SS-4290 PROJECT

Universiti Brunei Darussalam

FINAL REPORT DRAFT

PROJECT TITLE:

A modularized library for motion planning of robots

SUPERVISOR NAME:

Dr. Ajaz Ahmad Bhat

STUDENT REGISTRATION NUMBER:

18b9062

STUDENT NAME:

Chew Keng Siong @Morgan

DATE OF SUBMISSION:

7th November 2023

Table of Content

Project Objectives

The main objective of this project is to develop and implement a class based on the *The PMP model* from the paper ***Towards a learnt neural body schema for dexterous coordination of action in humanoid and industrial robots*****,** under the section ***Neural acquisition of an internal body model by a humanoid robot.***

This class will encapsulate the key concepts of the model, a PMP for any body chain, and with multiple parallel goals, along with the following:

- Allows control for desired position x, velocity as x_dot, configuration q, joint velocities, q_dot, force, joint torques,
- Allows this for any robot when given their forward Kinematics (fK), Jacobian J, Stiffness K, Admittance is provided or calculated beforehand.

In summary, this class would make the implementation of inverse kinematics for any robot models more easier and modularized.

Tools and Installation

For this project, the main softwares used were *Webots* (R2023a Version), *Miniconda3-latest-Windows-x86_64* and *Python* (3.11.2). Miniconda is used to set up the environment that is used for testing and implementation for the project, under the environment name ***fyp***. Webots is used to carry out tests simulating the robot and the functions created for it in a constructed isolated digital environment. With these functions being coded primarily in *python*. All code created and utilized in this project is being uploaded into a Github repository

(https://github.com/MorganChew/18b9062-Final-Year-Project), this is done for obvious reasons such as version control and keeping project progress.

Setup for the project is relatively simple and requires only these three aforementioned softwares, requiring the user to acquire the three setup installation files for these softwares. Then install them in this specific order. That being

1. *Miniconda*
    a. Then Create the environment by utilizing the "*conda create --name myenv"* command in the Miniconda command prompt (Replacing myenv with the environment name of your choosing).
    b. After that, activate the environment using the "*conda activate myenv*" command (There is a need to call this command at every instance of conducting tests for the project).
2. *Python*
    a. Then after executing the installation file, while inside the environment (refer to step 1.b). Use the "*pip install python*" command in the Miniconda command prompt.
3. *Webots*
    a. Simply activating the setup file is enough.
    b. When attempting to launch Webots, refrain from using the desktop shortcut created by instead launching the application by typing the "*webots*" command into the Miniconda command prompt (While the environment created in step 1.a is activated) .

Installation troubleshooting

If errors are encountered during installation such as the *No QT Plugin Error,* please ensure you are using the correct version of the softwares specified above, while also making sure that they (The softwares) are compatible with your version of windows (or any other operating system).

Guide on Webots

These are the key Important Guides and walkthrough videos studied for this project:

- *Robotics Videos by ThatsEngineering*
- *Webots Crash Course by Solf Illusion*

These video playlists were used to familiarize with the Webots simulations and for the education of robot creation that were required for the project. It is highly recommended that these videos be at least viewed once, if any future contributors have no prior experience with the Webots software.

URDF2Webots

For simplicity and modular nature of the project, finding an existing iCub model was chosen over creating one from scratch, as the latest version (when the project was being conducted) no longer had the sample iCub model available as a Proto. Which is the format best suited to be imported and easily modifiable in Webots. A conversion tool under the search term *Cyberbotics/URDF2Webots* available on GitHub (https://github.com/cyberbotics/urdf2webots),

was tested to convert the more easily acquirable URDF files of robot models for importation and testing for this project.

One such model was downloaded from *robotology* (https://github.com/robotology/icub-models), where the model was extracted and placed in an easily accessible path on the computer.

1. Then using "*pip install --no-cache-dir --upgrade urdf2webots"* in the Miniconda prompt to install the utility tool needed to convert the URDFmodel file to Proto.
2. Then using the *"Python -m urdf2webots.importer --input="model.urdf" --output="Output_File"* command to convert the model file.
3. Then move the converted file into the protos file (of your project environment).
4. when in Webots, import the robot from the current project.

PMP Model

This section will cover the core concepts from the paper that will be important during implementation

What is the Passive Motion Paradigm (PMP) Model

To simplify, *Equilibrium Point Hypothesis (EPH)* suggests that the brain does not fully control the body posture down to every last detail. Instead, There is a balance between all the different forces in our muscles and surroundings. It is like riding a bike, as a natural balance is achieved when riding without paying much attention or thinking about maintaining this balance. This means that even complex actions do not require complicated and high-level planning. Instead, the natural dynamics of the body can find balance when doing specific actions.

