ΡΟΜΠΟΤΙΚΑ ΣΥΣΤΗΜΑΤΑ Ι

2η ΑΣΚΗΣΗ 2023-2024

Ονοματεπώνυμο: Γεώργιος Παπουτσάς

ΑΜ: 1083738

Έτος: 4ο

Contents

GitHub repository: https://github.com/Papiqulos/Ergasia_2_RSI

Modeling and Simulating the Franka Panda manipulator:

Δημιουργήθηκε μια κλάση Franka για όλες τις ακόλουθες λειτουργιές που απαιτούνται

```python

class Franka:
    """Franka Panda robot class"""

    def __init__(self):
        self.robot, self.model, self.data = self.load_franka()

        # Initialize the visualizer
        self.robot.setVisualizer(VISUALIZER())
        self.robot.initViewer()
        self.robot.loadViewerModel("pinocchio")

    def step_world(self, current_q:np.ndarray, current_u:np.ndarray

    def load_franka(self)->tuple[RobotWrapper, pin.Model, pin.Data,

    def simulate(self, q:np.ndarray, u:np.ndarray, control_t:np.nda

    def get_pose_profile(self, target_q:np.ndarray)->pin.SE3: ···

    def visualize(self, qs:np.ndarray): ···
```

Loading the robot:

Φορτώνουμε όλα τα απαραίτητα δεδομένα του βραχίονα στην **Frank.load_franka()**:

```
def load_franka(self)->tuple[RobotWrapper, pin.Model, pin.Data, pin.GeometryModel,
pin.GeometryData]:
        """
        Load the Franka Panda Panda robot model

        Returns:
        - robot: robot wrapper
        - model: pinocchio model
        - data: pinocchio data
        """
        # Load the URDF model
        current_path = os.path.abspath('') # where the folder `robot` is located
at
        robot_path = os.path.join(current_path, "robot")

        # Read URDF model
        robot = RobotWrapper.BuildFromURDF(os.path.join(robot_path,
"franka.urdf"), package_dirs = robot_path)

        # Extract pinocchio model and data
        model = robot.model
        data = robot.data

        return robot, model, data
```

Stepping into the world:

Η **Franka.step_world()** κάνει ένα βήμα dt για το εκάστοτε σήμα ελέγχου ροπής:

```
def step_world(self, current_q:np.ndarray, current_u:np.ndarray,
control_t:np.ndarray, dt:float)->tuple[np.ndarray, np.ndarray]:
        """
        Step the world for a given time step dt

        Args:
        - model: pinocchio model
        - data: pinocchio data
        - current_q: current position of each joint (angles of each joint)
        - current_u: current velocity of each joint
        - control_t: control torque to be applied at each joint
        - dt: time step

        Returns:
        - new_q: new position of each joint
        - current_t: new velocity of each joint
        """
        # Get joint limits
        joint_limit_up = self.model.upperPositionLimit
        joint_limit_low = self.model.lowerPositionLimit
        velocity_limit = self.model.velocityLimit

        # Integrate the dynamics and get the acceleration
        aq = pin.aba(self.model, self.data, current_q, current_u, control_t)

        # Integrate the acceleration to get the new velocity
        current_u += aq * dt

        # Integrate the velocity to get the new position
        new_q = pin.integrate(self.model, current_q, current_u * dt)

        # Clip the position if it exceeds the joint limits
        new_q = np.clip(new_q, joint_limit_low, joint_limit_up)

        # Clip the velocity if it exceeds the velocity limits
        current_u = np.clip(current_u, -velocity_limit, velocity_limit)

        return new_q, current_u
```

Visualization of the manipulator

Για το visualization γίνεται χρήση της **Franka.visualize()**:

```python
def visualize(self, qs:np.ndarray):
        """
        Visualize the robot

        Args:
        - robot: robot wrapper
        - qs: list of states of the robot at each time step or a single state
        """

        # Visualize the robot
        if len(qs) == self.model.nq:
            self.robot.display(qs)
        else:
            for q in qs:
                self.robot.display(q)
                time.sleep(0.01)
```

Task-Space Controller:

Η συνάρτηση που δημιουργήθηκε για τον ελεγκτή είναι η εξής:

```
(function) def pid_torque_control(
    model: Model,
    data: Data,
    target_T: ndarray,
    current_q: ndarray,
    dt: float,
    Kp: float = 120,
    Ki: float = 0,
    Kd: float = 0.1,
    Kp_theta: float = 10,
    Ki_theta: float = 0,
    Kd_theta: float = 0
) -> tuple[ndarray, ndarray]
```

```
PID torque controller with null space controller

Args:

• model : Pinocchio model
• data : Pinocchio data
• target_T : target pose profile
• current_q : current joint configuration
• dt : time step
• Kp : proportional gain
• Ki : integral gain
• Kd : derivative gain
• Kp_theta : proportional gain for null space controller
• Ki_theta : integral gain for null space controller
• Kd_theta : derivative gain for null space controller

Returns:

• control_t : control torque
• error : error
```

To error signal υπολογίζεται ως εξής:

$$X_e(t) = \begin{bmatrix} \boldsymbol{t}_{wd}(t) - \boldsymbol{t}_{wb}(t) \\ \log(\boldsymbol{R}_{wd}(t)\boldsymbol{R}_{wb}(t)^T) \end{bmatrix}$$

Με $T_{wb}(t)$ το τωρινό pose profile

O ελεγκτής είναι ένας απλός PID ελεγκτής και λειτουργεί ως εξής:

- Υπολογίζουμε το τωρινό πίνακα μετασχηματισμού (pose profile)

```
frame_id = model.getFrameId("panda_ee")

# Compute current transformation matrix
fk_all(model, data, current_q)
current_T = data.oMf[frame_id].copy()

current_rotation = current_T.rotation
current_translation = current_T.translation

target_rotation = target_T.rotation
target_translation = target_T.translation
```

- Υπολογίζουμε ξεχωριστά το σφάλμα κατεύθυνσης και μετατόπισης και δημιουργούμε το σήμα του σφάλματος

```
# Compute error
error_rotation = pin.log(target_rotation @ current_rotation.T)
error_translation = target_translation - current_translation

error = np.concatenate((error_translation, error_rotation))
# print(f"Error: {error}")

if not init:
    prev_error = np.copy(error)
```

- Εφαρμόζουμε το σήμα στον PID ελεγκτή

```
sum_error += (error * dt)
diff_error = (error - prev_error) / dt

error = Kp * error + Kd * diff_error + Ki * sum_error

prev_error = np.copy(error)

if not init:
    sum_error = 0.
    prev_error = None
    init = False
else:
    init = True
```

- Υπολογίζουμε το αντίστοιχο σήμα ροπής

```
# Compute Jacobian
J = pin.computeFrameJacobian(model, data, current_q, frame_id, pin.ReferenceFrame.LOCAL_WORLD_ALIGNED)

# Compute control torque
# Without null space controller
control_t = J.T @ error
```

- Επαναλαμβάνουμε την διαδικασία για τον null space controller με $2^{ον}$ task μια διαμόρφωση στο κέντρο των ορίων του βραχίονα

