GROUP 24
Member1: Osti Simone, simone.osti@studenti.unipd.it
Member2: Russo Christian Francesco, christianfrancesco.russo@studenti.unipd.it
Member3: Spinato Matteo, matteo.spinato@studenti.unipd.it
Repository link: https://bitbucket.org/ir2324-group-24/assignment2/src/main/
Repository link (alternative):
https://drive.google.com/drive/folders/1tBsoJdGamzs8pe4WCymPNPWQmwSZ_8FC?usp=sharing
Note: if the project in the first repository link does not compile (compiles only on some machines) use the project at this alternative link which contains two projects `assignment2` and `custom_msgs`!
Google Drive link to video:
https://drive.google.com/file/d/1_ggFq9QACcfdPw1MeClhebJPWb2J0l9c/view?usp=sharing
Note: we have implemented the extra points part of the assignment!

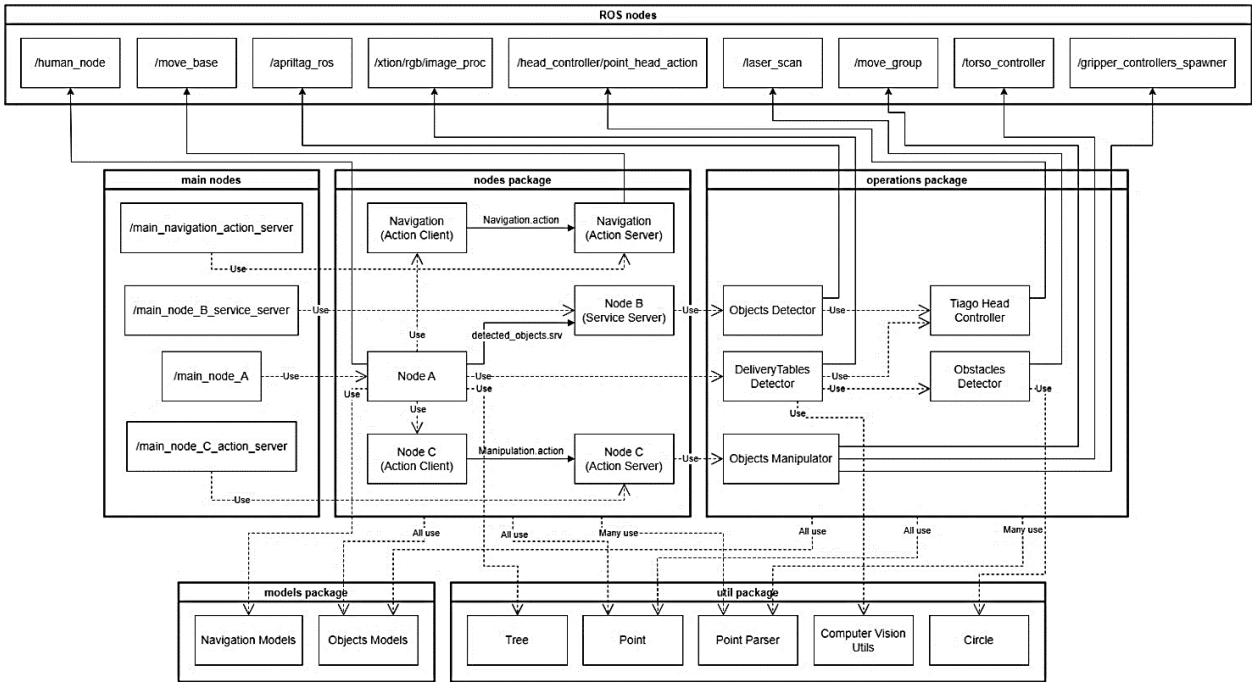
Report Assignment 2 of Intelligent Robotics

How to run the program

Launch file: `roslaunch assignment2 pickup_delivery_routine.launch`
Note: to enable/disable the extra points part (enabled by default) set to true/false the input parameter `extra_points_flag` of Node A in the launch file.