Passive Motion Paradigm (PMP) adds to this idea by suggesting that there is an internal picture of the body the brain has to in order to keep balance. Imagine you want to touch a spot. Your brain decides how to move your body parts to reach it. It does this by pretending a tiny push is helping you get there.

Instead of trying to predict every detail of how muscles should move, the brain sort of "learns" from the body's natural behavior. It's like the brain and body are working together to find the best way to move efficiently. This process, where the brain uses the body's natural dynamics to guide movements.It can be a bit tricky to decide whether the brain's control causes the body's equilibrium or if the body's equilibrium shapes the brain's control.But by doing things this way, *we don't need to do lots of hard calculations*. It's like using a shortcut that turns a confusing problem into an easier one.

This idea step by step:

1. There is a specific goal you want your hand to reach
2. Your brain figures out how to move your arm and hand joints to reach that spot. But it doesn't need to overthink every single movement.
3. Instead of directly commanding each movement, the brain spreads out the work among the different joints.
4. The brain then imagines what would happen if a tiny push your arm towards the target.
    a. It imagines a small external force pulling your arm towards the target. This force can be interpreted as your intention to reach the spot.
5. Your brain calculates how much each joint should move to make the arm reach the target point, trying to figure out the all the ways it reach the target

Computational Model

The Important equations taken from the text are:

- $F = K_{ext}(x_T - x)$
- $T = J^T F$
- $\dot{q} = A_{int} T$
- $\dot{x} = J \cdot \dot{q}$
- $x(t) = \int J \dot{q} d\tau$

$x = f(q)$

- q refers to how the current joint positions of the robot are being set to. (Which angles they are currently bent to).
- x refers to the end effector of the arm

The relationship between these two is called a "kinematic transformation." This tells us how the robot's joint angles (**q**) determine where its end is in the real world (**x**). It's like a set of instructions that guides the robot's movements based on the angles of its joints.

$\dot{x} = J(q) \cdot \dot{q}$

- $\dot{\mathbf{x}}$ refers to the speed at which the end effector of the arm are moving
- $\dot{\mathbf{q}}$ refers to the speed at which the robot's joints are moving (changing angles)

When the robot moves, its end position changes.

We can figure out how fast the robot's joints are moving ($\dot{\mathbf{q}}$) using something called the "Jacobian matrix" (**J(q)**).

The equation helps us understand how tiny changes in the robot's joint angles ($\dot{\mathbf{q}}$) lead to small changes in where its end is ($\dot{\mathbf{x}}$). So, if we know how the robot's joints are moving, we can find out how its end is moving.

$F = K_{ext}(x_T - x)$

- F represents the strength of the simulated force which guides the robot's hand.
- K_{ext} is a number that decides how strong this force should be. We call it "virtual stiffness."
- x_T is where we want the hand to be, the target position.
- **x** is where the hand is right now.

As the hand gets closer to the target, the force becomes weaker. It's like the push or pull becomes gentler, so the hand doesn't overshoot the target.

Kext if it's too big, the robot will feel a stronger push or pull. If it's small, the push or pull is weaker.

$$T = J^T F$$

- ***T*** *represents the twist or* "virtual torque,"
- ***JT represents the Matrix - Transpose Jacobian.***
    - *This matrix helps us "translate" the force into a twist in the robot's joints*
- ***F*** is the *force from the equation above*

Now instead of just pushing or pulling the arm, for arm movements such as turning or twisting we use this equation.

$$\dot{q} = A_{int} T$$

- $\dot{q}$ refers to the speed at which the robot's joints are moving (changing angles)
- ***Aint*** represents how the robot's joints behave when they're being guided by the force. It's like a set of rules for how much the joints will listen to the guiding force.
    - helps control how much each joint responds to the force you've created.
        - If a joint is more compliant, it's more flexible and adjusts more easily to the force.
        - If a joint is less compliant, it's stiffer and resists changing as much.
    - Spreads out the effect of the force among the different joints. It decides how much each joint "listens" to the force and adjusts its movement accordingly.

$$x(t) = \int J \dot{q} d\tau$$

- x(t) represents where the robot's hand is at a specific time.
- The integral symbol ∫ means adding up little pieces over time. It's like summing up all the tiny movements of the hand as time passes.
- J is the Jacobian matrix, which we've talked about before.
- $\dot{q}$ represents the joint speeds, how the robot's joints are changing over time.
- dτ represents a tiny bit of time.

In simpler terms, this equation lets you track the robot's hand as it moves over time. Using the integration equation, you accumulate all the small hand movements based on how the joints are altering. This guides you to the end position of the hand, where its motion gradually reduces and it settles in a particular place.

