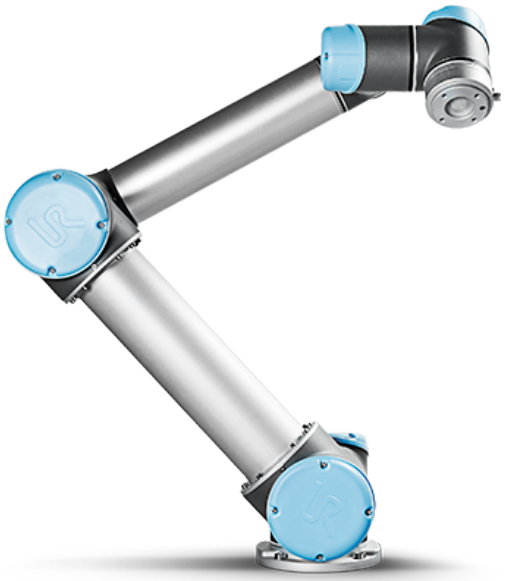
IMPLEMENTACJA MODUŁU DO PLANOWANIA RUCHU MANIPULATORA UR5 W ROS2 (HUMBLE) Z WYKORZYSTANIEM MOVEIT

METODY I ALGORYTMY PLANOWANIA RUCHU

Piórkowska Agnieszka 144548
Gajewski Miłosz 144550
Roboty i Systemy Autonomiczne



Politechnika Poznańska
Wydział Automatyki Robotyki i Elektrotechniki
Automatyka i Robotyka

Poznań, 20 maja 2023

Spis treści

Spis rysunków

Spis listingów kodów

1 Zakres dokumentu

Dokument jest raportem z realizacji projektu implementacji modułu do planowania ruchu manipulatora UR5 w ROS2 (Humble) z wykorzystaniem modułu MoveIt w ramach przedmiotu Metody i Algorytmy Ruchu dla kierunku Automatyka i Robotyka, specjalność Roboty i Systemy Autonomiczne. W raporcie znajdują się idea oraz założenia projektowe, etapy realizacji poszczególnych składowych projektu wraz z objaśnieniami, a także przykładowy zestaw instrukcji uruchomieniowych wybranego scenariusza.

2 Założenia projektowe

Celem projektu było uruchomienie symulacji modelu robota UR5 w Rviz oraz implementacja węzła do planowania ruchu robota w języku C++ z wykorzystaniem modułu MoveIt wraz z komunikacją przy pomocy serwisów dostępnych w ROS2. Na poszczególne składowe projektu składało się zaplanowanie trajektorii do celu wyrażonego w joint - space, zaplanowanie trajektorii do celu wyrażonego w cartesian - space, obliczenie trajektorii liniowej w cartesian - space oraz planowanie ścieżki z unikaniem kolizji z obiektami na scenie. Przy implementacji planowania trajektorii należało także umożliwić wybór planera oraz porównać kilka algorytmów planowania.

3 Realizacja projektu

3.1 Komunikacja między węzłami

Komunikacja między węzłami została zrealizowana poprzez ROS2 Services. Serwisy umożliwiają przesyłanie żądań i odbieranie odpowiedzi między węzłami, w przeciwieństwie do tematów, gdzie dane są publikowane i subskrybowane przez wiele węzłów jednocześnie, serwisy obsługują komunikację punkt - punkt. Węzeł żądający usługi nazywany jest klientem, natomiast węzeł dostarczający usługi nazywany jest serwerem. Zdecydowano się na wspomniane rozwiązanie z uwagi na charakter zadania - węzły realizujące trajektorię do celu i planujące trajektorię liniową, a także umożliwiające dodanie obiektów do sceny, wymagają otrzymania konkretnych danych.

Uruchomienie serwera (server) uzyskuje się za pomocą komendy:

```
foo@bar:~/ros2_miapr$ ros2 launch miapr_ur5e trajectory_control_server.launch.py
```

gdzie plik *launch* uruchamia serwer dla usługi ROS2 o nazwie *trajectory_control_server* z określonymi parametrami, takimi jak opis modelu robota i jego kinematyki. Kod źródłowy pliku *trajectory_control_server.launch.py* dostępny jest pod adresem: **trajectory_control_server.launch.py**.