Structure of the project

Conceptual UML diagram of the project



Project structure and files organization

Main files: these files just define the main files of each node by instantiating the corresponding node class defined in the `nodes` package:

- **`main_node_A.cpp`:** contains the main of the Node A. This file receives the `extra_points_flag` in input from the user to enable/disable the extra points part and instantiates the `NodeA`.
- **`main_node_B_service_server.cpp`:** contains the main of the Node B. This file instantiates the `NodeBServiceServer`.
- **`main_node_C_action_server.cpp`:** contains the main of the Node C. This file instantiates the `NodeCActionServer`.
- **`main_node_navigation_action_server.cpp`:** (adapted from assignment 1) contains the main of the node navigation. This file instantiates the `NavigationActionServer`.

Package `nodes`: these files define the nodes of the project and their communication logic, where each node executes the corresponding operations defined in the `operations` package:

- **`NodeA.h/.cpp`:** this file implements a node which defines the navigation logic using a tree data structure (see section **Navigation logic using tree data structure**), handles the program main execution workflow, and behave as a "proxy" with other nodes as follows:
  - Acts as a service client for `human_node` service server.
  - Uses `NavigationActionClient` to communicate with the `NavigationActionServer`.
  - Acts as a service client for `NodeBServiceServer`.
  - Uses `NodeCActionClient` to communicate with the `NodeCActionServer`.
- **`NodeBServiceServer.h/.cpp`:** this file implements a service server to execute the detection of objects and obstacles on the pick-up table by using `ObjectsDetector`.
  Note: the reason why we implemented a service server for Node B is that we just need to exchange a message once a while with Node A, when it requires it, and we want this communication to be synchronized with Node A workflow.
- **`NodeCActionClient.h/.cpp`:** this file implements an action client which is used from `NodeA` to communicate with the `NodeCActionServer`.
- **`NodeCActionServer.h/.cpp`:** this file implements an action server to execute the manipulation (pick-up or place) of the target object by using `ObjectsManipulator`.
  Note: the reason why we implemented an action server for Node C is that we want Node A to define a goal for the robot's arm (target object + detected objects and obstacles), once a while and in a synchronous way, and we want to receive feedback and result about the state of execution of the robot's arm.
- **`NavigationActionClient.h/.cpp`:** (adapted from assignment 1) this file implements an action client which is used from `NodeA` to communicate with the `NavigationActionServer`.
- **`NavigationActionServer.h/.cpp`:** (adapted from assignment 1) this file implements an action server which allows the robot to navigate for reaching a specific target pose.

Note: the reason why we divided operations and nodes is that, in this way, we are able to decouple the part of the definition and of the messages exchange logic of each node, from the operations executed within each node.

Package `operations`: these files define the operations that are executed by nodes:

- **`ObstaclesDetector.h/.cpp`:** (adapted from assignment 1) this file allows to detect the central positions of the obstacles in the map by using the robot's laser scan.
- **`TiagoHeadController.h/.cpp`:** this file allows to incline the robot's head by setting the desired pitch angle.

- **`ObjectsDetector.h/.cpp`:** this file allows to detect the positions of the objects and of the obstacles on the pick-up table. For all the details about how it works, see the section **Objects detection**.
- **`ObjectsManipulator.h/.cpp`:** this file allows to pick-up the objects from the pick-up table and to place the objects on the delivery table by using the robot's arm. For all the details about how it works, see the section **Objects manipulation**.
- **`ObjectsManipulator.h/.cpp`:** this file implements the extra points part for detecting the position of the delivery tables by using laser scan + camera. For all the details about how it works, see the section **Extra points part**.

Package `models`: these files provide a representation for each entity defined by the assignment, in particular they are containers which allow to conveniently store, handle and exchange information between different parts of the project:

- **`ObjectsModels.h/.cpp`:** this file provides containers for storing, handling, linking together and to exchange information concerning the entities of this assignment (e.g., pick-up object, pick-up obstacle, delivery table, etc.).
- **`NavigationModels.h/.cpp`:** this file provides containers for storing, handling and exchanging information concerning the navigation steps/poses.

