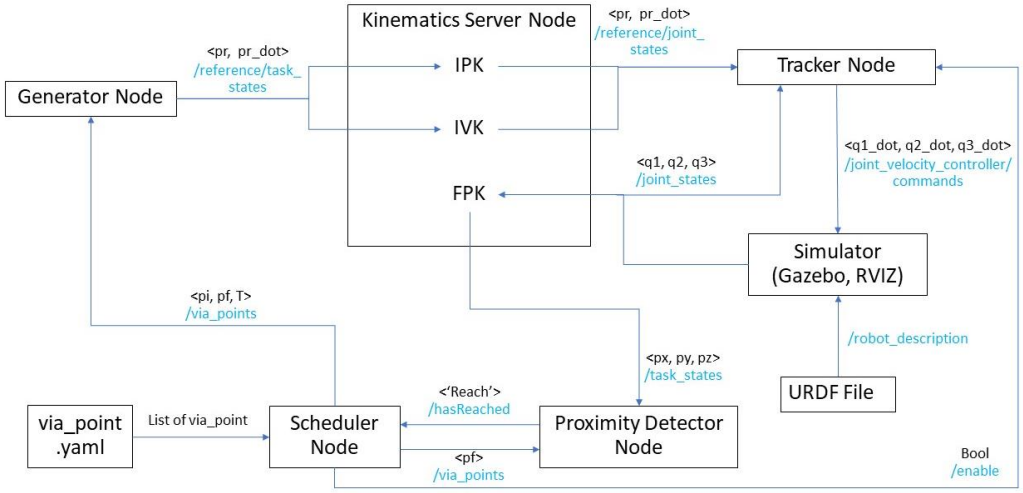
Lab4 Report

1. System Architecture



ขั้นตอนการทำงานทั้งหมดจะเริ่มจากการที่ Scheduler Node รับ list of via point มาจาก yaml file จากนั้นตัว Scheduler จะทำการส่งตำแหน่งปัจจุบัน, ตำแหน่งเป้าหมายและเวลาที่ใช้ในการเคลื่อนที่จากจุดหนึ่งไปยังอีกจุดหนึ่ง ให้กับ Generator node จากนั้น Generator node ก็จะทำการสร้าง Trajectory สำหรับการเคลื่อนที่จากจุดหนึ่งไปยังอีกจุด แล้วส่ง Position และ Velocity ปลายแขนให้กับ Kinematics server เพื่อทำการหา Joint config และ Joint velocity จากนั้นจะนำเข้า Tracker node เพื่อทำ control loop คุมให้แขนกลใน gazebo เคลื่อนที่ไปตาม trajectory ที่ได้วางแผนไว้จาก Generator node โดย Proximity detector node จะทำการ check ว่าตอนนี้แขนกลเคลื่อนที่ถึงตำแหน่งเป้าหมาย (pf) แล้วหรือไม่ ถ้าถึงแล้วก็จะทำการส่ง /hasReached ให้ scheduler node เพื่อทำการสั่งให้หุ่นเคลื่อนที่ไปยังตำแหน่งเป้าหมาย (pf) ถัดไป

2. Launch File

ใน launch file จะประกอบไปด้วย

- การ Simulation หุ่นขึ้น RVIZ

```
package_name = 'advance_control'
rviz_file_name = 'simple_kinematics.rviz'
rviz_file_path = os.path.join(
    get_package_share_directory(package_name),
    'config',
    rviz_file_name
)
rviz_Node = Node(
    package='rviz2',
    executable='rviz2',
    name='rviz',
    arguments=['-d', rviz_file_path],
    output='screen')
```

- การ Simulation หุ่นขึ้น GAZEBO

```
gazebo = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('gazebo_ros'), 'launch'), '/gazebo.launch.py']),
    )


spawn_entity = Node(package='gazebo_ros', executable='spawn_entity.py',
                    arguments=['-topic', 'robot_description',
                               '-entity', 'robot'],
                    output='screen')
```

