
**SS-4290 PROJECT**

**Universiti Brunei Darussalam**

INTERIM REPORT

**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:**

24th April 2023

**Table of Contents**

**Goals of the Project**

These are the main overall objectives of the project,

1. Create an iCub robot inside the Webots Environment. Then create functions for it that allow it to extend and move the joints to reach and touch an object.
2. Create a PMP class/library that offers, a PMP for any body chain, and with multiple parallel goals [Inverse kinematics]
    3. Allows control for desired pose x, velocity x_dot, configuration q, joint velocities, q_dot, force, joint torques,
    4. Allows this for any robot given forward Kinematics (fK), Jacobian J, Stiffness K, Admittance A)
5. Create a push request and potentially add this class to the Ikpy library python

In Summary**,** the project aims to contribute and add to the existing IKPY library. Making the implementation of inverse kinematics for any robot models more easier and modularized.

**Explanation of the iCub robot**

The robot chosen for this project is the *iCub* robot developed at IIT as part of the EU project RobotCub. The model of iCub specifically used in this project at the moment, is the *standing iCub* created for the Webots sample world *icub_stand.wbt.* As stated in the Webots User Guide, the robot has 53 motors that move the head, arms & hands, waist, and legs. These motors each represent *joints*, in terms of Webots. A joint allows two bodies to connect and move in relation to one another, with properties such as limits of motion, torque and velocity being able to be configured. The benefit of using an iCub robot model is the amount of joints it possesses. As

the project requires modularity and should be able to be implemented into any other model. The iCub model provides an ideal opportunity to test the algorithm and ensure that it is functional on any of the joints the model contains (such as arms, legs and torso).

The robot was imported to Webots for the project, using the URDF file provided by the supervisor. The URDF file, While attempts to acquire different models of the iCub were tried, as the model being currently used is rudimentary and lacks complex and even basic features of an average iCub model (Such as finger joints for the hands), but due to both possible error and lack of understanding of the conversion method used to convert Proto to URDF (This will be covered later in the report), the more simple *iCub_stand* model is being used.

**Joint limits of iCub**

| Left Arm | | | | | |
|---|---|---|---|---|---|
| Row | Name | Min_Position | Max_Position | Max_Torque | Max_Velocity |
| 8 | left_arm_1 | -0.8727 | 4.0143 | 34 | 10 |
| 9 | left_arm_2 | -2.618 | 1.5708 | 40 | 10 |
| 10 | left_arm_3 | -1.5708 | 1.5708 | 34 | 10 |
| 11 | left_elbow | 0 | 2.4435 | 20 | 10 |
| 12 | left_forearm | -0.5236 | 0.5236 | 0.45 | 10 |
| 13 | left_wrist_1 | -1.5708 | 1.5708 | 0.65 | 10 |
| 14 | left_wrist_2 | -1.5708 | 1.5708 | 0.65 | 10 |

*Table 1*

As like our very own physical human body parts, the joints of the iCub robot have limits to how far, how much and how fast it can be articulated. Table 1 has the recordings of the four property limits of the joints under the left arm of the iCub robot. That being the *minimum and maximum positions* able to be set for the joints, this property (*Position)* is currently most

significant in the project, as the aim is to figure out where to appropriately set the position of the individual joints to reach a target. Setting values lower than the minimum position or higher than the maximum position will result in a warning message from Webots informing the user of this discrepancy, with the software automatically assigned a new more appropriate value to the joint position. The *maximum torque*. Which refers to the highest amount of physical force the joint can handle. This property is currently not relevant at this stage of the project. The *maximum velocity*, which refers to the highest possible rate of movement it can undertake in a certain direction, the Velocity is currently being set to 1 by default. As any value higher than it will result in the robot falling and any value lower than it will be too slow and inefficient when testing.

Currently the four aforementioned properties of 31 joints are recorded in a table under the name "*Table for Range_of_Icub_Joint*". With the body parts of the iCub robot that have been recorded being the Neck, Torso, both right and left arms and both left and right legs.

