the length of the passive arm that is attached to the end effector
- `r_b`: radius of base (distance from center of the base to an actuator)
- `r_ee`: radius of end effector (distance from center of base to a pin)
- `alpha`: degree of each motor in relation to each other (default 120)

>> *Method: forward_kin*

Input: theta
Output: pos

This method calculates the forward kinematics of the robot:

- theta: a three element array - [theta1, theta2, theta3] (angles of each actuator)
- pos: a three element array - [x0, y0, z0] (positions of the end effector in 3D space)

The full documentation of how it is calculated can be found in the following link:

https://github.com/ArthasMenethil-A/Delta_Robot/blob/main/theory/Inverse%20Kinematics%20(Delta%20Robot).pdf

>> *Method: inverse_kin*

Input: _3d_pose
Output: theta
This method calculates the inverse kinematic of the robot.

- _3d_pose: a three element array - [x0, y0, z0] (positions of the end effector in 3D space)
- theta: a three element array - [theta1, theta2, theta3] (angles of each actuator)

File name: path_planning.py

This file contains [1 class(es), 0 function(s)]:

- PathPlannerPTP

```
class PathPlannerPTP:
	def __init__(self, ee_pos_i, ee_pos_f, theta_dot_max):
		self.ee_pos_i	= np.array(ee_pos_i)
		self.ee_pos_f	= np.array(ee_pos_f)
		self.theta_i = np.zeros((3, 1))
		self.theta_f = np.zeros((3, 1))
		self.theta_dot_max  = theta_dot_max*6 # convert rpm to deg/s

	def point_to_point_467(self, robot):
		self.theta_i = robot.inverse_kin(self.ee_pos_i).reshape((3, 1))
		self.theta_f = robot.inverse_kin(self.ee_pos_f).reshape((3, 1))

		FREQUENCY = 1000
		# overall time period
		T = 35/16*(self.theta_f - self.theta_i)/self.theta_dot_max
		T = math.floor(max(T)*FREQUENCY)
		tau = np.array(range(0, T))/T

		# theta time profile
		s_tau = -20*tau**7 + 70*tau**6 - 84*tau**5 + 35*tau**4
		theta_t = np.array(self.theta_i) + np.array(self.theta_f -
self.theta_i)*s_tau

		# theta dot time profile
		s_tau_d = -140*tau**6 + 420*tau**5 - 420*tau**4 + 140*tau**3
		theta_dot_t = np.array(self.theta_f - self.theta_i)/T*s_tau_d

		# checking the forward kinematics
		ee_pos_t = np.zeros(theta_t.shape)
		for idx, i in enumerate(theta_t.transpose()):
			ee_pos_t[:, idx] = robot.forward_kin(theta_t[:, idx])

		# plot theta_t
		plt.grid(True)
		plt.plot(tau, theta_t.transpose(), label=['theta_1', 'theta_2',
'theta_3'])
		plt.title("angle-time plot")
```

```
            plt.legend()
            plt.xlabel("normalized time")
            plt.ylabel("angle theta (deg)")
            plt.savefig("4567-theta.png")
            plt.clf()

            # plot theta_dot_t
            plt.grid(True)
            plt.plot(tau, theta_dot_t.transpose(), label=['theta_dot_1',
'theta_dot_2', 'theta_dot_3'])
            plt.title("angular velocity-time plot")
            plt.legend()
            plt.xlabel("normalized time")
            plt.ylabel("angular velocity theta_dot (deg/s)")
            plt.savefig("4567-theta-dot.png")
            plt.clf()

            # plot EE-position
            plt.grid(True)
            plt.plot(tau, ee_pos_t.transpose(), label=['x', 'y', 'z'])
            plt.title("position-time plot")
            plt.legend()
            plt.xlabel("normalized time")
            plt.ylabel("EE position (m)")
            plt.savefig("4567-ee-position.png")
            plt.clf()
```

> *class: PathPlannerPTP*

This class contains the point-to-point path planning methods.

>> *method: __init__*

Inputs: ee_pos_i, ee_pos_f, theta_dot_max
Outputs: -

This initializes the path planner class and the following values:
- Ee_pos_i and Ee_pos_f: the initial and final values of the end effector positions
- Theta_i and Theta_f: the initial and final values of actuator angles
- Theta_dot_max: the max angular velocity in rpm (then it is converted to deg/s)

>> *method: point_to_point_467*

Inputs: robot
Outputs: -

This creates some plots for us:

- robot: robot object (for calculating the inverse kinematics)

This method outputs 3 plots:

- Actuator angle against time
- Actuator angular velocity against time
- Actuator angular acceleration against time