- การเรียกใช้ yaml file ในส่วนของ controller และ via_point

```
robot_controllers = PathJoinSubstitution(
    [
        FindPackageShare("advance_control"),
        "config",
        "myrobot_controllers.yaml",
    ]
)

config = os.path.join(
    get_package_share_directory('advance_control'),
    'config',
    'via_point.yaml'
    )
```

- การเปิด run node ใน launch file

```
scheduler=Node(
package = 'advance_control',
name = 'scheduler_node',
executable = 'scheduler.py',
parameters = [config]
)

generator=Node(
package = 'advance_control',
executable = 'generator.py'
)

kinematics_server=Node(
package = 'advance_control',
executable = 'kinematics_server.py'
)

tracker=Node(
package = 'advance_control',
name = 'scheduler_node',
executable = 'tracker.py',
parameters = [config]
)

proximity_detector=Node(
package = 'advance_control',
name = 'scheduler_node',
executable = 'proximity_detector.py',
parameters = [config]
)
```

-

3. YAML File
   - การเขียน yaml file : จะใช้ชื่อ namespace ว่า ‘scheduler node’ ros_parameters จะมี Ki, Kp ที่จะถูกเรียกใช้โดย Tracker node , Thereshold ที่จะถูกเรียกใช้โดย Proximity detector node, และ position ในแกน YZ รวมถึงตำแหน่งที่จะต้องยก end-effector ขึ้น ที่จะถูกเรียกใช้โดย Scheduler node

```yaml
scheduler_node:
  ros__parameters:
    thereshold: 0.005
    kp: 3.5
    ki: 0.5
    p_y: [0.15, 0.15, 0.1, 0.15, 0.15, 0.1, 0.1, 0.05, 0.05, 0.05, 0.05, -0.05, -0.05, -0.075, -0.05, -0.075, -0.05, -0.05, -0.05, -0.1, -0.1, -0.15, -0.15, -0.1, -0.1, 0.0]
    p_z: [0.05, 0.09, 0.09, 0.09, 0.185, 0.185, 0.185, 0.185, 0.185, 0.05, 0.05, 0.05, 0.05, 0.07, 0.09, 0.13, 0.185, 0.05, 0.05, 0.05, 0.05, 0.05, 0.185, 0.185, 0.05, 0.185]
    end_up: [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]

```

- การถึง parameter เข้ามาใช้ใน python : จะทำการประกาศ declare parameters และ get_parameters จะที่ declare ไป โดยสามารถเลือกรับค่าออกมาเป็น data type double_array และ double ได้

```python
self.declare_parameters(
    namespace='',
    parameters=[
        ('p_y', None),
        ('p_z', None),
        ('end_up', None)
    ])

self.p_y = self.get_parameter('p_y').get_parameter_value().double_array_value
self.p_z = self.get_parameter('p_z').get_parameter_value().double_array_value
self.end_up = self.get_parameter('end_up').get_parameter_value().double_array_value
```

4. Scheduler

ในส่วนของ scheduler node มีหน้าที่รับ list of via point มาจาก yaml ส่งข้อมูลตำแหน่งปัจจุบัน (pi) ตำแหน่งเป้าหมาย (pf) รวมถึงเวลาที่ใช้ในการเคลื่อนที่จากจุดหนึ่งไปยังอีกจุดหนึ่ง ให้กับ node อื่น และทำหน้าที่ในการ update ตำแหน่งเป้าหมาย เมื่อ /hasReached ส่งมาจาก Proximity node

```python
def reached(self,msg:String): #เรียกใช้งานเมื่อ proximity check แล้วว่าตอนนี้แขนถึงตำแหน่งเป้าหมายแล้ว
    send_via_point = Float32MultiArray()
    enable = Bool()

    if msg.data == 'Reach':
        # enable false to Tracker node
        enable.data = False
        self.enable_publisher.publish(enable)

        if self.count < len(self.p_y) - 1:
            send_via_point.data = [self.p_y[self.count],self.p_z[self.count],self.p_y[self.count+1],self.p_z[self.count+1],self.T,self.end_up[self.count]] #update via_point

            self.get_logger().info(f'send_via_point: {self.count}')

            # publish via_points
            self.via_point_publisher.publish(send_via_point)

            # enable True to Tracker node
            enable.data = True
            self.enable_publisher.publish(enable)

            self.count += 1
```