Wywołanie usługi *joint_trajectory_service* z użyciem interfejsu *JointTrajectoryInterface* zdefiniowanego w paczce *miapr_ur5e_interfaces* pozwala na wysłanie trajektorii robota, gdzie cel wyrażony jest w joint - space. Cel definiuje się poprzez ustawienie docelowej wartości każdego z sześciu przegubów, co widać w poniżej załączonej komendzie. Ustawiając konkretną wartość pola *controller* dokonuje się wyboru planera. Domyślnie ustawinym planerem jest *RRTConnectkConfigDefault*. Wybierając *controller: 1*, jak ma to miejsce w poniższym przykładzie, planer ustawiony

jest na *ESTkConfigDefault*, czyli planer ruchu oparty o algorytm EST (Elipse Space Tree), często stosowany do planowania trajektorii w przestrzeni jointów dla robotów manipulacyjnych.

```
foo@bar:~/ros2_miapr$ ros2 service call /joint_trajectory_service
    miapr_ur5e_interfaces/srv/JointTrajectoryInterface "{j1:␣0,␣j2:␣-1.57,␣j3:␣
    -1.57,␣j4:␣-1.57,␣j5:␣1.57,␣j6:␣0,␣controller:␣1}"
```

Listing kodu 1: JointTrajectoryInterface.srv

```
#
# Request: joiny values, Response: status
# Controlers: 0 - SBLkConfigDefault, 1 - ESTkConfigDefault, 2 -
    LBKPIECEkConfigDefault, 3 - BKPIECEkConfigDefault, 4 - KPIECEkConfigDefault, 5 -
     RRTkConfigDefault
# 6 - RRTConnectkConfigDefault, 7 - RRTstarkConfigDefault, 8 - TRRTkConfigDefault,
    9 - PRMkConfigDefault, 10 - PRMstarkConfigDefault
#
float64 j1
float64 j2
float64 j3
float64 j4
float64 j5
float64 j6
uint8 controller
---
bool status
```

Analogicznie ma to miejsce przy planowaniu ruchu do celu wyrażonego w cartesian - space, gdzie podczas wywoływania usługi *cartesian_trajectory_service* z użyciem interfejsu *CartesianTrajectoryInterface* należy podać punkt docelowy jako pozycję wyrażoną we współrzędnych x, y, z oraz orientację podaną w kwaternionach. Tu również możliwy jest wybór jednego spośród dostępnych planerów ruchu poprzez odpowiednie wypełnienie pola *controller*.

```
foo@bar:~/ros2_miapr$ ros2 service call /cartesian_trajectory_service
    miapr_ur5e_interfaces/srv/CartesianTrajectoryInterface "{x:␣0.2,␣y:␣0,␣z:␣0,␣
    qw:␣0,␣qx:␣0,␣qy:␣0,␣qz:␣0,␣controller:␣1}"
```

Listing kodu 2: CartesianTrajectoryInterface.srv

```
#
# Request: position, Response: status
# Controlers: 0 - SBLkConfigDefault, 1 - ESTkConfigDefault, 2 -
    LBKPIECEkConfigDefault, 3 - BKPIECEkConfigDefault, 4 - KPIECEkConfigDefault, 5 -
     RRTkConfigDefault
# 6 - RRTConnectkConfigDefault, 7 - RRTstarkConfigDefault, 8 - TRRTkConfigDefault,
    9 - PRMkConfigDefault, 10 - PRMstarkConfigDefault
#
float64 x
float64 y
float64 z
float64 qw
float64 qx
```

```
float64 qy
float64 qz
uint8 controller
---
bool status
```

Wyznaczania trajektorii liniowej można dokonać za pomocą komendy:

```
foo@bar:~/ros2_miapr$ ros2 service call /cartesian_linear_trajectory_service
    miapr_ur5e_interfaces/srv/CartesianTrajectoryInterface "{x:␣0.2,␣y:␣0,␣z:␣0,␣
    qw:␣0,␣qx:␣0,␣qy:␣0,␣qz:␣0}"
```

gdzie cel określony jest przy pomocy pozycji wyrażonej we współrzędnych x, y, z oraz orientacji wyrażonej przy pomocy kwaternionów.
Projekt przewiduje również możliwość wyznaczenia trajektorii z unikaniem kolizji. Przygotowane zostały serwisy umożliwiające dodawanie oraz usuwanie obiektów ze sceny robota. Dodanie przeszkody może zostać zrealizowane poprzez wywołanie komendy:

```
foo@bar:~/ros2_miapr$ ros2 service call /obstacle_add_service
    miapr_ur5e_interfaces/srv/ObstacjeInterface "{x:␣1,␣y:␣1,␣z:␣-2,␣box_x:␣1,␣
    box_y:␣1,␣box_z:␣1}"
```

gdzie przyjmowanymi parametrami są pozycja, w której ma się znaleźć przeszkoda na scenie oraz wymiary prostopadłościennej przeszkody.

