Robotics Project – ΑΜΙΤΣΗΣ ΛΕΥΤΕΡΗΣ

PITask class:

- Το πρόγραμμα ξεκινάει με την δημιουργία της κλάσης του controller
- **init function:** δίνουμε τιμές στης μεταβλητές που θα χρησιμοποιήσουμε στα επόμενα functions, το dt που είναι η μεταβολή του χρόνου των υπολογισμό των παραμέτρων Kp που στην ουσία είναι η ποσότητα που καθορίζει πως θα ανταποκριθεί ο ελεγκτής στο σφάλμα δηλαδή πόσο γρήγορα η αργά θα το διορθώσει και Ki που αντιπροσωπεύει τον ρυθμό μείωσης των σφαλμάτων που έχουν συσσωρευτεί κατά την διάρκεια της επιθυμητής κίνησης.
- **target function**: Περνάμε στην κλάση του ελεγκτή το tf της “τοποθεσίας ” στην οποία θέλουμε να καταλήξει ο end-effector.

```
class PITask:
    def __init__(self, target, dt, Kp , Ki ):
        self._target = target
        self._dt = dt
        self._Kp = Kp
        self._Ki = Ki
        self._sum_error = 0

    def set_target(self, target):
        self._target = target
```

- **error function:** Στο function αυτό υπολογίζουμε το σφάλμα στο world frame απευθείας παίρνοντας ξεχωριστά το translation και το rotation που θέλουμε να καταλήξουμε και τα

πολλαπλασιάζουμε με το transpose του transformation που βρισκόμαστε. Τέλος επιστρέφουμε ένα array με το συνολικό(και rotation και translation) σφάλμα.

- **update function:** Εδώ ξεκινάμε δίνοντας στην μεταβλητή error_in_the_world_frame το error που υπολογίσαμε στο error function το αθροίζουμε με το συνολικό σφάλμα αφού το πολλαπλασιάσουμε με το dt και κάνουμε return τον controller (self._Kp * error_in_world_frame + self._Ki * self._sum_error) όπου error_in_world_frame είναι το τωρινό σφάλμα και το sum το συσσωρευμένο.

```
def error(self, tf):
    rot_error = rd.math.logMap(self._target.rotation() @ tf.rotation().T)
    lin_error = self._target.translation() - tf.translation()
    return np.r_[rot_error, lin_error]

def update(self, current):
    error_in_world_frame = self.error(current)

    self._sum_error = self._sum_error + error_in_world_frame * self._dt

    return self._Kp * error_in_world_frame + self._Ki * self._sum_error
```

Εδώ δίνουμε τιμές στο dt,simulation time και total steps. Για λόγους υλοποίησης αύξησα το simulation time παρόλο που ζητείται να μην πειραχτεί. Μετά την επεξήγηση της κλάσης myRobot θα εξηγήσω γιατί.

```
dt = 0.001 # you are NOT allowed to change
simulation_time = 27.0 # you are allowed to
total_steps = int(simulation_time / dt)
```

Παρακάτω όπως φαίνεται είναι ο κώδικας που μας δόθηκε για τη δημιουργία των κουτιών και η τοποθέτηση τους στον χώρο και ύστερα η τυχαία επιλογή προβλήματος, δηλαδή η σειρά με την οποία πρέπει να επιλεχθούν τα κουτιά και πως πρέπει να στοιβαχθούν .

```
# Red Box
# Random cube position
red_box_pt = np.random.choice(len(box_positions))
box_pose = [0., 0., 0., box_positions[red_box_pt][0], box_positions[red_box_pt][1], box_size[2] / 2.0]
red_box = rd.Robot.create_box(box_size, box_pose, "free", 0.1, [0.9, 0.1, 0.1, 1.0], "red_box")

# Green Box
# Random cube position
green_box_pt = np.random.choice(len(box_positions))
while green_box_pt == red_box_pt:
    green_box_pt = np.random.choice(len(box_positions))
box_pose = [0., 0., 0., box_positions[green_box_pt][0], box_positions[green_box_pt][1], box_size[2] / 2.0]
green_box = rd.Robot.create_box(box_size, box_pose, "free", 0.1, [0.1, 0.9, 0.1, 1.0], "green_box")

# Blue Box
# Random cube position
box_pt = np.random.choice(len(box_positions))
while box_pt == green_box_pt or box_pt == red_box_pt:
    box_pt = np.random.choice(len(box_positions))
box_pose = [0., 0., 0., box_positions[box_pt][0], box_positions[box_pt][1], box_size[2] / 2.0]
blue_box = rd.Robot.create_box(box_size, box_pose, "free", 0.1, [0.1, 0.1, 0.9, 1.0], "blue_box")
```

```
problems = create_problems()
problem_id = np.random.choice(len(problems))
problem = problems[problem_id]
```