5. Trajectory Generator

- ในส่วนของ Trajectory Generator Node มีหน้าที่สร้างเส้นทางการเคลื่อนที่ของ via_point ที่ส่งมาจาก scheduler node โดยจะแบ่งเป็นการสร้าง Trajectory ในส่วนของ position และ velocity
- ในส่วนของ position จะทำการสร้างเป็นกราฟเส้นตรงตลอดการเคลื่อนที่ จากความสัมพันธ์

$$\mathbf{q}_\mathrm{r}(t) = \big(1 - \alpha(t)\big) \cdot \mathbf{q}_\mathrm{i} + \big(\alpha(t)\big) \cdot \mathbf{q}_\mathrm{f}$$

โดยที่

$$\alpha(0) = 0$$
$$\alpha(T) = 1$$

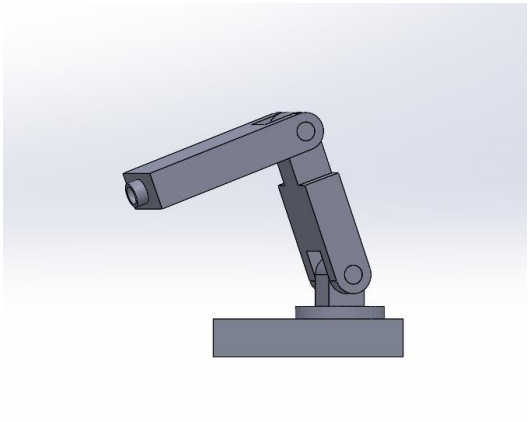
- ในส่วนของ velocity จะทำการสร้างกราฟที่ให้ความเร็วนั้นคงที่อยู่ที่ค่าค่าหนึ่ง ตลอดการเคลื่อนที่ จากความสัมพันธ์

$$\dot{\mathbf{q}}_{\mathrm{r}}(t) = \dot{\alpha}(t) \cdot (\mathbf{q}_{\mathrm{f}} - \mathbf{q}_{\mathrm{i}})$$

โดยที่

$$\dot{\sigma}(0) = 0$$
$$\dot{\sigma}(T) = 0$$
$$\dot{\sigma}(t) = 0.05$$

6. Kinematics Server



$i-1$	i	a_i	α_i	d_i	θ_i
0	1	-0.025	0	0.03	0
1	2	0	$\frac{\pi}{2}$	0.035	0
2	3	0	0	0.12	0

a. Forward Position Kinematics
   - ทำการหา Homogenous matrix ของแต่ละ joint เทียบ world frame จากนั้นทำการคูณ matrix เทียบ frame ไปเรื่อย ๆ เพื่อให้ได้ Homogenous matrix ของปลายแขนเทียบกับ world frame และดึงค่าในส่วนของ translation มาใช้

```python
def get_homo(self,q1,q2,q3):

    # ที่มาอยู่ในกระดาษทด
    h0_1 = np.array([[cos(q1), -sin(q1), 0, -0.025],
            [sin(q1) , cos(q1), 0, 0],
            [0, 0, 1, 0.03],
            [0, 0, 0, 1]])  #homogeneous matrix of joint 1 relative with base_link

    h1_2 = np.array([[cos(q2), -sin(q2), 0, 0],
            [0, 0, -1, 0],
            [sin(q2), cos(q2), 0, 0.035],
            [0, 0, 0, 1]])  #homogeneous matrix of joint 3 relative with joint 2

    h2_3 = np.array([[cos(q3), -sin(q3), 0, 0],
            [sin(q3) , cos(q3), 0, 0.12],
            [0, 0, 1, 0],
            [0, 0, 0, 1]])  #homogeneous matrix of joint 1 relative with base_link

    h3_e = np.array([[1, 0, 0, 0.145],
            [0, 0, -1, 0],
            [0, 1, 0, 0],
            [0, 0, 0, 1]])  #homogeneous matrix of joint end effector relative with joint 3

    h0_2 = h0_1 @ h1_2
    h0_3 = h0_2 @ h2_3
    h0_e = h0_3 @ h3_e

    h0_1 = h0_1[(0,1,2),:]
    h0_2 = h0_2[(0,1,2),:]
    h0_3 = h0_3[(0,1,2),:]
    h0_e = h0_e[(0,1,2),:]

    p0_1 = h0_1[:,-1]       #Position ของ Joint 1 เทียบกับ Frame Gazebo
    p0_2 = h0_2[:,-1]       #Position ของ Joint 2 เทียบกับ Frame Gazebo
    p0_3 = h0_3[:,-1]       #Position ของ Joint 3 เทียบกับ Frame Gazebo
    self.p0_e = h0_e[:,-1]      #Position ของ End-effector เทียบกับ Frame Gazebo
```