- *As you continue summing up these tiny hand movements over time, you eventually reach a point where significant changes cease. It's similar to when you're driving and start to decelerate as you near your destination. This particular point is referred to as "equilibrium." At this stage, the robot's hand has found its stable position.*

Time-Varying Gain (t)

- increase as the system approaches equilibrium.
- amplifies the influence of other factors in the equation $\dot{q} = A_{int} T \cdot t$
    - This makes the robot's joint movements ($\dot{q}$) more responsive to compliance (A_{int}) and torques (T) when t is higher.

This benefit of this is that there is a faster adjustments in the robot's arm configuration and achieving Equilibrium sooner.

Development process

As summarized in the conclusion of the interim report, the implementation and testing of the PMP class had not been conducted but important concepts such as Inverse Kinematics and libraries such as IKPY and Roboticstoolbox have been researched and tested during the first semester of this project.

During the start of the second semester, the supervisor discussed incorporating variables within the class itself and introducing two functions: 'fkin' for forward kinematics and 'jacobian' for Jacobian matrix calculation. These functions will take 'q_current' as an input to represent the robot's current configuration. Notably,while located within the 'pmp' class, these functions won't form part of the core algorithm, ensuring a modular structure that separates kinematics from the algorithm itself.

```
class PMP:
    x, x_dot, q, q_dot, F, tau = []
    FKin, Jac, K_stiff, A_admittance = []
    TBG = []
    def stateUpdate(self):
        #update the above variables from robot
        pass
    def setGoal(self):
        # set any of the variables (say pose) as a goal for the system
        pass
    def setRobotBody(self):
        #specify robot specific details of FKin, Job, K_stiff, A_admittance
        pass
    def step(self):
        #does 1step of PMP simulation on the body chain specified
        Pass
```

The initial outline of the PMP class

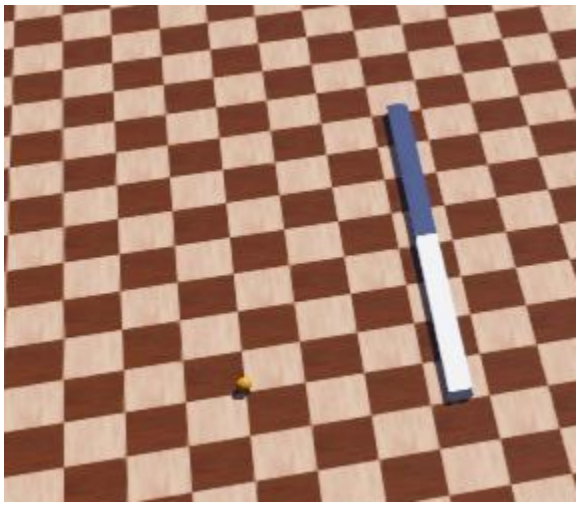
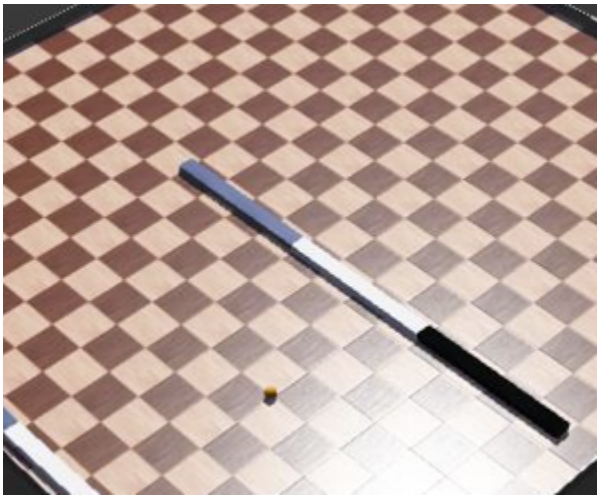
At the point in time of the project, all the code was written on one file. Due to the modularity requirement of the project. A proper class PMP file was defined, encapsulating the logic of the Position-based Motion Planning controller. The following is a more detailed outline of the PMP class and what methods it would contain:

- **Class Initialization:** The __init__ method initializes the class attributes. It takes parameters such as my_bot (robot instance), x_target, q_current, robotics_Library, K_ext, A_int, and TBG for external control parameters.
- **Stiffness and Admittance Settings:** set_stiffness and set_admittance methods allow setting the stiffness and admittance matrices for control.
- **Kinematics Calculation:** The calculate_fkine method calculates forward kinematics based on the provided robotics library (either 'rtb' or 'ikpy'). If 'ikpy' is chosen, it loads a URDF file and calculates the transformation matrix.