Package `util`: these files provide some general useful functionality:

- **`Point.h/.cpp`:** contains some util structs which represent a polar point and a cartesian point, plus some helper functions (from assignment 1), and some new util structs which represent a position and a pose, plus some helper functions.
- **`Circle.h/.cpp`:** (from assignment 1) contains an util struct which represents a circle, plus some helper functions.
- **`PointParser.h/.cpp`:** contains a set of conversion functions which allow to extract position/pose information from a certain kind of message and to build a certain kind of message starting from position/pose information.
- **`Tree.h/.cpp`:** contains a generic tree data structure class, plus some helper functions which implement the depth first search exploration algorithm.
- **`ComputerVisionUtils.h/.cpp`:** contains a set of util functions for computer vision, imported from the computer vision final project we wrote last year.

Note: sources and header (C++) files are respectively organized in the folders `src` and `include`.

Action files:

- **`Navigation.action`:** (from assignment 1)
  - **Goal:**
    - `std_msgs/Header header`
    - `geometry_msgs/Pose target_pose`
  - **Result:**
    - `std_msgs/Header header`
    - `geometry_msgs/Pose[] obstacle_positions`
  - **Feedback:**
    - `std_msgs/Header header`
    - `string status`

- **`ObjectManipulation.action`**:
  - **Goal:**
    - `std_msgs/Header header`
    - `bool modality # 1: pick-up, 0: place`
    - `assignment2/PickUpObject target_object`
    - `assignment2/PickUpObject[] objects`
    - `assignment2/PickUpObstacle[] obstacles`
    -
  - **Result:**
    - `std_msgs/Header header`
    - `bool is_succeeded`
  - **Feedback:**
    - `std_msgs/Header header`
    - `string status`

Service files:

- **`detected_objects.srv`:**
  - **Request:**
    - `std_msgs/Header header`
    - `bool is_ready`
  - **Response:**
    - `std_msgs/Header header`
    - `assignment2/PickUpObject[] objects`
    - `assignment2/PickUpObstacle[] obstacles`

Custom messages:

- **`PickUpObject.msg`:**
  - `int32 object_id`
  - `geometry_msgs/Pose object_position`
- **`PickUpObstacle.msg`:**
  - `int32 obstacle_id`
  - `geometry_msgs/Pose obstacle_position`

Launch file: `pickup_delivery_routine.launch` runs all the nodes of this project and **creates a new window for showing only the execution status of Node A**.

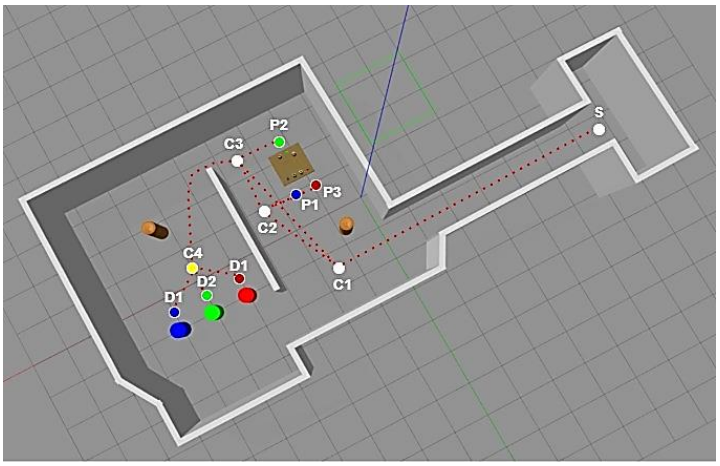
Approaches adopted

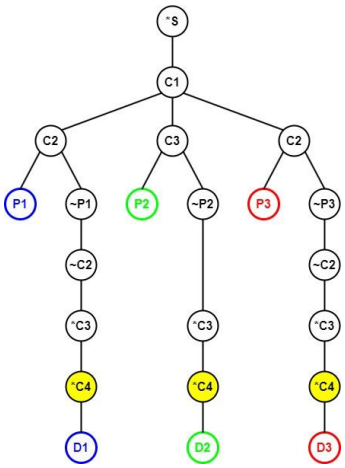
Navigation logic using tree data structure

The navigation logic is implemented in `NodeA`, and is here described:

- **Build of the navigation tree:** the navigation path is created by using a tree data structure, where branches are built automatically given the order of the objects to pick-up.
- **Execution of the navigation tree:** the nodes of the navigation tree path are visited iteratively with depth first search (DFS) algorithm. For moving from one node to another is used the `NavigationActionServer` of assignment 1. In some visited nodes are executed some routine operations by means of a callback function.
- **Information stored in nodes:** each node contains the following information (from `NavigationModels`):
  - The pose the robot has to assume in that node.
  - The pose type: crossing pose, pick-up pose or delivery pose.