- ทำการเรียก publish topic /robot_marker ทุกๆ ครั้งที่ปลายแขนนั้นทำการเขียนตัวหนังสือ

```python
def solve_pos_fk(self,j1,j2,j3):
    self.id_count += 1
    self.get_homo(j1,j2,j3)
    self.x = self.p0_e[0]
    self.y = self.p0_e[1]
    self.z = self.p0_e[2]
    send_task_states = Float32MultiArray()
    send_marker = Marker()

    send_marker.header.frame_id = 'world'
    send_marker.id = self.id_count

    send_marker.type = Marker.SPHERE

    send_marker.action = Marker.ADD

    send_marker.scale.x = 0.01
    send_marker.scale.y = 0.01
    send_marker.scale.z = 0.01

    send_marker.color.r = 1.0
    send_marker.color.b = 0.0
    send_marker.color.a = 1.0
    send_marker.color.g = 0.0

    send_marker.pose.orientation.w = 1.0
    send_marker.pose.position.x = self.x
    send_marker.pose.position.y = self.y
    send_marker.pose.position.z = self.z

    send_task_states.data = [self.x,self.y,self.z]
    self.task_states_publihser.publish(send_task_states)

    if self.end_up == 0.0:
        self.maker_publihser.publish(send_marker)
```

b. Inverse Position Kinematics

ทำการ Solve Position ปลายแขนโดยใช้วิธี Geometric approach

```python
def solve_pos_ik(self,x,y,z):
    #base_offset
    x = x + 0.025
    z = z - 0.03 #base
    z = z - 0.035 #joint1 to joint2
    d1 = 0.12
    d2 = 0.145
    config = -1

    r = math.sqrt((x*x)+(y*y)+(z*z))
    self.joint1 = math.atan2(y,x)

    c2 = ((x*x)+(y*y)+(z*z)-(d1*d1)-(d2*d2))/2/d1/d2
    s2 = config * np.sqrt(1-(c2*c2))
    self.joint3 = math.atan2(s2,c2) + (math.pi/2)
    self.joint2 = math.asin(z/r) - math.atan2(d2*s2,d1+(d2*c2)) - (math.pi/2)
```