Listing kodu 3: ObstacleInterface.srv

```
#
# Position: x,y,z Dimensions: box_x, box_y, box_z
#
float64 x
float64 y
float64 z
float64 box_x
float64 box_y
float64 box_z
---
bool status
```

Natomiast usuwanie obiektów ze sceny zrealizować można za pomocą:

```
foo@bar:~/ros2_miapr$ ros2 service call /obstacle_del_service
    miapr_ur5e_interfaces/srv/ObstacleDelInterface
```

gdzie interfejs *ObstacleDelInterface* wyrażony jest w pliku *.srv*:

```
#
# Clear the stage
#
bool delete_all
---
bool status
```

3.2 Trajektoria do celu wyrażonego w joint space

Przestrzeń joint - ów odnosi się do możliwych konfiguracji przegubów manipulatora. Opisuje położenie i orientację robota poprzez wartości kątów poszczególnych jointów. Każdy przegub i zagres jego ruchu można kontrolować niezależnie. Zadając cel wyrażony w joint - space manipulator wyznacza trajektorię dla każdego przegubu, by osiągnąć zadaną pozycję.

Wybór celu wyrażonego w joint - space przy pomocy serwisu i interfejsu został opisany w sekcji dotyczącej komunikacji. Natomiast sama realizacja zadania wyznaczania trajektorii do celu wyrażonego w joint - space została zawarta w funkcji *callbackJointTrajectoryService* dostępnej w kodzie **trajectory_control_server.cpp**. Widać tu realizację odbierania danych dotyczących celu:

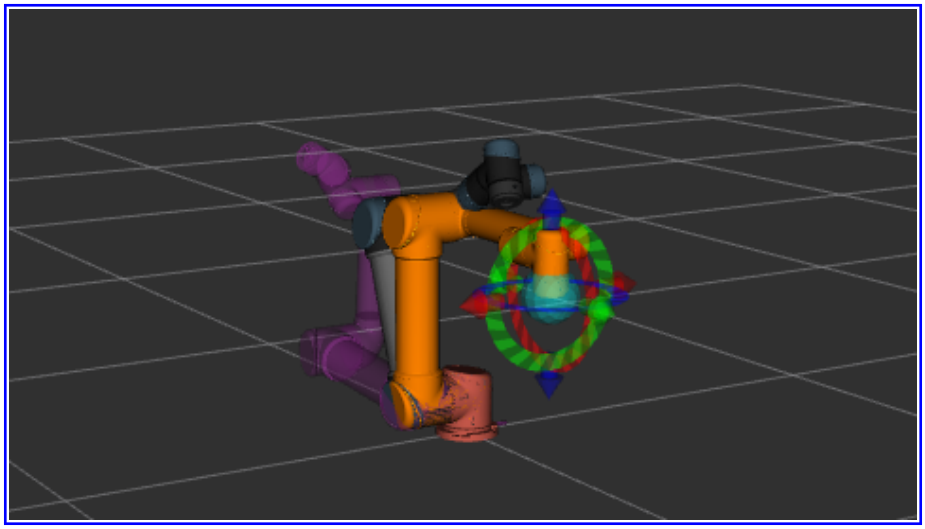
```
std::vector<double> target_joint_values = {request->j1, request->j2, request->j3, request->j4,
    request->j5, request->j6};
```

i ustawianie danych jako cel planowania trajektorii:

```
auto move_group_interface = MoveGroupInterface(move_group_node, "ur_manipulator");
move_group_interface.setJointValueTarget(target_joint_values);
```