- The target object to pick-up/deliver.
- A flag which indicates whether execute the extra points part.





In the figure above is represented an example of possible navigation tree path that can be generated by the program, where is important to notice that:

- For nodes represented as $*N_i$ both the position and the orientation are prefixed.
- For nodes represented as N_i the position is prefixed, whereas the orientation is computed as $\text{angle}(N_i, N_{i+1})$, this allows to obtain a smoother navigation path.
- For nodes represented as $\sim N_i$ the position is the same as for N_i, whereas the orientation is different and is computed analogously as $\text{angle}(N_i, N_{i+1})$.
- Black nodes are crossing poses, hence no routine operation is executed there.
- Blue, green and red nodes P_i are prefixed pick-up poses where the pick-up routine is executed; whereas blue, green and red nodes D_i are delivery poses (prefixed only in the case in which the extra points part is disabled) where the delivery routine is executed.
- Yellow nodes are the prefixed poses where, if enabled, is executed the extra points routine; in that case the poses D_i's are computed by `DeliveryTablesDetector` which implements the extra points logic using laser scan + camera (see section **Extra points part**).

Objects detection

The objects detection logic with AprilTag is implemented in the `ObjectsDetector` class, which is initialized also with Tiago's head movement. In particular, the head joints are moved to a specified position (0 for joint 1, -0.65 for joint 2).

Detection Function (detect_objects_on_pick_up_table):

- It waits for a message on the "`/tag_detections`" topic using `ros::topic::waitForMessage`.
- For each detected object in the message, the object's pose is transformed from the camera reference frame to the robot's base link frame, using the `tf` transform, and is performed a creation of a `PickUpObject` or a `PickUpObstacle` according to the ID.
- Detected objects and obstacles are stored in vectors (objects and obstacles), with also their position. Those vectors are then used by other functions.

For this part, we did not implement a standard callback function, since we had synchronization problems with the rest of the processing logic. We instead opted for a simple message waiting, that allows us also to simplify the management of the same object being detected multiple times.

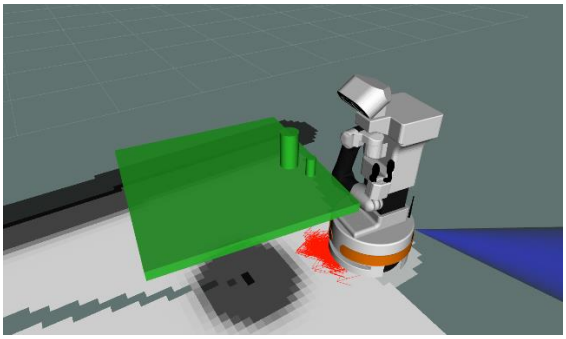
Objects manipulation

The object manipulation logic for Tiago implements pick-and-place operations using `MoveIt!` library. The primary goal is to handle the picking up and placing of various objects using Tiago's robotic arm and gripper. The code is structured into two main functions: `pick_up_target_object` and `place_target_object`. These functions organize the sequence of movements, gripper actions, and collision handling required for the successful object manipulation.

Picking Up Target Object:

The `pick_up_target_object` function initiates the object manipulation sequence.
Key steps include:

- **Setting Home, Safe, Above, Pick and Pick Positions:** Positions for home (the current pose at the beginning of the routine), safe, above pick, and pick configurations are defined based on the type of target object detected. These positions ensure proper alignment for the robotic arm during the pick-up operation, we try to pick all the target object from the side and from above but the second option seams to give better result.
- **Creating Collision Objects:** The function generates collision objects in the environment based on detected objects and obstacles. These objects are crucial for collision avoidance during the arm's movement. For each object found within the vector **`object_detected`** and **`obstacle_detected`**, create a collision object in the shape of a cylinder for the blue hexagon and for the obstacles, and in the shape of a cube for the cube and triangle. Subsequently, create a collision object for the table with only the supporting surface, appropriately thickened, so that Tiago's arm can pass underneath it during movement. All object positions are calculated relative to Tiago's **`base_footprint`** reference frame.