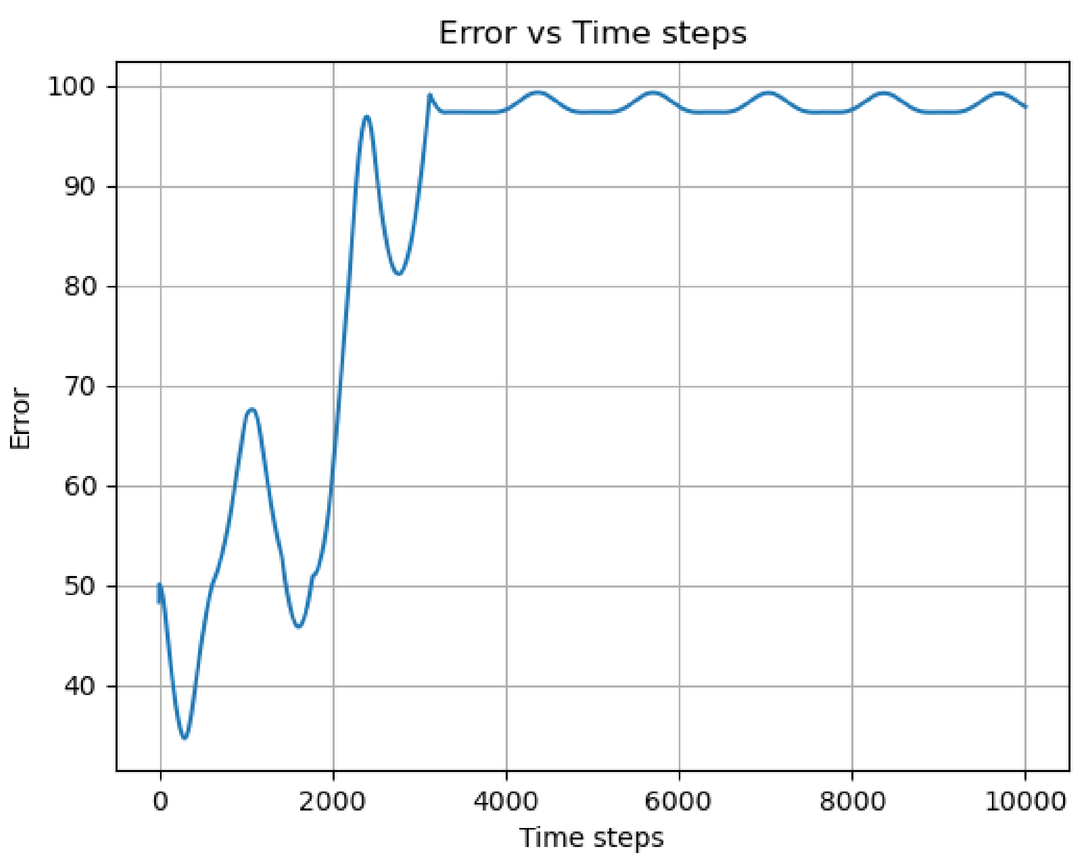
```
# With null space controller
# One implementation of the null space controller
q_target = (model.upperPositionLimit - model.lowerPositionLimit) / 2. + model.lowerPositionLimit
error_null = current_q - q_target

if not init_null:
    prev_null = np.copy(error_null)

sum_null += error_null * dt
diff_null = (error_null - prev_null) / dt
t_reg = Kp_theta * error_null - Ki_theta * sum_null -Kd_theta * diff_null
prev_null = np.copy(error_null)

if not init_null:
    sum_null = 0.
    prev_null = None
    init_null = False
else:
    init_null = True

control_t += (np.eye(model.nv) - J.T @ np.linalg.pinv(J.T)) @ t_reg

return control_t, error
```

Testing the Simulator and Controller:

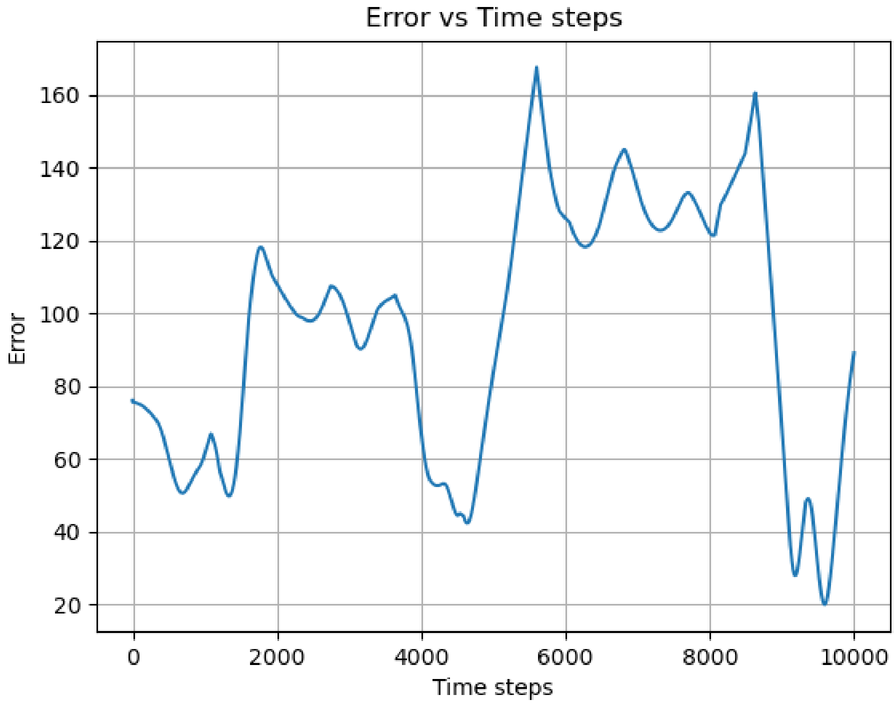
```
----------------------------------
Target pose profile 1:   R =
 0.838022  0.175121   0.51677
0.0244541 -0.958199  0.285055
 0.545088 -0.226246 -0.807274
  p = -0.119831 -0.226101  0.790294

Final Error: [  4.67968036 -12.885694    12.28157814  33.9026774  -89.89856978
   0.43104177]
Final error norm: 97.8269280888503
Initial configuration: [0. 0. 0. 0. 0. 0. 0.]
Final configuration: [-2.8973      1.7628     -2.8973     -3.0718      2.8973     -0.0175
 -2.59242309]
Target configuration: [ 0.6   -0.645 -0.65  -0.15  -0.31   0.18   0.3  ]
----------------------------------
Target pose profile 2:   R =
 0.654158    0.7307 -0.195331
0.0887847  -0.33065 -0.939568
-0.751129  0.597283 -0.281173
  p =  0.137327 -0.660807  0.358383

Final Error: [-21.67107437 -14.10475652 -17.04104289   8.38532783 -46.74852304
 -68.75450406]
Final error norm: 89.11731999033475
Initial configuration: [0. 0. 0. 0. 0. 0. 0.]
Final configuration: [-0.84587949  1.7628      2.8973     -1.66597781 -2.8973      1.26645725
 -2.37568003]
Target configuration: [ 0.5   -0.645 -1.65  -2.15  -2.31   2.18   0.3  ]
----------------------------------
Target pose profile 3:   R =
 0.531512  0.799772  0.279033
 0.391536 0.0601434 -0.918195
-0.751129  0.597283 -0.281173
  p =  0.437323 -0.514075  0.358383

Final Error: [-20.02073061 -13.04263888   6.18301431  13.12712568  15.92626234
   3.48430259]
Final error norm: 32.361665197298144
Initial configuration: [0. 0. 0. 0. 0. 0. 0.]
Final configuration: [-0.52068544  1.7628      1.64993598 -0.76187979  2.8973      1.57353238
  0.20760876]
Target configuration: [ 1.    -0.645 -1.65  -2.15  -2.31   2.18   0.3  ]
----------------------------------
```

```
----------------------------------
Target pose profile 4:   R =
 0.654158    0.7307 -0.195331
0.0887847  -0.33065 -0.939568
-0.751129  0.597283 -0.281173
  p =  0.137327 -0.660807  0.358383

Final Error: [-17.86023146 -33.5165028  -19.45073894  34.23075184 -49.43419676
 -87.19554018]
Final error norm: 114.18941054859543
Initial configuration: [0. 0. 0. 0. 0. 0. 0.]
Final configuration: [-0.34779081  1.7628      2.8973     -2.17518839 -2.8973      0.93560477
 -2.42596476]
Target configuration: [ 0.5   -0.645 -1.65  -2.15  -2.31   2.18   0.3  ]
----------------------------------
```

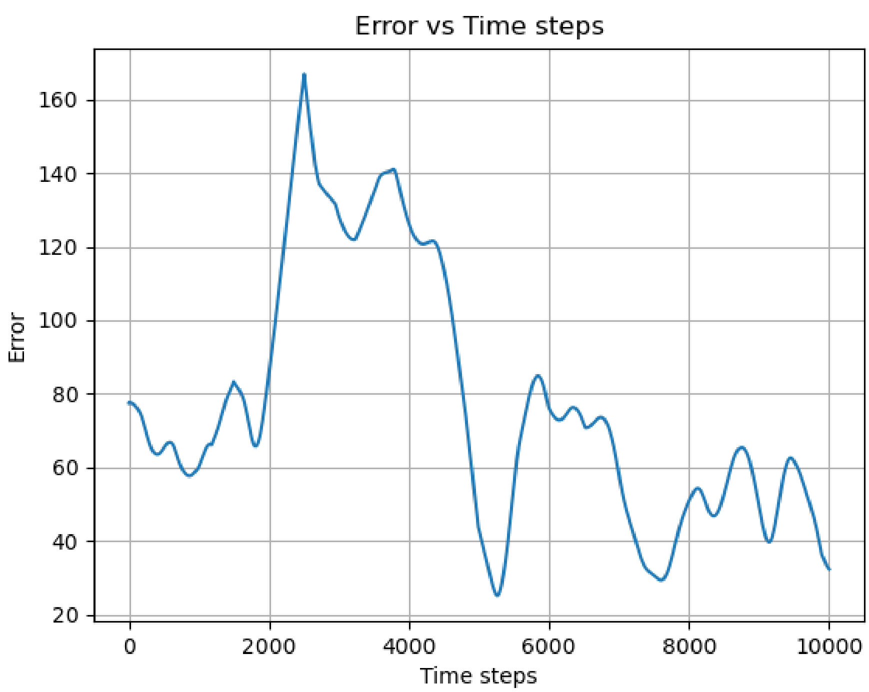
Ο ελεγκτής και για κοντινά και για πιο απομακρυσμένα configurations παρουσιάζει μεγάλα σφάλματα, με τη νόρμα τους να κυμαίνεται συνήθως πάνω από 80. Πιο συγκεκριμένα φαίνεται και στις παρακάτω γραφικές παραστάσεις:



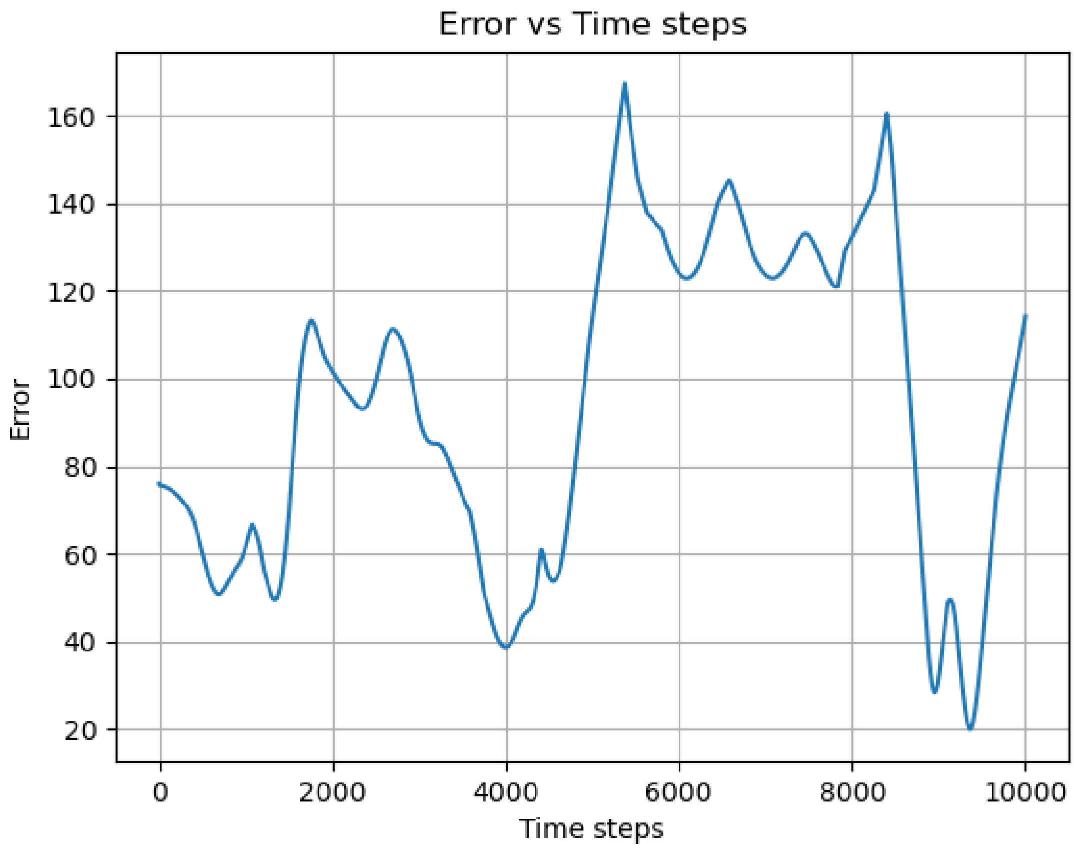
Εικόνα 1:Σφάλμα πρώτης διαμόρφωσης



Εικόνα 3:Σφάλμα δεύτερης διαμόρφωσης



Εικόνα 2:Σφάλμα τρίτης διαμόρφωσης



Εικόνα 4:Σφάλμα τέταρτης διαμόρφωσης