1. Differential Drive Low-Level controller

Twist 값이 주어졌을 때, 해당 값과 같이 Differential Drive 로봇이 움직이게 하기 위해서는 각 바퀴의 회전속도를 구하여야 한니다. 이를 위해서 Differential Drive Kinematics 공식을 응용하면

$$lw + rw = x$$
$$rw - lw = L\omega_z$$

라는 것을 쉽게 알 수 있습니다. 그리고 각각의 lw, rw 는 아래와 같이 구해지고,

$$lw = r\dot{\omega}_l, \quad rw = r\dot{\omega}_r$$

로봇의 제원에서 바퀴의 지름 $2r = 0.066(m)$, 바퀴 사이의 간격 $L = 0.16(m)$ 임을 알 수 있습니다. 따라서 각 바퀴의 회전 속도는 아래의 수식을 풀어 구할 수 있습니다.

$$\dot{w}_l = \frac{x}{r} - \frac{L\omega_z}{2r}$$
$$\dot{w}_l = \frac{x}{r} + \frac{L\omega_z}{2r}$$

이 수식을 코드에 적용하면 아래와 같이 실행 코드를 만들 수 있습니다. 이를 실행하면 아래 링크의 영상과 같이 "$rqt_robot_steering$" 패키지를 통해 생성된 $/cmd_vel$ 토픽에 따라 각 바퀴의 회전 속도가 계산되어 동작에 반영됩니다.

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <trajectory_msgs/JointTrajectory.h>

geometry_msgs::Twist twist_;

void twist_subscriber_callback(const geometry_msgs::Twist::ConstPtr& twist)
{
 twist_ = *twist;
 ROS_INFO_STREAM_THROTTLE(10, "cmd_vel changed. " << twist->linear.x << ", " << twist->angular.z);
}

int main(int argc, char** argv)
{
 ros::init(argc, argv, "differential_controller");
 ros::NodeHandle nh;

 ros::Publisher joint_cmd_pub_ = nh.advertise<trajectory_msgs::JointTrajectory>("/joint_cmd", 10);
 ros::Subscriber twist_sub_ = nh.subscribe<geometry_msgs::Twist>("/cmd_vel", 10, twist_subscriber_callback

 trajectory_msgs::JointTrajectory state;
 state.header.frame_id = "base";
 state.joint_names.push_back("wheel1");
 state.joint_names.push_back("wheel2");
 state.points.push_back(trajectory_msgs::JointTrajectoryPoint());
 state.points.front().positions.push_back(0.0);
 state.points.front().positions.push_back(0.0);
```

```
 state.points.front().velocities.push_back(0.0);
 state.points.front().velocities.push_back(0.0);

 const double wheel_distance = 0.16;
 const double wheel_radius = 0.033;
 const double wheel_circumference = wheel_radius * 2 * M_PI;

 ros::Time last_update = ros::Time::now();

 while (ros::ok())
 {
  double linear_speed = twist_.linear.x;
  double angular_speed = twist_.angular.z;

  double linear_wheel_speed = linear_speed / wheel_radius;
  double angular_wheel_speed = (wheel_distance * angular_speed) / (wheel_radius * 2);

  state.points.front().velocities[0] = linear_wheel_speed - angular_wheel_speed;
  state.points.front().velocities[1] = linear_wheel_speed + angular_wheel_speed;

  ROS_INFO_STREAM_THROTTLE(1,
  "Velocity published. " << state.points.front().velocities[0] << ", " << state.points.front().velocities[

  joint_cmd_pub_.publish(state);
  ros::spinOnce();
 }
}
```

2. Macanum Wheels Low-Level Controller

마찬가지로 Macanum Wheel 의 동작을 구현할 때에도 각 바퀴의 회전 속도를 구해야 합니다. 다만 Macanum Wheel 은 Kinematics 에 사용되는 행렬이 정방행렬이 아니기 때문에 Matrix Inverse 를 이용해서 Twist 에 따라 각 바퀴의 속도를 연산하는것이 어렵습니다. 대신 이 바퀴들이 각각 정방향으로 회전하였을 때 어느방향으로 움직이는 힘이 작용하는지를 행렬형태로 정리할 수 있습니다. 여기에서 1번과 3번 바퀴의 회전축이 $(0, -1, 0)$ 이라서 정방향 회전시 2번 4번 바퀴와 반대로 돌기 때문에 이를 반영하면 아래와 같은 행렬이 구해집니다.

$$\alpha = l_1 + l_2 = 0.1125 + 0.0925 = 0.205$$

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} -1 & -1 & -\alpha \\ 1 & -1 & -\alpha \\ -1 & 1 & -\alpha \\ 1 & 1 & -\alpha \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix}$$

따라서 우측 벡터에 원하는 속도와 각속도를 넣으면 각 휠의 동작 속도를 구할 수 있게 됩니다. 이를 코드로 구현하면 아래와 같이 작성됩니다.