Funkcjonalność planowania i egzekwowania ruchu jeśli planowanie się powiodło wyrażone zostało jako:

```
auto const [success, plan] = [&move_group_interface]
{
    moveit::planning_interface::MoveGroupInterface::Plan msg;
    auto const ok = static_cast<bool>(move_group_interface.plan(msg));
    return std::make_pair(ok, msg);
}();
moveit::core::MoveItErrorCode mgi_status;
if (success)
{
    response->status = true;
    mgi_status = move_group_interface.execute(plan);
}
else
{
    RCLCPP_ERROR(this->get_logger(), "Planing failed!");
    response->status = false;
    executor.cancel();
}
```



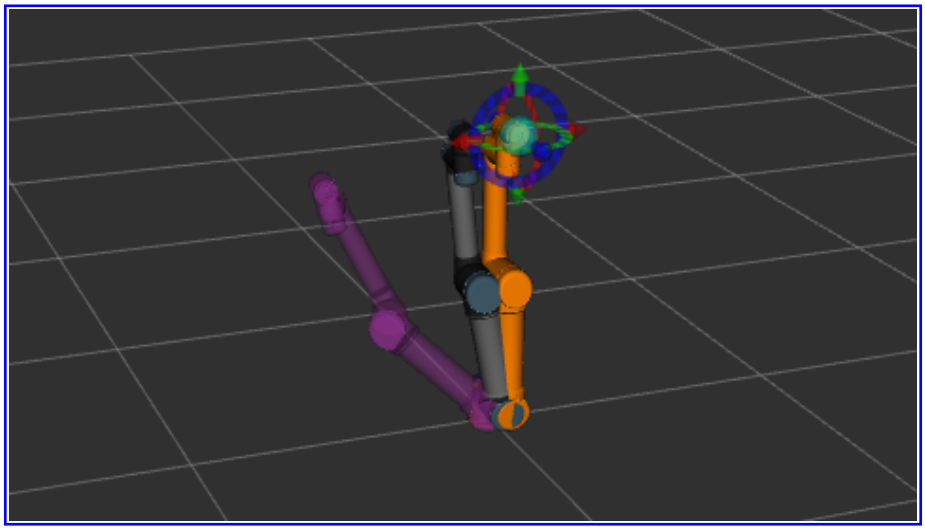
Rysunek 1: Rviz - planowanie ruchu do celu wyrażonego w joint - space

3.3 Trajektoria do celu wyrażonego w cartesian space

Przestrzeń kartezjańska odnosi się do położenia i orientacji manipulatora w trójwymiarowej przestrzeni kartezjańskiej. Jest ona bardziej intuicyjna dla użytkownika (w porównaniu z przestrzenią przegubów), gdyż pozwala bezpośrednio określić położenie i orientację efektora w przestrzeni trójwymiarowej.

Planowanie trajektorii do celu wyrażonego w cartesian - space dostępne jest w funkcji *callbackCartesianTrajectoryService* w kodzie **trajectory_control_server.cpp**. Cel zadaje się w przestrzeni kartezjańskiej:

```
auto const target_pose = [=]
{
    geometry_msgs::msg::Pose msg;
    msg.orientation.w = request->qw;
    msg.orientation.x = request->qx;
    msg.orientation.y = request->qy;
    msg.orientation.z = request->qz;
    msg.position.x = request->x;
    msg.position.y = request->y;
    msg.position.z = request->z;
    return msg;
}();
move_group_interface.setPoseTarget(target_pose);
```



Rysunek 2: Rviz - planowanie ruchu do celu wyrażonego w cartesian - space

3.4 Trajektoria liniowa w cartesian space

Obliczanie trajektorii liniowej możliwe jest dzięki opcji *computeCartesianPath*. Na podstawie położenia początkowego, zadanego punktu docelowego oraz pośrednich punktów, obliczana jest trajektoria liniowa ruchu manipulatora w przestrzeni kartezjańskiej. Funkcja uwzględnia ograniczenia kinematyczne manipulatora oraz skanuje zaplanowaną trajektorię pod względem kolizji z obiektami umieszczonymi na scenie robota. W związku z tym trajektoria liniowa w cartesian - space wyznaczona przy pomocy *computeCartesianPath* jest bezkolizyjna i spełnia wymagania kinematyczne robota. Realizacja projektu zakłada możliwość zadania punktu docelowego w cartesian - space i oblicza pośrednie sekwencje położeń efektora w kolejnych punktach czasu, tak by osiągnąć zada-