MyRobot class:

- Στην κλάση αυτή περιέχεται η αρχικοποίηση του ρομπότ(Franka) , τη θέση του καθώς και τις εντολές που θέτουν τα πάνω και κάτω όρια των δυνάμεων που μπορούν να του ασκηθούν μεταξύ του finger1 και finger2 . Του δίνουμε αεροκινητήρες και ξεκινάμε να βρούμε το αρχικό transformation του end effector δίνουμε τιμές στα dt,kp,ki και κάνουμε
- **function close:** Σε αυτό το function κλείνουμε το finger1(το finger2 ακολουθεί) για να πιάσουμε τον κύβο.

- **function open:** Σε αυτό το function ανοίγουμε το finger1(το finger2 ακολουθεί) για να ελευθερώσουμε τον κύβο.
- **function update:** Εδώ γίνεται η διαδικασία του task-space control. Αρχικά παίρνουμε το transform στο οποίο θέλουμε να πάμε, παίρνουμε το velocity καλώντας το function update της κλάσης controller, ύστερα παίρνουμε τον jacobian της θέσης (στο world frame) για να υπολογίσουμε τον psedoinverse και να εκτελέσουμε την εντολή jac_pinv @ vel που θα υλοποιήσει την μετακίνηση του ρομπότ.
- **function reached_target:** Στο function αυτό καλούμε την συνάρτηση error του controller προκειμένου να ελέγξουμε το σφάλμα κατά την αλλαγή κατάστασης στο fsm. Έτσι δεν αλλάζει κατάσταση μέχρι το σφάλμα να ελαχιστοποιηθεί.

  Σε αυτό το σημείο να αναφέρω πως όταν το ρομπότ άλλαζε καταστάσεις πριν προλάβει να κλείσει και να ανοίξει ο γάντζος οπότε εκμεταλλεύτηκα την καθυστέρηση που έδινε ο υπολογισμός του σφάλματος για να προσθέσω μερικές καταστάσεις στις οποίες αλλάζω ελάχιστα την θέση του ee έτσι ώστε να προλάβει να ανοίξει ή να κλείσει. Επίσης προστέθηκαν κάποιες καταστάσεις έτσι ώστε ο ee να προσεγγίζει τα τουβλάκια από ψηλά και να μην παρασέρνει στην διαδρομή του αυτά που δεν έχει πιάσει ακόμα ή ακόμα και την στοίβα. Για τον λόγο αυτό αύξησα και το simulation time προκειμένου η διαδικασία να προλάβει να τερματίσει.

  Έπειτα ακολουθούν οι καταστάσεις από τις περνάει το ρομπότ μέσω του fsm.

- **on_enter_A:** Στην πραγματικότητα είναι η θέση στην οποία θα καταλήξει ο ee μετά το πέρας ολόκληρης της διαδικασίας και

στην ουσία απλά κάνουμε control τον ee ώστε να πάει σε ένα σημείο λίγο διαφορετικό από την αρχική του θέση.

- **on_enter_B:** Εδώ ξεκινάμε αρχικοποιώντας μία λίστα με τους αριθμούς 2 και 3 προκειμένου να αλλάζουμε το ύψος στο οποίο θα αφήσουμε το τουβλάκι για κάθε πέρασμα του B και έπειτα μπαίνουμε σε μια if προκειμένου να μπορούν να υλοποιηθούν όλα τα πιθανά προβλήματα. Τέλος δίνουμε στην μεταβλητή controller την κλάσης ένα νέο αντικείμενο PITask για να γίνει η εκ νέου κίνηση.
- **on_enter_E/G:** Παρόμοια με την B μπαίνουμε σε μια if για κάθε πιθανό χρώμα (blue,red,green) και δημιουργούμαι ένα νέο αντικείμενο PITask.