- **Arm Movement and Gripper Actions:** The robotic arm moves the safe position, then to above position where the collision object related to the target object is removed, and then process continues towards the pick position. The gripper is closed, and the target object is virtually attached to the gripper using `attachObjects` implement with **`gazebo_ros_link_attacher`**. A note about the ROS link attacher: we made use of the attacher functionality, since Tiago cannot always grasp the object in a proper way to transport it (this is particularly true for the green triangle object). However, the attacher module seems not to be working well in all the cases, in particular for the detach. As shown in the video, sometimes the object is no longer able to being dropped, even with the detach function.
- **Finalizing and Removing Collision Objects:** After successful pick-up, the arm returns to above pick position, then go to safe position the home position ready to carry the object to

the place position without collision during the navigation task. Before the end all the collision objects are cleaned from the scene.

Placing Target Object:
The place_target_object function manages the placing operation. Key steps include:

- **Defining Home, Safe_Place and Place Positions:** Positions for home, safe_place, above_place and place position are determined based on the type of target object.
- **Creating Collision Objects:** A collision object representing the place table is generated, whit the shape of a cylinder and added to the environment in position calculated relative to Tiago's `**base_footprint**` reference frame.
- **Arm Movement and Gripper Actions:** The arm moves to the safe place position, then to the `above_place_position` and final place position. The gripper is opened, and the target object is detached using `detachObjects`.
- **Finalizing and Removing Collision Objects:** After successful placement, the place table collision object is removed, and the arm returns to the home position.

For all movements of the robotic arm, `**move_arm(goal_pose)**` is utilized.
The `move_arm` function plans and executes the movement of the robotic arm to a desired position using the `MoveIt!` framework. Planning is performed using the planning group `group_arm_torso`, which includes both the arm and torso of the robot.
Key Steps:

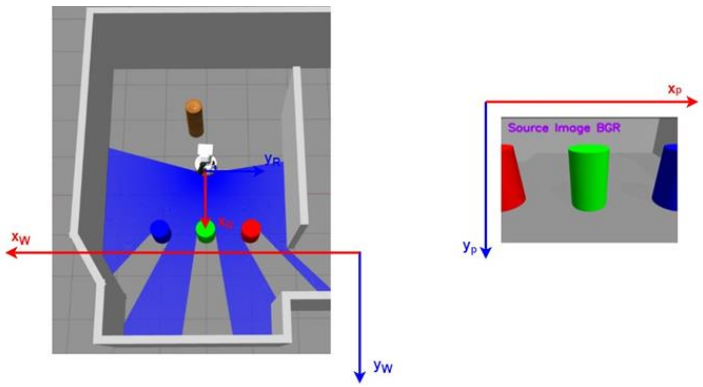
- Configuration of the planning group by setting the reference frame of the pose and the target pose.
- Movement planning using the plan method of the planning group with a maximum planning time of 5.0 seconds.
- Setting a goal pose tolerance of 0.05.
- Execution of the planned motion using the move method and wait to movement to be finished.

Printing planning information and the duration of the motion.

Extra points part

The extra points routine is implemented in `DeliveryTablesDetector` and is executed (if enabled) when the robot reaches the pose in front of the delivery tables (as represented in the figure below):

- **Delivery tables position detection:**



  - The central position of each delivery table detected in front of the robot, in the **robot reference frame**, is obtained by `ObstaclesDetector` (from assignment 1), which uses laser scan.

- The central position of each delivery table is converted from cartesian coordinates to polar coordinates:

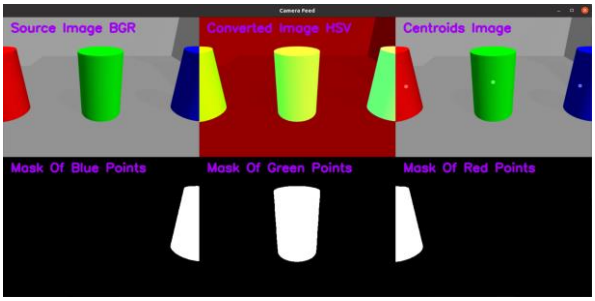
$$P_{R,\text{table}} = (x_{R,\text{table}}, y_{R,\text{table}}) \quad \rightarrow \quad P_{R,\text{table}} = (\rho_{R,\text{table}}, \theta_{R,\text{table}})$$

  in order to obtain a translation vector w.r.t. the current pose of the robot in the world reference frame $P_{W,\text{robot}} = (x_{W,\text{robot}}, y_{W,\text{robot}}, \theta_{W,\text{robot}})$.
- Then, the pose the robot has to assume in front of each delivery table in the **world reference frame** is obtained as:

$$P_{W,\text{pose}} = \begin{cases} x_{W,\text{pose}} = x_{W,\text{robot}} + (\rho_{R,\text{table}} - C) \cdot \sin(\theta_{R,\text{table}}) \\ y_{W,\text{pose}} = y_{W,\text{robot}} + (\rho_{R,\text{table}} - C) \cdot \cos(\theta_{R,\text{table}}) \\ \theta_{W,\text{pose}} = -\left(\frac{\theta_{R,\text{table}}}{D} \cdot \frac{180}{\pi}\right) + 90 \end{cases}$$

  where C is a constant offset which represents the distance between the robot and the center of the delivery table, D is a correction factor for the (yaw) orientation of the robot, and the orientation angle is expressed in degrees ($+90$ to convert from robot reference frame to world reference frame).
- These poses are stored in a vector `detected_delivery_tables` and are sorted in ascending order depending on their position along the x_W-axis.

- **Delivery tables color association:**



  At this point we have the positions of the delivery tables (expressed as poses the robot has to assume in front of each of them), but their colors are still unknown.
  The procedure for associating the color to each delivery table is the following:
  - The robot's camera is used to obtain an image of the obstacles in front of it.
  - The image is converted in HSV format (Hue Saturation Brightness), which is useful for easily filter the different colors in the image with thresholding.
  - By means of the thresholding, each color: blue, green, red, is filtered in three separated images, in this way we obtain three masks, each one containing as foreground pixels the points of one colored delivery table.
  - For each mask, we compute the centroid $(x_{p,\text{center}}, y_{p,\text{center}})$ of the distribution of pixels, in the **image pixel reference frame**.
  - Each centroid, associated with its corresponding color, is stored in a vector `color_centroids`, and are sorted in ascending order depending on their position along the x_p-axis (note: x_p-axis is mirrored w.r.t. the x_W-axis, but the two axes have opposite orientation).
  - Finally, for mapping pose-color of each delivery table, we simply map one-to-one: `detected_delivery_tables[i]` to `color_centroids[i]`.
- **Correct delivery table selection:** `detected_delivery_tables` is a vector of `DeliveryTable`, from `ObjectsModels`, hence given the color of the `target_object` (of type `PickUpObject` ,

from `ObjectsModels`), we just need to iterate the vector for fining the corresponding delivery table with the same color as `target_object`.

Members contribution to the project

Osti Simone:

- Fully designed, implemented and written report about the arm movement with MoveIt!.
- Fully designed, implemented and written report for add and remove collision object.
- Testing poses and trajectories for pick and place object.
- Modify of assignment2 and creation of pkg custom_msgs for importing msg, srv and action

Russo Christian Francesco:

- Designed, implemented and written report about the general structure of the program.
- Fully designed, implemented and written report about nodes A, B, C definition and communication with messages, services, and actions files.
- Fully designed, implemented and written report about the robot navigation logic part (with tree data structure).
- Fully designed, implemented and written report about the extra points part (laser scan + camera).
- Designed, implemented and written report about:
    - All files in `nodes` package (and relative main files).
    - The following files in `operations` package: `ObstaclesDetector` (adjusted from assignment 1), `DeliveryTablesDetector`.
    - All files in `models` package.
    - All files in `util` package.
    - Launch file.

Spinato Matteo:

- Fully designed, implemented and written report about the part of detection logic with AprilTag.
- Fully designed and implemented the head movement rationale.
- Fully designed and implemented the attach logic from gazebo_ros_link_attacher.
- Other contribution on the manipulation part, in particular the way the objects are grasped, the gripper movements and testing.