ny cel. Podczas obliczania trajektorii liniowej skorzystano z opcjonalnej funkcjonalności zadania ograniczeń na ścieżce efektora poprzez:

```
// Set the end effector trajectory constraints (optional)
moveit_msgs::msg::Constraints trajectory_constraints;
trajectory_constraints.name = "move_constraints";
moveit_msgs::msg::PositionConstraint position_constraint;
position_constraint.header.frame_id = "base_link";
position_constraint.link_name = "end_effector";
position_constraint.target_point_offset.x = 0.1;
position_constraint.target_point_offset.y = 0.1;
position_constraint.target_point_offset.z = 0.1;
position_constraint.constraint_region.primitive_poses.push_back(target_pose);
trajectory_constraints.position_constraints.push_back(position_constraint);
move_group_interface.setPathConstraints(trajectory_constraints);
```

Obliczanie trajektorii liniowej odbywa się we fragmencie:

```
// Compute the Cartesian path
double eef_step = 0.01;  // Step size in meters
double jump_threshold = 0.0;  // No jumping allowed between waypoints
moveit_msgs::msg::RobotTrajectory trajectory;
double fraction = move_group_interface.computeCartesianPath(waypoints, eef_step, jump_threshold,
    trajectory);
```

Kod wyznaczający trajektorię liniową w cartesian space dostępny jest w funkcji *callbackCartesian-LinearTrajectoryService* w kodzie **trajectory_control_server.cpp**.

3.5 Wybór planera z poziomu kodu

Istnieją różne algorytmy planowania ruchu manipulatora, które dobierane powinny być ze względu na charakter zadania, obrany cel i nałożone ograniczenia. Jak zostało to już wcześniej uwzględnione w sekcji dotyczącej komunikacji, wyboru planera dokonuje się poprzez uzupełnienie pola *controller* odpowiednią liczbą. Wybór planera zrealizowany jest w kodzie poprzez:

```
move_group_interface.setNumPlanningAttempts(5);
switch(request->controller){
    case 0:
        move_group_interface.setPlannerId("SBLkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "SBLkConfigDefault planner selected");
        break;
    case 1:
        move_group_interface.setPlannerId("ESTkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "ESTkConfigDefault planner selected");
        break;
    case 2:
        move_group_interface.setPlannerId("LBKPIECEkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "LBKPIECEkConfigDefault planner selected");
        break;
    case 3:
        move_group_interface.setPlannerId("BKPIECEkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "BKPIECEkConfigDefault planner selected");
        break;
    case 4:
        move_group_interface.setPlannerId("KPIECEkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "KPIECEkConfigDefault planner selected");
        break;
    case 5:
        move_group_interface.setPlannerId("RRTkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "RRTkConfigDefault planner selected");
        break;
    case 6:
        move_group_interface.setPlannerId("RRTConnectkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "RRTConnectkConfigDefault planner selected");
        break;
    case 7:
        move_group_interface.setPlannerId("RRTstarkConfigDefault");
```

```
        RCLCPP_INFO(this->get_logger(), "RRTstarkConfigDefault planner selected");
        break;
    case 8:
        move_group_interface.setPlannerId("TRRTkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "TRRTkConfigDefault planner selected");
        break;
    case 9:
        move_group_interface.setPlannerId("PRMkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "PRMkConfigDefault planner selected");
        break;
    case 10:
        move_group_interface.setPlannerId("PRMstarkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "PRMstarkConfigDefault planner selected");
        break;
    default:
        move_group_interface.setPlannerId("RRTConnectkConfigDefault");
        RCLCPP_INFO(this->get_logger(), "RRTConnectkConfigDefault planner selected");
        break;
}
```

3.6 Porównanie algorytmów planowania

Jak zostało wspomniane w sekcji dotyczącej komunikacji między węzłami, zrealizowany projekt przewiduje możliwość wyboru algorytmu planowania. Dostępne do wyboru planery ruchu można podejrzeć w kodzie źródłowym: **trajectory_control_server.cpp**.

3.6.1 ESTkConfigDefault

ESTkConfigDefault jest algorytmem EST (Expansive-Space Trees) dla zadań znalezienia trajektorii manipulatorów. Jest wydajny dla manipulatorów o dużych zakresach ruchu, ale nie daje gwarancji znalezienia optymalnej trasy. Działa odrobinę wolniej w porównaniu z algorytmem Rapidly-Exploring Random Tree, ale zdecydowanie szybciej w porównaniu do Rapidly-Exploring Random Tree Star.