**Explanation of Kinematic Terms**

**Inverse Kinematics** is the most important concept being implemented in the Project. To explain briefly, it refers to the computation of the joint angles that are needed to move a ***limb*** (in the case of this project) to reach the desired end-effector position (***Target***). Using the results of these calculations to set new positions and rotate the joints to angles that bring the limb to the target. This method is commonly done using a jacobian matrix.

**Forward Kinematics,** in simple terms, refers to computing the end point of the limb based on the positions of its joints; it involves using transformation matrices to calculate how each joint moves the end point and combining them together to get the final position and orientation of the end point.

The **Jacobian matrix** is a mathematical tool that describes the relationship between the motion of a robot end-effector and the movement of individual joints. By utilizing the Jacobian matrix, it becomes possible to calculate the necessary motion of each joint required to achieve a desired velocity and position for the end-effector. Once the required joint velocities are determined through the Jacobian matrix, they can then be integrated over time to obtain the joint angles necessary for the robot to reach the desired end-effector position.

**Stiffness** is related to the condition number of the Jacobian matrix. This being a measure of how sensitive the output of a function is to small changes made into the input.

A high condition number indicates that the system is stiff, this means that any small errors in the velocity of the desired end-effector (target) can result in large and prominent errors in the joint velocities. This can cause the system to become unstable or unable to accurately achieve the desired trajectory.

Meanwhile, a low condition number indicates that the system is compliant, meaning that it can tolerate and handle small errors in the velocity of the desired end-effector (target) without causing large errors in the joint velocities. This can make the system more stable and allow for more flexibility in the trajectory. An ideal state to maintain in a system.

**Admittance** is the method of controlling a robot by specifying the desired behavior of the force and velocity of the end-effector (target). Then by adjusting the admittance of the system until achieving the desired level of stiffness or compliance required or specified for development and testing.

**The Equilibrium Point Hypothesis** (EPH) proposes that the posture of the body is not directly controlled by the brain but is instead a biomechanical consequence of equilibrium among these forces. This means that complex actions can be achieved without a complex optimization process, by allowing the intrinsic dynamics of the neuromuscular system to seek its equilibrium state when triggered by intended goals.

The **Passive Motion Paradigm** (PMP) builds upon the Equilibrium Point Hypothesis (EPH) to explain how the brain controls movement. According to PMP, the brain uses an internal simulation process on the body schema to determine how to move the joints in order to achieve a specific goal.

Overall, PMP suggests that the brain controls movement by simulating the distribution of forces across the joints, rather than by directly controlling the movement of each joint. This approach is more efficient and requires less computational power than other methods, such as **inverse kinematics.**

**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).

**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.

**Issues faced for using URDF2Webots**

Currently not using the converted models (**iCubGenova09**), as several unresolved issues are encountered with models imported from the method. Primarily issues regarding the texture and packages that are required by the models, as models imported are not able to be observed in the Webots environment and in some instances, the model falling out of the environment. Notable attempts to resolve this issue included. Using the "*Cd*" command to locate the file path of the URDF model, this resolved the inability to convert the model and allowed importation into Webots. But as mentioned before, errors regarding the appearance and functionality of the robot are still apparent.

The current theory to resolve this issue, is that Appearance and Texture packages pathing to the texture files are correct but may be undetected for a currently unknown reason. Due to time constraints, this tool is currently not being used at the moment for this project.

**Robotics Toolbox**

Robotics Toolbox is a MATLAB based toolbox designed for robotics applications. It contains a set of tools and functions for simulation, analysis, and control of robotic systems. It can be very useful for solving inverse kinematics problems, which involve determining the joint angles required to reach a specific position and orientation.