- **Vector Calculations:** Methods like calculate_error_vector, calculate_setpoint_x, calculate_current_x, and jacobian handle vector calculations, such as error computation, setpoint calculation, current position calculation, and Jacobian computation.
- **Torque and Velocity Calculations:** calculate_Torque, calculate_q_dot, and calculate_Force methods calculate torque, joint velocities, and control forces using provided matrices and vectors.
- **Time-Based Gain (TBG) Calculation**: calculate_tbg computes a time-based gain factor based on the distance between the current and target positions.
    - The "**calculate_tbg**" function uses the "x_error" vector, which shows how much the robot's end effector position differs from the desired target. The function calculates the Time-Based Gain (TBG) that affects how the robot moves. It measures the distance between the current end effector position and the target. The TBG value is made by adding 1 to this distance, creating a factor that grows when the robot is far from the target. This makes the robot move more carefully as it gets closer to the target, ensuring smooth and controlled motion.
- **Motion Planning Algorithm:** The step method implements the main position-based motion planning algorithm, updating joint positions and calculating errors iteratively until convergence.
- **PMP Kinematics Function:** pmp_kinematics method takes a target name and calculates the joint trajectory using the step method.

- **Motor Control:** set_motors method sets motor velocities and positions based on the calculated joint angles.
- **Execution of PMP:** The execute_pmp_kinematics method encapsulates the execution of the PMP kinematics. It calculates the trajectory, final joint angles, and sets motor positions accordingly.
- **Main Execution**: In the main part of the script, the Supervisor is initialized, and stiffness and admittance matrices are defined. An instance of the PMP class is created, and the PMP controller is executed for a specified target ('Ball').

This was the outline of the code before it was later broken down into separate components. After this several worlds were made for Webots.

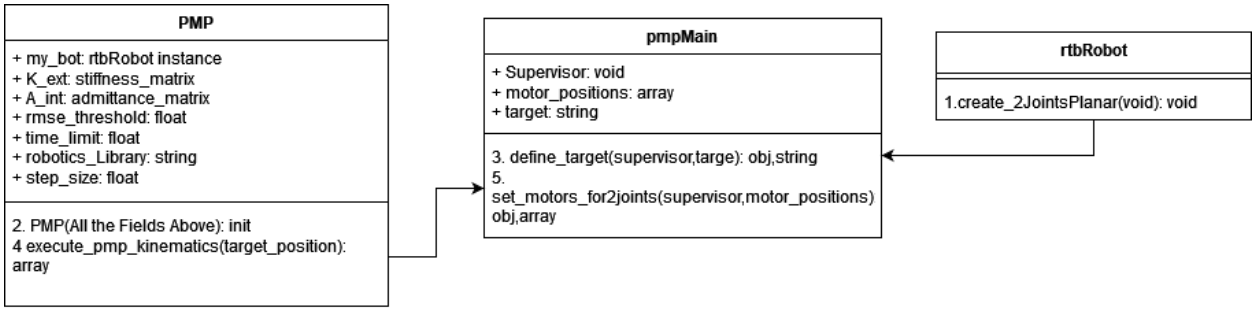



Pictured above are the two worlds made to test the PMP class, that test being seeing if the class could return the correct calculations to set the joints to.

Implementation of the Class

This section will breakdown the components created in this project, mainly using an example of how PMP would be implemented to a robotics system and used to reach a ball target. In this example, the robot being used for this section is a simple 2 joints planar. The three main files are:

- pmpClass (stores the PMP class)
- pmpMain (The Main driver file)
- rtbRobot (Creates a robot using RTB library)



class diagram of the three main files (methods are numbered in order of execution)

PMP Class

This class is a Python implementation of the aforementioned ***Position-Based Motion Planning*** (PMP), used for controlling and planning the motion of a robot or robotics system. The following is an explanation and breakdown of methods found within the class file(It is named **pmpClass.py**):

1. Constructor method:

- Initializes the PMP class with various parameters, these being:
    - the robot instance (my_bot),
    - target position (x_target),
    - current joint configuration (q_current),
    - external stiffness matrix (K_ext),
    - internal admittance matrix (A_int),
    - root mean square error threshold (rmse_threshold),
    - time limit for execution (time_limit),
    - step size for motion updates (step_size).

2. calculate_fkine method:

Calculates the forward kinematics transformation matrix of the robot given the current joint configuration (q_current), using the provided `my_bot` instance.

3.calculate_error_vector method:
Computes the error vector between a setpoint position (`setpoint_X`) and the current position (`current_X`). This vector represents the deviation from the desired target.