  Όλες οι υπόλοιπες καταστάσεις υλοποιούν το κλείσιμο και το άνοιγμα του γάντζου καθώς και κινήσεις για την ομαλότερη ολοκλήρωση της διαδικασίας. Ενδεικτικά ακολουθεί η επεξήγηση της F.

- **on_enter_F:** Στην κατάσταση αυτή δίνουμε στην μεταβλητή x την τιμή 1 που θα επιτρέψει στο simulator να εκτελέσει την close function και μεταβάλουμε ελάχιστα το translate της προηγούμενης κίνησης, σε αυτή την περίπτωση της κατάστασης E, στον άξονα z προκειμένου να γίνει σώστο κλείσιμο του ee.

Έπειτα ακολουθεί η δημιουργία των γραφικών και το configuration του αντικειμένου προσημείωσης.

```
gconfig = rd.gui.Graphics.default_configuration()
gconfig.width = 1280 # you can change the graphics
gconfig.height = 960 # you can change the graphics
graphics = rd.gui.Graphics(gconfig)

# Create simulator object
simu = rd.RobotDARTSimu(dt)
simu.set_collision_detector("fcl") # you can use b
simu.set_control_freq(100)
simu.set_graphics(graphics)
graphics.look_at((0., 4.5, 2.5), (0., 0., 0.25))
simu.add_checkerboard_floor()
simu.add_robot(robot.robot)
simu.add_robot(red_box)
simu.add_robot(blue_box)
simu.add_robot(green_box)
simu.add_visual_robot(goal_robot)
```

Ύστερα δηλώνουμε τα states και δίνουμε το fsm ,δηλαδή την σειρά με την οποία θα υλοποιηθούν οι καταστάσεις και υπό ποια συνθήκη θα έχουν φτάσει στον στόχο τους . Φτιάχνουμε το αντικείμενο machine και ξεκινάμε το simulation.

```
FSM_states = ['A', 'B', 'C','D','E','F','G','H','I','J']

# And some transitions between states. We're lazy, so we'll leave out
# the inverse phase transitions (freezing, condensation, etc.).
FSM_transitions = [
    { 'trigger': 'arrived', 'source' : 'A', 'dest': 'E', 'conditions' : 'reached_target'},
    { 'trigger': 'arrived', 'source' : 'E', 'dest': 'F', 'conditions' : 'reached_target'},
    { 'trigger': 'arrived', 'source' : 'F', 'dest': 'B', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'B', 'dest': 'C', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'C', 'dest': 'D', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'D', 'dest': 'G', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'G', 'dest': 'I', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'I', 'dest': 'H', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'H', 'dest': 'J', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'J', 'dest': 'B', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'B', 'dest': 'C', 'conditions' : 'reached_target'},
    {'trigger' : 'arrived', 'source' : 'C', 'dest': 'A', 'conditions' : 'reached_target'}

]

machine = Machine(model=robot, states=FSM_states, transitions=FSM_transitions, initial='A')
```

Στο simulation αρχικά καλούμαι το update της κλάσης robot όπου θα μετακινήσει το ρομπότ βάση του τελευταίου PITask αντικειμένου που έχει λάβει ως τιμή η μεταβλητή controller της κλάσης robot έπειτα έχουμε δύο if για τις περιπτώσης που κλείνουμε και ανοίγουμε τον γάντζο και τέλος arrived για τον έλεγχο κατάστασης και μετάβαση στην επόμενη αν έχει ολοκληρωθεί.

```
for step in range(total_steps):
    if (simu.schedule(simu.control_freq())):
        goal_robot.set_base_pose(robot.target())


        robot.update()


        if robot.x==1:
            robot.close()
            #wait=1
        else:
            print("")
        # tick FSM
        if robot.y==1:
            robot.open()
            #wait=1
        else:
            print("")

        robot.arrived()
        pass

    if (simu.step_world()):
        break
```