Here are some ways that Robotic Toolbox can be useful with inverse kinematics:

- The Robotics Toolbox offers a diverse array of algorithms and functions to effectively solve inverse kinematics problems. These functions facilitate fast and effortless determination of the necessary joint angles for a given end-effector orientation and position.
- The Robotics Toolbox provides the ability to define and simulate intricate robot models, incorporating multiple degrees of freedom. This feature can prove advantageous in verifying and testing inverse kinematics solutions on various robot configurations and environments.
- The Robotics Toolbox contains functions utilized for generating trajectories and motion planning, allowing for the generation of seamless and efficient paths for the

robot to pursue. This feature is advantageous in optimizing the movement of the robot and mitigating inaccuracies in the inverse kinematics solution.

- Additionally, Robotic Toolbox also provides visualization tools that can help you to visualize and debug your robot models and inverse kinematics solutions. This is obviously useful and beneficial for understanding and improving the performance of your robotic system.

In summary, Robotics Toolbox can be an invaluable asset for resolving inverse kinematics problems and optimizing robotic systems. It provides the user with an extensive selection of functions and tools that can be used for simulating, analyzing, and controlling robotics applications, which can assist users in efficiently resolving intricate problems and enhancing the overall performance of their system.

**Testing and Implementation**

This section covers the concepts and libraries that were tested and implemented during the first semester of the research project. Hence represents the bulk of the work conducted that was not strictly research.

**Move Function**

This function uses the body parts of the iCub model as arguments and moves the body parts (Such as the Left arm, torso or right leg) that were passed to the function. With the position and velocity of the individual joints of the body parts called by the function being set according

to the config specified in the function arguments. The current version of this function can be found under the controller file named "*icub_jointsV2.py*".

*in the section #Get Joints* of the code

- Here the list of joints is initialized for each body part (referred to as "limbs" in the project) of the robot.

In the Section *#Defines the MoveLimb function*

The Function takes in the limbs of the robot as arguments, being able to take up to six possible arguments.

- Then it would move the joints of the limbs that were passed to the function.
- The limb argument passed to the function would enter a for loop that would iterate over each joint in the limb (here in the form of a list)
- Then the position and velocity of each joint would be set based on values specified in ***values*** dictionary
- The ***position*** value determines the desired position of the joint,
- while the ***velocity*** value determines the desired velocity at which the joint should move to the desired position.

In the Section *#movesets*

Here you would select a move set for a limb by creating lists of tuples. Each tuple contains a joint from the corresponding limb list and a dictionary with the joint's desired position and velocity.

- These tuples are then passed as arguments to the *move_Limb* function which moves the robot's limbs according to the specified positions and velocities.

This is the current capabilities of this function, with the  position and velocity for the joints being able to be set but only to predefined movesets (Currently only tested with left and right arm). The justification for writing the code framework for the functions for the iCub model this way, was because it was suggested by the supervisor to convert the limbs to a dictionary structure. Reasons being,

- Allowing the ability to Call the move function with the dictionary of limbs as arguments can then be implemented.
- Organizing the limbs into this dictionary structure would make it easier to access and manipulate each limb individually.

However, it is important to make sure that the dictionary keys correspond to the actual limb names and that the values contain the necessary information such as position and velocity.