3.6.2 RRTkConfigDefault

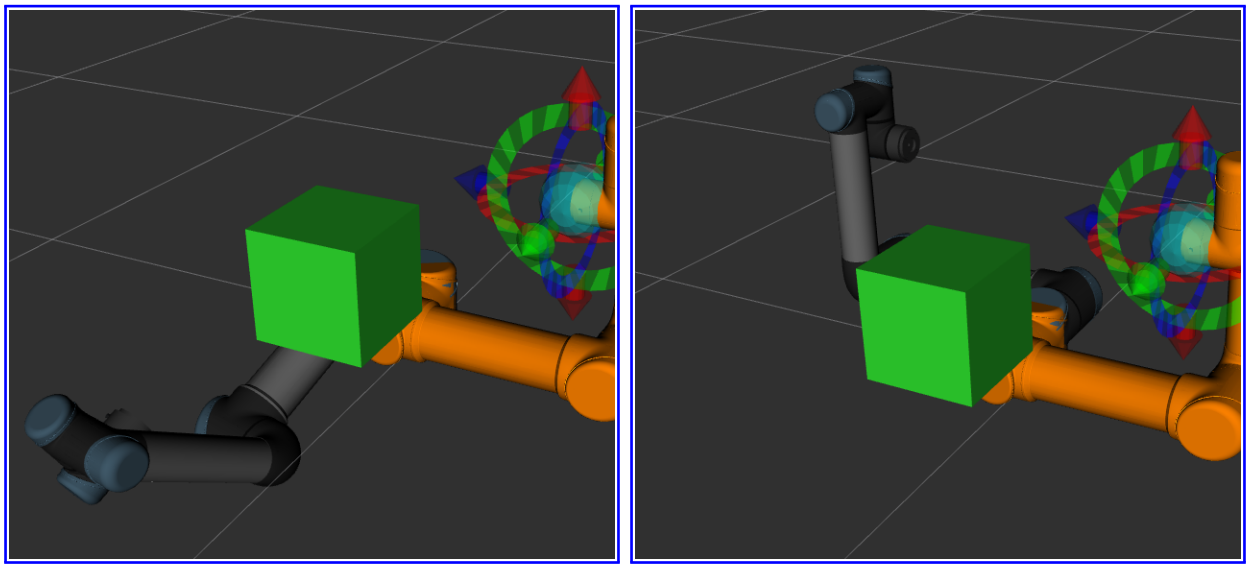
Rapidly-Exploring Random Tree to algorytm tworzący drzewo zgodnie z zasadą najbliższego sąsiada. Jest probabilistycznym algorytmem starającym się wyznaczyć trajektorię w przestrzeni jointów. Opisany algorytm nie daje gwarancji znalezienia najbardziej optymalnej trasy, ale porównując czas planowania trajektorii tego planera z ESTkConfigDefault oraz RRTstarkConfigDefault okazał się najszybszym algorytmem.

3.6.3 RRTstarkConfigDefault

Rapidly-Exploring Random Tree Star jest ulepszoną wersją algorytmu RRT, która stara się znaleźć optymalną trajektorię. Działa wolniej niż podstawowa wersja Rapidly-Exploring Random Tree, jednak jest w stanie zoptymalizować znalezioną trajektorię. Algorytmowi Rapidly-Exploring Random Tree Star planowanie trajektorii zajęło najdłużej spośród trzech porównywanych algorytmów (prawie 90 razy dłużej w porównaniu do Rapidly-Exploring Random Tree i 50 razy dłużej porównując do Expansive-Space Trees), jednak jego zaletą jest gwarancja znalezienia najbardziej optymalnej trasy.

3.7 Unikanie kolizji

W celu dodania obiektu do sceny korzysta się z funkcji *callbackObstacleAddService* dostępnej w **trajectory_control_server.cpp**. Po dodaniu obiektu do sceny, manipulator planując trajektorię uwzględnia unikanie kolizji. Poniżej przedstawiono przykład wykonywania zaplanowanej trajektorii do celu z unikaniem kolizji z obiektem:



Rysunek 3: Rviz - wykonywanie ruchu z unikaniem przeszkody i docelowa pozycja

4 Przykładowy scenariusz

Poniżej przedstawiono zestaw instrukcji jakie należy wykonać chcąc zaplanować trajektorię do celu wyrażonego w cartesian - space wraz z uniknięciem kolizji z dodanym obiektem.