```
#include <ros/ros.h>
#include <Eigen/Eigen>
#include <geometry_msgs/Twist.h>
```

```
#include <trajectory_msgs/JointTrajectory.h>

geometry_msgs::Twist twist_;

void twist_subscriber_callback(const geometry_msgs::Twist::ConstPtr& twist)
{
 twist_ = *twist;
 ROS_INFO_STREAM_THROTTLE(10, "cmd_vel changed. " << twist->linear.x << ", " << twist->angular.z);
}

int main(int argc, char** argv)
{
 ros::init(argc, argv, "mecanum_drive");
 ros::NodeHandle nh;

 ros::Publisher joint_cmd_pub_ = nh.advertise<trajectory_msgs::JointTrajectory>("/joint_cmd", 10);
 ros::Subscriber twist_sub_ = nh.subscribe<geometry_msgs::Twist>("/cmd_vel", 10, twist_subscriber_callback

 trajectory_msgs::JointTrajectory state;
 state.header.frame_id = "base";
 state.joint_names.push_back("wheel1");
 state.joint_names.push_back("wheel2");
 state.joint_names.push_back("wheel3");
 state.joint_names.push_back("wheel4");
 state.points.push_back(trajectory_msgs::JointTrajectoryPoint());
 state.points.front().velocities.push_back(0.0);
 state.points.front().velocities.push_back(0.0);
 state.points.front().velocities.push_back(0.0);
 state.points.front().velocities.push_back(0.0);

 // mecanum wheel radius : 0.5 (subwheel was not considered, but it may ok)
 // l1 = 0.1125, l2 = 0.0925
 const double alpha = 0.1125 + 0.0925;
 Eigen::Matrix<double, 4, 3> macanum_kinematics_matrix;
 macanum_kinematics_matrix <<
 -1, -1, -alpha,
 1, -1, -alpha,
 -1, 1, -alpha,
 1, 1, -alpha;
 macanum_kinematics_matrix /= 0.05; // wheel radius

 while (ros::ok())
 {
  Eigen::Vector3d velocities(twist_.linear.x, twist_.linear.y, twist_.angular.z);
  Eigen::Vector4d angular_velocities = macanum_kinematics_matrix * velocities;
  state.points.front().velocities[0] = angular_velocities(0);
  state.points.front().velocities[1] = angular_velocities(1);
  state.points.front().velocities[2] = angular_velocities(2);
```

```
  state.points.front().velocities[3] = angular_velocities(3);

  ROS_INFO_STREAM_THROTTLE(1,
  "Velocity published. " <<
  state.points.front().velocities[0] << ", " << state.points.front().velocities[1] << ", " <<
  state.points.front().velocities[2] << ", " << state.points.front().velocities[3]);

  joint_cmd_pub_.publish(state);