**Design of the PMP Class**

```
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 outline of the PMP class*

The following is an implementation of the PMP model shown in the research paper "*Towards a learnt neural body schema for dexterous coordination of action in humanoid and Industrial Robots*". The paper presents these following concepts and calculations:

***F = K ext (x T – x)*** *(Generate a target-centered, virtual force field in the extrinsic space)*

In robotic manipulation, the goal is to move the robot's end-effector to a specific target location. The virtual stiffness of the attractive field, denoted as **K ext**, is a property that determines how difficult or easy it is for the robot to move towards the target. The intensity of the

force field F generated by K ext decreases as the end-effector gets closer to the target, making it easier for the robot to reach the target without overshooting or undershooting.

In the simplest case, K ext is proportional to the identity matrix, which means the force field generated is the same in all directions and is aligned with the robot's current position, resulting in a straight path towards the target. However, in more complex scenarios where the robot needs to avoid obstacles or manipulate tools, K ext needs to be adjusted to generate more complex force fields that guide the robot's end-effector along a curved path towards the target. This can be done either by manually adjusting K ext or by using a learning algorithm to find the appropriate values.

***T = J T F*** *(Map the force field from the extrinsic space into a virtual torque field in the intrinsic space)*

The variable **T** denotes the torque field, a kind of force that can induce rotational or circular motion. Matrix **J** helps to transform the force field from the extrinsic space (real world) into the intrinsic space (virtual world) of the torque field. The transposed Jacobian matrix (**JT**) optimizes the performance of matrix J for our purposes. A body schema model, which represents the connection and movement of various body parts, can be used to calculate this matrix. In conclusion, this equation facilitates the conversion of force fields from the real world into torque fields, which can be utilized for controlling a virtual robot. The transposed Jacobian matrix, calculated through a body schema model, plays a vital role in this process.

$\dot{q} = A\ int\ T$ *(Relax the arm configuration to the applied field)*

The formula $\dot{q} = AintT$ enables us to control the arm's movement by adjusting a matrix called the virtual admittance matrix (**Aint**), which represents the arm's compliance or flexibility at each joint. Through the adjustment of the Aint matrix, we can control the amount of torque applied to each joint and the arm's movement response to external forces. This allows us to relax the arm and promote free movement without excessive force or resistance.

$\dot{x} = J\ \dot{q}$ *(Map the arm movement into the extrinsic workspace)*

This formula would calculate the velocity of the arm movement in the external world based on the velocity of the arm joints (**q**).

The "**J**" in the formula refers to the Jacobian matrix, which describes how each joint contributes to the movement of the arm in the external world. By multiplying this matrix by the velocity of the joints (**q**), we can calculate the velocity of the arm movement in the external world (**x**).

$x\ (t) = \int t\ J\ \dot{q} d\tau$*(Integrate over time until equilibrium)*

This formula integrates the velocity of the arm movement over time until it reaches a stable position and orientation in the external world.

**Pseudo code based on the concepts discussed previously**

```
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 1 step of PMP simulation on the body chain specified based on concepts in this passage
        # Step 1: Generate a target-centered, virtual force field in the extrinsic space
        F = K_ext * (x_t - x)
```

```
        # Step 2: Map the force field from the extrinsic space into virtual torque field in the intrinsic space

        T = Jac_T * F

        # Step 3: Relax the arm configuration to the applied field

        q_dot = A_int_T * T

        # Step 4: Map the arm movement into the extrinsic workspace

        x_dot = Jac * q_dot

        # Step 5: Integrate over time until equilibrium

        x = x + (x_dot * dt)

        # update state variables

        self.x = x

        self.x_dot = x_dot

        self.q = q

        self.q_dot = q_dot

        self.F = F

        self.tau = T

        self.TBG = TBG
```

**Robot reaching task using IKPY**

Given an object in front of the robot, the task requires the robot to angle the joints of the iCub body parts (The left arm in this task) to reach the object. To achieve this with inverse kinematics, the code from the sample world from *inverse_kinematics.wbt* was implemented into the project. With the robot now being set as a *supervisor.*

```
import math
from controller import Supervisor
from ikpy.chain import Chain
from ikpy.link import OriginLink, URDFLink
import tempfile
import time
```

*Depicted above are the imports used in the code.*

Notable lines are the Webots Supervisor class, which is used to control the simulation. And The lines that import classes from the *IKPy library*, which are used for inverse kinematics calculations.

```
target = robot.getFromDef('Ball')
arm = robot.getSelf()
```

The first line specifies the target of this task (The node defined as "*Ball*")

```
16
17 IKPY_MAX_ITERATIONS = 4
18
```

*This line sets the maximum number of iterations for the inverse kinematics.*

```
link_joint_names = ['base_link','torso_1','torso','torso_2','solid','torso_3','solid_8','left_arm_1','left_arm',
                    'left_arm_2','solid_9','left_arm_3','solid_10','left_elbow','solid_11','left_forearm',
                    'solid_12','left_wrist_1', 'solid_13','left_wrist_2', 'solid_14']
```

*This line defines a list of joint names for the robot arm, currently only the torso and left arm are being included.*