4.calculate_setpoint_x method:
Simply returns the provided `x_target` as the setpoint position.

5. calculate_current_x method:
Uses the provided joint configuration (`q_current`) to calculate the current position of the robot's end effector using forward kinematics.

6. calculate_jacobian method:
Computes the Jacobian matrix for the robot given the current joint configuration. The Jacobian matrix describes the relationship between joint velocities and end-effector linear and angular velocities.

7. calculate_Torque method:
Calculates the joint torques required to achieve a given force (`Force`) at the robot's end effector. It uses the Jacobian matrix to perform this computation.

8. calculate_q_dot method:
Determines the joint velocity (`q_dot`) needed to generate the desired torque using the internal admittance matrix (`A_int`) and the computed torque.

9. calculate_Force method:
Computes the force vector to be applied at the end effector based on the external stiffness matrix (`K_ext`) and the error vector (`x_error`).

10.calculate_Rmse method:
Calculates the root mean square error (RMSE) of the error vector (`x_error`). RMSE is a measure of the average deviation between the setpoint and current positions.

11. run_step method:
Executes a motion planning and control loop to move the robot's end effector from the current position to the target position while considering external stiffness and internal admittance. The method iterates until the RMSE is below the specified threshold or the time limit is reached.

```python
def step(self,x,x_dot,force,q,q_dot,torque,setpoint_x, K_ext, A_int):
        x_error = self.calculate_error_vector(setpoint_x, x)
        self.force = self.calculate_Force(self.K_ext, x_error)
        self.jacobian = self.calculate_jacobian(q)
        self.torque = self.calculate_Torque(self.jacobian, self.force)
        self.q_dot = self.calculate_q_dot(self.A_int, self.torque)
        self.q_dot = [float(val) for val in self.q_dot]
        self.q = [q + (q_dot_val * self.step_size) for q, q_dot_val in zip(self.q, self.q_dot)]
        self.x = self.calculate_fkine(q)
        return self.x,self.x_dot,self.force,self.q,self.q_dot,self.torque
```

12**. step method:**
Performs a single step in the motion planning and control loop. It updates the robot's state, calculates the error vector, force, torque, joint velocities, and joint positions.

13. pmp_kinematics method:
This method is responsible for computing the trajectory of joint angles needed to move the robot's end effector from the current position to the target position while considering external stiffness and internal admittance. It calls the `run_step` method to execute the motion planning.

14. calculate_final_q_current method:
Retrieves the final joint configuration from a computed trajectory, which can be useful to get the robot's state after completing the motion.

15. execute_pmp_kinematics method:
This method orchestrates the execution of the PMP-based motion planning, including motion trajectory generation, error analysis, and joint configuration updates. It returns the computed trajectory, execution time, RMSE, and error information.

In summary, the `PMP` class encapsulates a range of methods and attributes essential for motion planning, execution, and error handling in a robotic system, offering a structured approach to controlling a robot's motion based on external stiffness and internal admittance.

Simulating the Internal Model

By Importing the Robotics toolbox library, the following code defines and creates a 2-joint planar robot using the Robotics Toolbox (RTB) in Python. The following is an explanation and breakdown of methods found within the class file (It is named **rtbRobot.py**):

1. It defines a series of elementary transformation sequences (ETS) for the robot's kinematic chain. Each ETS element represents a transformation in the robot's structure or motion. In this case, the ETS sequences define the following transformations:
    - t0: A translation along the x-axis by -0.5 units.
    - r1: A rotation about the z-axis (no specified angle, so it's an identity rotation).
    - t1: A translation along the x-axis by 1 unit.
    - r2: Another rotation about the z-axis (again, no specified angle).
    - t2: Another translation along the x-axis by 1 unit.
2. These ETS sequences are used to define the kinematic structure of the robot by creating an instance of the rtb.Robot class. The rtb.Robot class takes an ETS sequence as its argument, which represents the robot's kinematic structure. The my_bot variable is set to this newly created robot.
3. Finally, the function returns the my_bot object, which represents the 2-joint planar robot with the specified kinematic structure defined by the ETS sequences.

In summary, the code creates and returns a 2-joint planar robot with specific translation and rotation transformations along the x-axis and z-axis using the RTB library. This robot's kinematic structure is defined by the elementary transformation sequences.

Using the PMP class

The following is an example of how the PMP class would be imported and used, it is a driver code file and is using both the PMP class and rtbRobot.py.