  ros::spinOnce();
 }
}
```

해당 코드를 빌드하여 실행하면 아래와 같이 실행 결과를 얻을 수 있습니다.

3. 4R Robot Arm IK

우선 로봇 팔의 역기구학을 풀기 위해서 파라미터를 아래와 같이 정리하였습니다.

$$d_1 = \begin{bmatrix} 0 \\ 0 \\ 0.0595 \end{bmatrix}, \quad l_2 = \begin{bmatrix} 0.024 \\ 0 \\ 0.128 \end{bmatrix} \quad l_3 = \begin{bmatrix} 0.124 \\ 0 \\ 0 \end{bmatrix}, \quad l_4 = \begin{bmatrix} 0.0817 \\ 0 \\ 0 \end{bmatrix}$$

여기에 목표위치를 P_t 라고 두었을 때, P_t 위치와 회전으로 로봇 팔이 정렬되기 위해서는 P_t 의 z 축 방향에 로봇의 마지막 Joint 가 정렬되어야 합니다. 따라서 마지막 joint 이후의 link 인 l_4 의 길이 만큼 P_t 위치를 기준으로 한 $-z$ 방향으로 나머지 l_1, l_2, l_3 를 배치할 수 있다면 목표 지점의 IK 값을 구할 수 있습니다. 따라서 우선 Offset 된 위치를 아래와 같이 계산하였습니다.

$\mathrm{p}_w = p_t + l_4\vec{z_t}$
$P_t\vec{x_t}\vec{y_t}\vec{z_t}p_t0001$

$\mathrm{p}_w = p_t + l_4\vec{z_t}$

남은 l_1, l_2, l_3 중 z 축을 회전축으로 가지는 회전축은 1축밖에 없고, 따라서 우선 x, y 성분을 이용해 로봇 팔의 방향을 정렬할 수 있습니다.

$$q_1 = \mathrm{atan2}(p_{wy}, p_{wx})$$

이후 남은 두 Y 축 회전 Joint 를 이용해 p_w 까지의 자세를 만들어야 하고, 이는 l_2, l_3 의 길이와 p_w 까지의 거리를 가지고 코사인법칙을 이용하면 각각의 각을 구할

수 있습니다.

$$m = ||((||p_{wx}, p_{wy})||^2, p_{wz} - d_1)||_2$$

$$\begin{aligned}
\phi &= \text{atan2}(l_{2x}, l_{2z}) \\
\gamma &= \frac{l_2^2 + l_3^2 - m^2}{2l_2l_3} \\
\beta &= \frac{m^2 + l_2^2 - l_3^2}{2ml_2} \\
q_2 &= \frac{\pi}{2} - \alpha - \beta - \phi \\
q_3 &= \phi - \gamma - \frac{\pi}{2}
\end{aligned}$$

여기까지 완료되면 q_4 값만 구하면 되는데, 우선 이를 위해서 q_1, q_2, q_3 가 반영된 FK 를 먼저 해석해야 합니다.

$$P_{q_1q_3} = \text{trans}(d_1) \times \text{rotz}(q_1) \times \text{roty}(q_2 + \phi) \times \text{trans}(l_2) \times \text{rotz}(q_3 - \phi) \times \text{trans}(l_3)$$

FK 결과가 가지고 있는 Rotation Matrix 와 목표 위치의 Rotation Matrix 를 이용해 q_4 의 각도를 구합니다.

$$\begin{aligned}
P_{q_1q_3} &= \begin{bmatrix} R_{q_1q_3} & p_{q_1q_3} \\ 0 & 1 \end{bmatrix} \\
R_{wt} &= R_{q_1q_3}^T R_t \\
q_4 &= \text{atan2}(R_{wt}^{(0,2)}, R_{wt}^{(0,0)})
\end{aligned}$$

따라서 각 joint 의 값인 q_1, q_2, q_3, q_4 를 구할 수 있게 되고, 이를 코드로 반영하면 아래와 같은 코드로 만들 수 있습니다. 다만 실제 구현 코드의 대부분이 목표위치를 지정하기 위한 Interactive Marker 를 표시하는 코드이기 때문에 해당 부분의 코드를 제외하고 실제 IK 부분의 코드만 리포트에 첨부하고, 전체 코드는 아래의 github repository 에 업로드 해 두었습니다.

```
...

geometry_msgs::Pose z_rotation_pose_;
geometry_msgs::Pose target_pose_;

std::vector<double> arm_target;

inline double get_center_angle_using_three_length(double a, double b, double c)
{
 return std::acos((a * a + b * b - c * c) / (2 * a * b));
}

Eigen::Matrix4d pose_to_marix(geometry_msgs::Pose pose)
{
 Eigen::Matrix4d matrix = Eigen::Matrix4d::Identity();
 matrix.block<3, 3>(0, 0) = Eigen::Quaterniond(pose.orientation.w, pose.orientation.x, pose.orientation.y,
 matrix(0, 3) = pose.position.x;
 matrix(1, 3) = pose.position.y;
 matrix(2, 3) = pose.position.z;
 return matrix;
```