```
active_joints_mask1 = [False,False, False,False,True,True,True,True,True,True,True]
```

*This line defines a list of Boolean values that specify which joints are active, currently only the left arm are enabled*

```
active_joints_mask1 = [False,False, False,False,True,True,True,True,True,True,True]
```

*This line creates an IKPy Chain instance from a URDF file, using the joint names and active joints specified earlier.*

```
motors = []
for link in armChain.links:
    if any(name in link.name for name in link_joint_names):
        print("Link name:",link.name)

        motor = robot.getDevice(link.name)
        motor.setVelocity(1.0)
        position_sensor = motor.getPositionSensor()
        position_sensor.enable(timeStep)
        motors.append(motor)
```

*The code above, is used to loop sets up the motors for each joint in the robot arm, by getting the corresponding device from the Webots simulation, setting its velocity to 1.0, and then enabling the position sensor of the joints.*

```
targetPosition = target.getPosition()
armPosition = arm.getPosition()
```

*Here these lines get the current position of the target object and the robot arm.*

```
x = targetPosition[0] - armPosition[0]
y = targetPosition[1] - armPosition[1]
z = targetPosition[2] - armPosition[2]
```

*Here these lines compute the position of the target relatively to the arm.*

```
initial_position = [0]*11
ikResults = armChain.inverse_kinematics([x, y, z], max_iter=IKPY_MAX_ITERATIONS, initial_position=initial_position)
```

*The first line creates a list of 11 elements, each initialized with a value of zero. This list will then be used as the initial position for the inverse kinematics calculations.*

*Then in the second line, the inverse_kinematics() method is called from the IKPY library, to determine the joint angles needed for a body part of the robot to reach a desired end effector position (The ball in this task).*

1. *[x, y, z]: a list of three elements representing the desired end effector position in 3D space.*
2. *max_iter=IKPY_MAX_ITERATIONS: a keyword argument that specifies the maximum number of iterations to be used in the inverse kinematics calculation. The value of IKPY_MAX_ITERATIONS is not shown in the code you provided, but it likely refers to a constant defined elsewhere in the code.*
3. *initial_position=initial_position: a keyword argument that specifies the initial position for the inverse kinematics calculation. This argument is set to the list created in the first line of the code.*

```
position = armChain.forward_kinematics(ikResults)
print("position:",position)
#squared_distance = (position[0, -1] - x)**2 + (position[1, -1] - y)**2 + (position[2,-1] - z)**2
squared_distance = (position[0, 3] - x)**2 + (position[1, 3] - y)**2 + (position[2, 3] - z)**2
if math.sqrt(squared_distance) > 0.03:
    ikResults = armChain.inverse_kinematics([x, y, z])
print("ikresults:",ikResults)
```

*Here the code above checks whether the robotic arm is already close to the target position (within 0.03 units) and, if not, calculates the joint angles needed to move the arm to the target position using inverse kinematics.*

```
for i in range(len(motors)):
    motors[i].setPosition(ikResults[i+1])
```

*Then finally these lines actuate the arm motors according to the IK results.*

Current state of this implantation is that the robot appears to utilize the results of the inverse kinematics functions to configure the position and angles of the joints to reach the object. But is unable to fully reach the object and stops a notable distance away from the object. The full code implementing this code can be found in the controller folder of the project environment, under the name "*icub_joints_ik.py*"

**Robot reaching task using RTB**

The was an attempt to implement the robotics toolbox class into the inverse_kinematics.wbt previously mentioned, But as of the moment of submission of the report, there is an unresolved importation issue. Hence it is currently not functioning.

## Conclusion

The implementation and testing of the PMP class has not been conducted at this stage of the project. 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. Which are needed to finalize the goals of the project.

## 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

Corke, P. (1996). A robotics toolbox for MATLAB. *IEEE Robotics & Automation Magazine, 3*(1), 24-32. doi:10.1109/100.486658