The following is a breakdown of the sequential sequence of code executions for the Python file (Named as **PmpMain_2JointsPlanar2.py**) that implements this class and uses to allow a 2 joint planar robot to reach a ball target:

1. **Importing Necessary Modules:**

- Imported PMP from pmpClass.py to utilize the PM class.
- Imported create_2JointsPlanar from rtbRobot.py to create a 2-joint planar robot using the robotics toolbox library.
- Imported numpy as np for creating identity matrices.

```
my_bot = create_2JointsPlanar()
```

2. Creating a 2-Joint Planar Robot:

The method *create_2JointsPlanar()* is called to create a 2-joint planar robot using Robotic Toolbox (RTB). that is defined using a sequence of transformations and rotations.

```
#WEBOTS
# Initialize the Supervisor
fyp_supervisor = Supervisor()
```

3. Initializing the Webots Supervisor:

The Webots Supervisor is initialized as *fyp_supervisor*, which allows the script to control and interact with the simulated robot in the Webots simulation.

```
#print("i                                    get",target)
target = supervisor.getFromDef(target) # Get the target object from the Supervisor
target_position = target.getPosition()
```

4. Defining a Target Position:

The ***define_target()*** function is used to define the target position. It does this by taking the supervisor and an optional target object name as arguments. The target position is then obtained from the supervisor.

5. Setting Motor Positions:

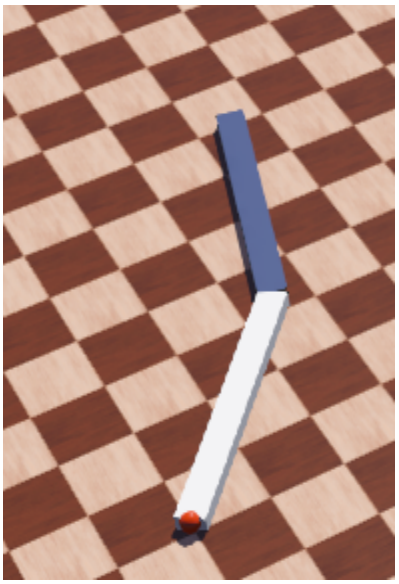
The ***set_motors_for2joints()*** function is called to set the motor positions of the robot based on the provided motor_positions list. It assumes that the robot has motors with specific names ("joint1" and "joint2").

7. Defining Stiffness and Admittance Matrices:

- ***Stiffness_matrix*** and ***admittance_matrix***, are defined as 3x3 and 2x2 identity matrices,
- respectively. These matrices are used to control the robot's stiffness and admittance.
- They both can be modified and adjusted accordingly to the robot being used.

```
67  # Create the PMP controller instance using Robotic Toolbox
68  pmp_controller = PMP(
69      my_bot, K_ext=stiffness_matrix, A_int=admittance_matrix, robotics_Library='rtb',
70      rmse_threshold=0.005,time_limit=1000, step_size=1/5)
```

8. Creating a PMP Controller Instance:



An instance of the PMP class is created as ***pmp_controller***,with the necessary parameters provided.

9. Executing PMP Kinematics:
The ***execute_pmp_kinematics()*** method of the pmp_controller instance is called to compute the joint trajectory of the robot needed to move the end effector to the target position, when provided the target position. This trajectory is stored as q_traj, and other variables such as time, rmse (root mean square error), and x_error are also calculated and stored.
10. Setting Motor Positions: The computed motor positions from q_traj are extracted and set as the new motor positions for the robot, the positions used is the value of the last element in the q_traj list.
The Image on the left showcases a successful Implementation of the PMP class for a 2 jointed planar robot.

Implementing the PMP class to a different robot

Changes needed to be made

Now after showcasing the successful implementation of the PMP class for the 2 jointed planar robot, this section will cover how this class can be used for a 3 jointed version of the planar robot.

As the PMP class is intended to be modular, no further changes are required to be made in the class file when implementing this robot, instead the only changes are made in the main driver controller, these changes are:

```
my_bot = create_3JointsPlanar()
```

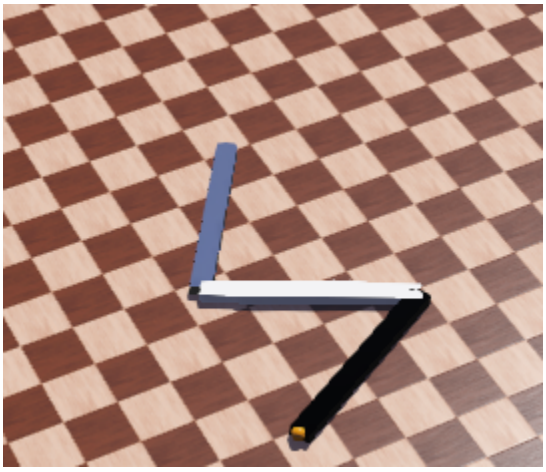
Instead of calling The function *create_2JointsPlanar(), for this robot the function create_3JointsPlanar() is called. This function is a better representation of the* sequence of transformations and rotations of the 3 jointed Planar robot.

```
def set_motors_for3joints(robot,motor_positions):
    motors = []
    motor_names = ["joint1", "joint2", "joint3"]
```