```
}

void solve_robot_arm_ik()
{
 Eigen::Vector3d link3_to_4(0.024, 0, 0.128);

 const double d1 = 0.0595;
 const double link3_length = std::sqrt(0.024 * 0.024 + 0.128 * 0.128);
 const double link4_length = 0.124;
 const double link5_length = 0.0817;

 const double link3_angle = std::atan2(0.024, 0.128);

 Eigen::Matrix4d target_transform = Eigen::Matrix4d::Identity();
 target_transform.block<3, 3>(0, 0) = pose_to_marix(z_rotation_pose_).block<3, 3>(0, 0);
 target_transform = target_transform * pose_to_marix(target_pose_);

 Eigen::Vector3d wrist_position = target_transform.block<3, 1>(0, 3) - target_transform.block<3, 1>(0, 2)

 double x_prime = Eigen::Vector2d(wrist_position.block<2, 1>(0, 0)).norm();
 Eigen::Vector2d m_vec = Eigen::Vector2d(x_prime, wrist_position.z() - d1);
 double m = m_vec.norm();
 double alpha = std::atan2(m_vec.y(), m_vec.x());

 double gamma = get_center_angle_using_three_length(link3_length, link4_length, m);
 double beta = get_center_angle_using_three_length(m, link3_length, link4_length);

 if (gamma > 0.0) gamma -= M_PI; // to make value in range of -M_PI ~ M_PI
 if (beta < 0.0) beta += M_PI;   // to make value in range of -M_PI ~ M_PI

 double q1 = std::atan2(wrist_position.y(), wrist_position.x());
 double q2 = M_PI / 2 - alpha - beta - link3_angle;
 double q3 = -(gamma - link3_angle + M_PI / 2);
 Eigen::Matrix4d link4_position = Eigen::Affine3d(
  Eigen::Translation3d(0, 0, d1) *
  Eigen::AngleAxisd(q1, Eigen::Vector3d::UnitZ()) *
  Eigen::AngleAxisd(q2 + link3_angle, Eigen::Vector3d::UnitY()) *
  Eigen::Translation3d(0, 0, link3_length) *
  Eigen::AngleAxisd(q3 - link3_angle, Eigen::Vector3d::UnitY()) *
  Eigen::Translation3d(0, 0, link4_length)).matrix();

 Eigen::Matrix3d orientation = link4_position.block<3, 3>(0, 0).transpose() * target_transform.block<3, 3>
 double q4 = std::atan2(orientation(0, 2), orientation(0, 0)) - M_PI / 2;

 std::cout << "Solution : " << q1 << ", " << q2 << ", " << q3 << ", " << q4 << std::endl;
 std::vector<double> solution;
 solution.push_back(q1);
 solution.push_back(q2);
 solution.push_back(q3);
 solution.push_back(q4);
 arm_target = solution;
}
```

```
...

void joint_state_callback(const sensor_msgs::JointState::ConstPtr& state)
{
 if (arm_target.size() == 0) return;

 trajectory_msgs::JointTrajectory joint_trajectory;
 joint_trajectory.header.frame_id = "base";
 joint_trajectory.joint_names.push_back("Arm1");
 joint_trajectory.joint_names.push_back("Arm2");
 joint_trajectory.joint_names.push_back("Arm3");
 joint_trajectory.joint_names.push_back("Arm4");
 joint_trajectory.points.push_back(trajectory_msgs::JointTrajectoryPoint());
 joint_trajectory.points.front().positions.push_back(arm_target.at(0));
 joint_trajectory.points.front().positions.push_back(arm_target.at(1));
 joint_trajectory.points.front().positions.push_back(arm_target.at(2));
 joint_trajectory.points.front().positions.push_back(arm_target.at(3));

 joint_cmd_pub_.publish(joint_trajectory);
}

int main(int argc, char** argv)
{
 ros::init(argc, argv, "robot_arm_drive");
 ros::NodeHandle nh;

 ...

 joint_cmd_pub_ = nh.advertise<trajectory_msgs::JointTrajectory>("/joint_cmd", 10);
 ros::Subscriber joint_state_sub_ = nh.subscribe<sensor_msgs::JointState>("/joint_states", 10, joint_state

 ros::spin();

 ...
 return 0;
}
```

해당 코드를 빌드하여 실행하면 아래 영상과 같은 결과를 얻을 수 있습니다.