1. Uruchomienie symulatora

```
foo@bar:~/ros2_miapr$ ros2 run ur_robot_driver start_ursim.sh -m ur5
```

2. Uruchomienie symulatora robota

```
foo@bar:~/ros2_miapr$ ros2 launch ur_robot_driver ur_control.launch.py
    ur_type:=ur5 robot_ip:=192.168.56.101 use_fake_hardware:=true
    launch_rviz:=false initial_joint_controller:=joint_trajectory_controller
```

3. Uruchomienie sterownika

```
foo@bar:~/ros2_miapr$ ros2 launch ur_moveit_config ur_moveit.launch.py
    ur_type:=ur5 launch_rviz:=true use_fake_hardware:=true
```

4.Uruchomienie serwera

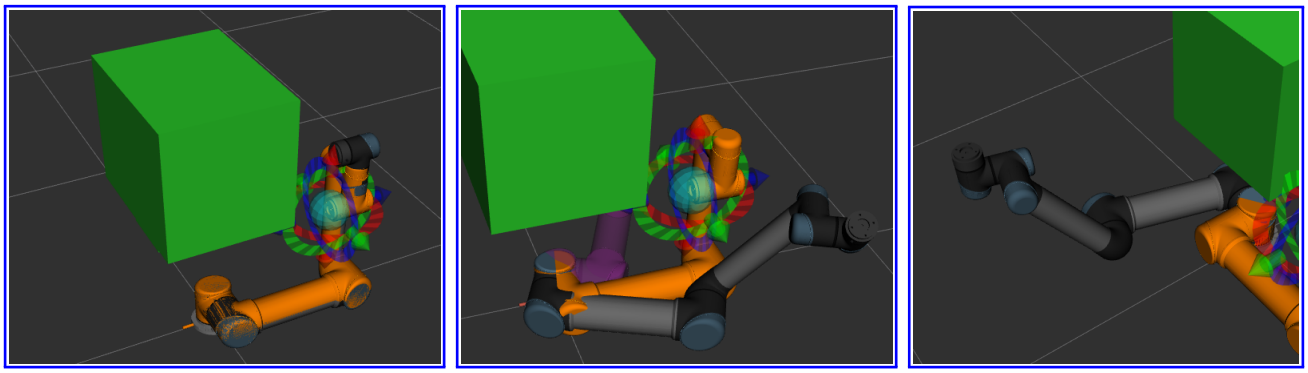
```
foo@bar:~/ros2_miapr$ ros2 launch miapr_ur5e trajectory_control_server.launch.py
```

5. Dodanie przeszkody do sceny

```
foo@bar:~/ros2_miapr$ ros2 service call /obstacle_add_service
    miapr_ur5e_interfaces/srv/ObstacjeInterface "{x:␣-0.4,␣y:␣0.0,␣z:␣0.3,␣box_x:␣
    0.5,␣box_y:␣0.5,␣box_z:␣0.5}"
```

6. Zaplanowanie i wykonanie trajektorii do celu wyrażonego w cartesian - space i przy wykorzystaniu planera ESTkConfigDefault

```
foo@bar:~/ros2_miapr$ ros2 service call /cartesian_trajectory_service
    miapr_ur5e_interfaces/srv/CartesianTrajectoryInterface "{x:␣0.7,␣y:␣-0.3,␣z:␣
    0.5,␣qw:␣0,␣qx:␣0,␣qy:␣0,␣qz:␣0,␣controller:␣1}"
```



Rysunek 4: Rviz - dodanie przeszkody, planowanie trajektorii, pozycja docelowa

5 Wnioski

Realizacja projektu umożliwiła bliższe zaznajomienie się ze sposobem komunikacji jakim są serwisy w ROS2, tworzeniem interfejsów oraz funkcjonalnością modułu MoveIt. Wykonanie projektu pozwoliło również poszerzyć wiedzę z zakresu planowania trajektorii i algorytmów wyznaczania ścieżek, zrozumienia przestrzeni kartezjańskiej, przestrzeni przegubowej oraz planowania trajektorii liniowej.