While previously The ***set_motors_for2joints()*** function is called to set the motor positions of the robot, here the function is renamed to ***set_motors_for3joints()*** and now included “joint3” in the motor_names list

```
#PMP
# Define stiffness and admittance matrice
stiffness_matrix = np.identity(3)
admittance_matrix = np.identity(3)
```

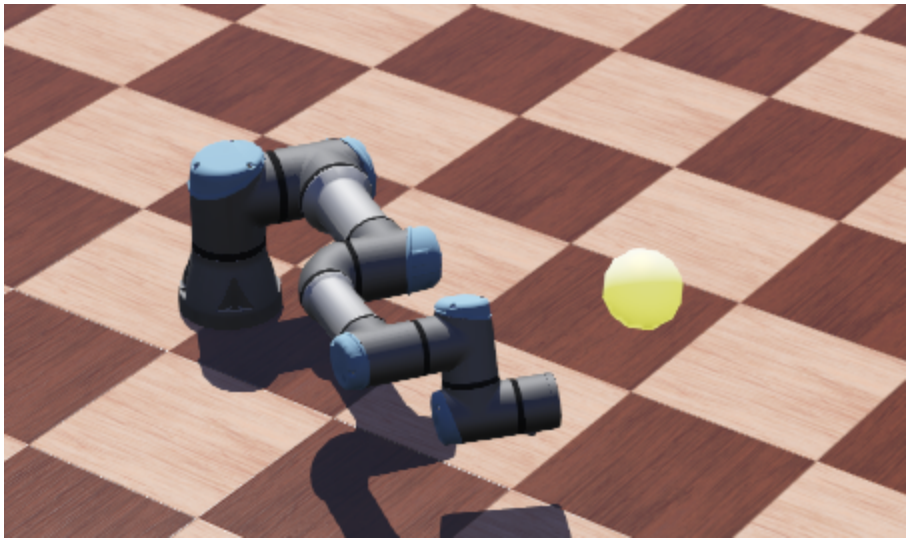
The admittance_matrix must now also be a 3x3 identity matrix or any 3x3 matrix (but in this example the former is used), not doing so will and using the previous 2x2 identity matrix will result in error messages.



That should be all the necessary changes needed to be made for the 3 jointed planar robot to utilize the PMP class. Pictured above is the result of the changes made to the main driver code and as can be deduced is a successful implementation of the PMP class.

Testing and Troubles faced

The obvious way of testing is to write test cases, as test-driven development allows programmers to write functional code and reduce time needed to diagnose errors. A test file written for the class file can be found in the same folder of the class under the name "**testpmp.py**". This file would set up an instance of the class and test all the methods of the class, to ensure not only that the methods are returning the correct values but also returning relevant values that are relevant to them.



When attempting to utilize the class for the URE3 (imported to the simulation using URDF2WEBOTS), the robot was unable to reach the target and instead moved away from it.

```
#PMP
# Define stiffness and admittance matrices
stiffness_matrix = [[0.05,0,0],[0,0.02,0],[0,0,0.01]]
admittance_matrix = np.identity(6)*0.1
```

During a meeting with the supervisor regarding this matter, it was proposed that focus be given on the value being passed for stiffness and admittance, this did produce more reasonable visual results. But further investigation by future contributors would be required. This issue with the URE3 robot also extends to all other robots with three degrees of freedom.

Further implementations and Improvements

TBG

Several implementations were unable to be made during the timeline of the project, one being the implementation of the time base generator (TBG), and to summarize it yet again for

this section. This means that TBG would produce a force that would be exerted on the end effector when it is very close to approaching the target. But since this was not implemented, the end effector simply slows down and slowly reaches the target, this is not ideal of course and can be time consuming.

```
#tbg is a union of time, so s
def calculate_tbg(self,x_error):
    # x_error is used as it represents the difference between the desired target position (setpoint) and the current end effector position of the robot. I
    distance_to_target = np.linalg.norm(x_error)
    tbg = 1.0 + distance_to_target #The TBG is calculated using the following formula:
    #distance_to_target represents the Euclidean distance between the current end effector position and the target position
    #The calculated tbg value adds 1 to the distance, effectively creating a scaling factor that increases with the distance to the target.
    # The idea is that the farther the end effector is from the target, the more the TBG will increase, influencing the joint angle update rate.
    return tbg
#the error keeps decreaing , force is decresing,  tbg says the closer to the target , the tbg would increase overtime as errir increases
#check the paper again
```