c. Inverse Velocity Kinematics

- ทำการหา Jacobian Matrix ในส่วนของ translation XYZ จากนั้นเอา Jacobian Matrix ที่ได้ทำการ inverse และนำไปคูณกับ twist ของ translation ปลายแขน เพื่อทำการหา velocity ของแต่ละ Joint

```python
r0_1 = h0_1[:,(0,1,2)]  #Rotation Matrix ของ Joint 1 เทียบกับ Frame Gazebo
r0_2 = h0_2[:,(0,1,2)]  #Rotation Matrix ของ Joint 2 เทียบกับ Frame Gazebo
r0_3 = h0_3[:,(0,1,2)]  #Rotation Matrix ของ Joint 3 เทียบกับ Frame Gazebo
r0_e = h0_e[:,(0,1,2)]  #Rotation Matrix ของ End-effector เทียบกับ Frame Gazebo

z1 = r0_1[:,2]  #การดึงค่า Z1 จาก Rotation Matrix r0_1
z2 = r0_2[:,2]  #การดึงค่า Z2 จาก Rotation Matrix r0_2
z3 = r0_3[:,2]  #การดึงค่า Z3 จาก Rotation Matrix r0_3

# Jacobain แถว 1-3 ซึ่งเป็นส่วนของ Angular Velocity
J1 = [z1[0],z2[0],z3[0]]
J2 = [z1[1],z2[1],z3[1]]
J3 = [z1[2],z2[2],z3[2]]

# Jacobain แถว 4-6 ซึ่งเป็นส่วนของ Linear Velocity โดยคำนวณตามแนวคิดที่อยู่ในกระดาษทดข้อแรก
J4 = [np.cross(z1, (self.p0_e-p0_1))[0],np.cross(z2, (self.p0_e-p0_2))[0],np.cross(z3, (self.p0_e-p0_3))[0]]
J5 = [np.cross(z1, (self.p0_e-p0_1))[1],np.cross(z2, (self.p0_e-p0_2))[1],np.cross(z3, (self.p0_e-p0_3))[1]]
J6 = [np.cross(z1, (self.p0_e-p0_1))[2],np.cross(z2, (self.p0_e-p0_2))[2],np.cross(z3, (self.p0_e-p0_3))[2]]

J = [J4,J5,J6]
J = np.asarray(J)
if (abs(np.linalg.det(J)) > 0.001):
    #print(J)
    self.inv = np.linalg.inv(J)
else:
    print("Sing")
```

```python
def solve_velo_ik(self,vel_x,vel_y,vel_z):
    self.get_homo(self.joint1,self.joint2,self.joint3)
    self.velmat = np.array([[vel_x],[vel_y],[vel_z]])
    ans = self.inv @ self.velmat
    self.joint1_dot = ans[0][0]
    self.joint2_dot = ans[1][0]
    self.joint3_dot = ans[2][0]

    print("velo_ik",self.joint1_dot,self.joint2_dot,self.joint3_dot)
```

7. Trajectory Tracker

ทำหน้าที่ในการรับค่า joint config และ joint velocity จาก Kinematics server เพื่อ PI Control ให้ได้ joint velocity ไปคุมให้ตำแหน่งปลายแขนใน gazebo นั้นไปยังตำแหน่งเป้าหมาย (pf) ที่เราต้องการ โดยความสัมพันธ์ที่จะเอาไป implement ลงในโปรแกรม คือ

$$\pi(t,q) = \dot{q}_{\mathrm{r}}(t) + K_p \cdot (q_{\mathrm{r}}(t) - q) + K_i \cdot \int_{\tau=0}^{t} (q_{\mathrm{r}}(\tau) - q)\ \mathrm{d}\tau$$

8. Proximity Detector

ทำหน้าที่ check ว่าตอนนี้ตำแหน่งของปลายแขนใน simulation นั้นถึงตำแหน่งเป้าหมายแล้วหรือไม่ โดยวิธีการตรวจสอบจะทำโดยการรับ /task_states ที่จาก forward position kinematics มาเทียบกับตำแหน่งเป้าหมาย (pf) ว่าอยู่ใน thereshold ที่กำหนดไว้หรือไม่ ถ้าอยู่ใน thereshold ที่กำหนดก็จะทำการส่ง /hasReached ไปให้กับ Scheduler

```
def task_states(self,msg:Float32MultiArray):

    if self.update_point == 1:
        #Recieve value from forward position kinematics
        self.p_x = msg.data[0]
        self.p_y = msg.data[1]
        self.p_z = msg.data[2]

        if abs(self.pf_x - self.p_x) <= self.thereshold and abs(self.pf_y - self.p_y) <= self.thereshold and abs(self.pf_z - self.p_z) <= self.thereshold:
            send_reach = String()
            send_reach.data = "Reach"

            self.reached_publihser.publish(send_reach)
            self.update_point = 0
```