Picture above is the code written as an attempt to implement TBG, but it is incomplete and does not fully incorporate the concept well. It is not found in the final version of the PMP class file but a simple recreation should be possible using the code above.

```
    #tbg = self.calculate_tbg(x_error)
    q_dot = self.calculate_q_dot(self.A_int, Torque) #* tbg
    q_current = q_current + (q_dot * 1/5)
```

To use the TBG in the existing code, recreate the TBG method using the images above and to utilize the method. Go to the step method and in the line of code shown above, uncomment the “tbg” and further development can proceed.

IKPY

Another area of improvement is the use of IKPY library, as the project initially started development using this library but due to technical circumstances, development switched to using the Robotics toolbox library. Therefore, all calculations for the jacobian and kinematics in the project are done using the RTB library, this was not the intended goal of the project. As it originally was meant to allow both libraries to be able to provide the calculations for this project, but due to the approaching deadline of the project, focus was solely placed on getting the project

to progress using RTB. Hence this is a major area of improvement, as the modularity of the project should allow the use of IKPY for kinematic calculations, it would also allow more testing and refinement for the PMP class.

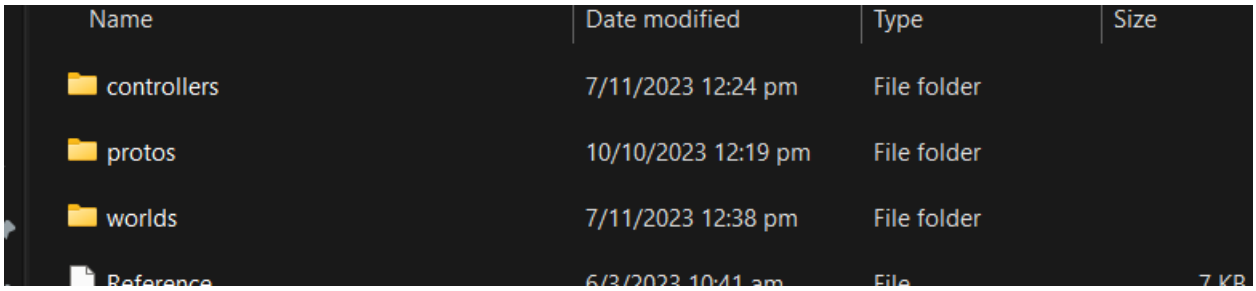
Implementing to 3D Robots

As Mentioned before, several issues were present when attempting to use this class for calculating positions for the joints of 3D robots, the key issue being the results produced for the angles being incorrect and resulting in the joints being unable to reach and at worst moving away from the target. This is obviously not unacceptable, as the project intended for the PMP class to be modularized and so should function with 3D robots without changes to the class file itself. But due to the deadline of the project, there was no time left to fully resolve this issue. So it is because of this, future contributors need to take note of this flaw and some of the proposed causes in the previous section mentioned in the project report.

How to contribute to the project

Future contributors for the project that would like to pick up the progress can find the code on the github repository mentioned under tools and installation. After installing Webots and all other listed necessary tools and libraries for this project, when in the Webots simulation, create a

new robot controller and this should create a new folder.



Navigate to this folder (**controllers**) and place the downloaded controller folders from the github Webots would create a new controller file when the folder was made, simply replace the code of this new file with the main driver code from the github. This main driver code is the one with the prefix of pmpMain and now delete the file that is redundant. That is how the controller can be easily imported in an existing or new project, world files for the project are also provided. Now simply build up the existing PMP class and hopefully the listed improvements can be made. The paper ***Towards a learnt neural body schema for dexterous coordination of action in humanoid and industrial robots*** will need to be requested from the Supervisor, Dr. Ajaz Ahmad Bhat. It is not unfeasible to continue without it but this would be ill-advised, as the project is built on the core concepts of this paper.

Conclusion

The PMP class is overall in need of major refactoring and even more testing is required, it is still experiencing issues with running 3D robots, hence it will require further investigation by future contributors. Hopefully these can be resolved and the final goal of pushing the class to an existing code base of a robotics library can be attained.

References

Bhat, A. A., Akkaladevi, S. C., Mohan, V., Eitzinger, C., & Morasso, P. (2016). Towards a learnt neural body schema for dexterous coordination of action in humanoid and Industrial Robots. *Autonomous Robots, 41*(4), 945-966. doi:10.1007/s10514-016-9563-3