Robotics Project of the course
EDAP20: Intelligent Autonomous Systems

Robotics and Semantic Systems Group, Lund University

September 2021
Revision 1.1

Contents

1 Introduction

Imagine an autonomous robot that is supposed to fetch and bring parts, like the care-o-bot bringing bread and butter from the fridge. In this project, we will simplify the problem. You will program our mobile platform such that it is able to fetch an object at some location A and bring it to some location B. The objects will have to be picked and placed. The robot will have to know where the locations are and how to get there. You will have to program the robot to do this.

In detail, you will be working on our robot HERON. The Heron robot should navigate in the LUCAS corridor (4th floor E-building) and collect small toy blocks from some locations and then place/stack them at a goal location. The map of the LUCAS corridor generated by the Heron robot in Gazebo is shown in the Figure 1. The locations of the boxes and the goal platform demonstrated in the Figure 1 are only tentative. Your respective TA will help you finalize those locations.

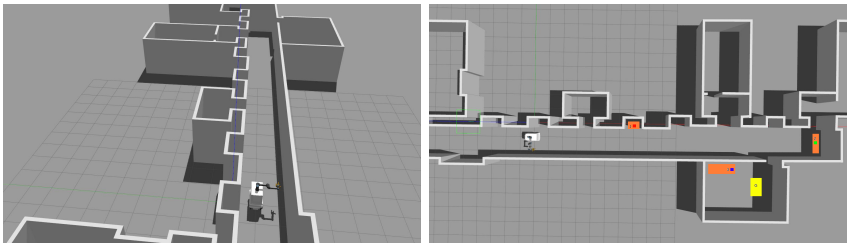

Figure 1: The location of the blocks are shown on the platforms 1,2 and 3. The yellow table shown in the image is the sample goal location where the robot has to place/stack the objects.

To complete the task the robot will need

- a *navigation* module that tackles the problem of finding its way in the environment without crushing into things.
- a *manipulation* module that allows the robot to pickup objects and place or stack them on a table.
- a *perception* module to help the robot to see the object it should pick.
- a *reasoning* module to coordinate the first three modules to achieve the goal.

With the help of our lectures and the ROS course you will be able to program these modules.

But be aware: the modules must also function together to achieve the fetch&bring task.

Each of the modules are not complicated but trying them out on a real robot will provide some additional challenges, fun and experience.

You will work in teams of 15 students, 5 students for each of the first three modules (navigation, manipulation, perception). To support the collaboration between the groups and their modules, we provide for each module a ROS API. Please talk to the TAs before extending the APIs. The reasoning module shall be coordinated by the navigation group but must be realized through a collaboration between all three groups.

Every group member is expected to contribute equally in the project.

1.1 Getting you started: Robot Setup

For this project you will use our robot **Heron** (see Fig 2), which has all the necessary capabilities to accomplish the task. Heron is made up of these parts:

- *Mobile Industrial Robots MIR200* - a mobile platform
- *Universal Robots UR5e* - a robot arm
- *Intel Realsense D435* - An RGB-D camera with a resolution of 1920*1080 pixels for the RGB camera and an output depth resolution of 1280*720; attached to the wrist of the arm. The minimum depth distance is 0.105 m.
- *Schunk WSG-50 parallel gripper* - a two finger gripper attached the *UR*
- RobotMind2 - Inside the casing, Heron has a powerful workstation that can be used to run the ROS component needed to use the robot. We refer to this workstation as RobotMind2.
- On board WiFi - Heron has it's own WiFi setup that you will use in order to connect everything through the ROS master process.

Installing the Setup:

You can find the *heron workspace setup* on the Gitlab server. You can follow those instructions for an installation on a Linux computer, e.g., the computer that you will use to control the robot.

Using the robot

Your respective TA will show you how to power on each of the components in the Heron robot. When everything is powered on, you will connect to its on board WiFi:
ssid: `Heron Mobile WiFi 5G`
password: `herongogo`

As soon as you're connected to the WiFi, you'll be able to use the MiR web interface (your TA will show you), and you'll be able to use ROS just like you

Figure 2: Heron in manual (joystick) driving mode with status blue color.

learned in `ROS Basics in 5 Days`.

You'll also be able to connect to the RobotMind2 workstation inside Heron. While it's entirely possible to run all the ROS components on your own computer, we highly recommend using RobotMind2 for the `roscore` and `bringup` (more on these later). Connecting to RobotMind2 is as simple as:

```
$ ssh ias_student@192.168.0.8
```

You will be asked to enter `ias_student`'s password, which is `ias2021`.

If you've never used `ssh` before, you may wonder what just happened. You've logged in remotely to the RobotMind2 computer, as the user `ias_student`, and you're given a terminal session *on RobotMind2*. In this terminal session, you can type your commands just like you would on you own computer.

1.2 The Big Picture

The overall structure of the project is depicted in Figure 3. The manipulation and perception teams are providing information and the capabilities to the navigation + integration team. The navigation team is then responsible for the overall integration of the project. For instance, if the task is to collect blocks, then it is the responsibility of the navigation team to call the required *action servers* from the various modules.

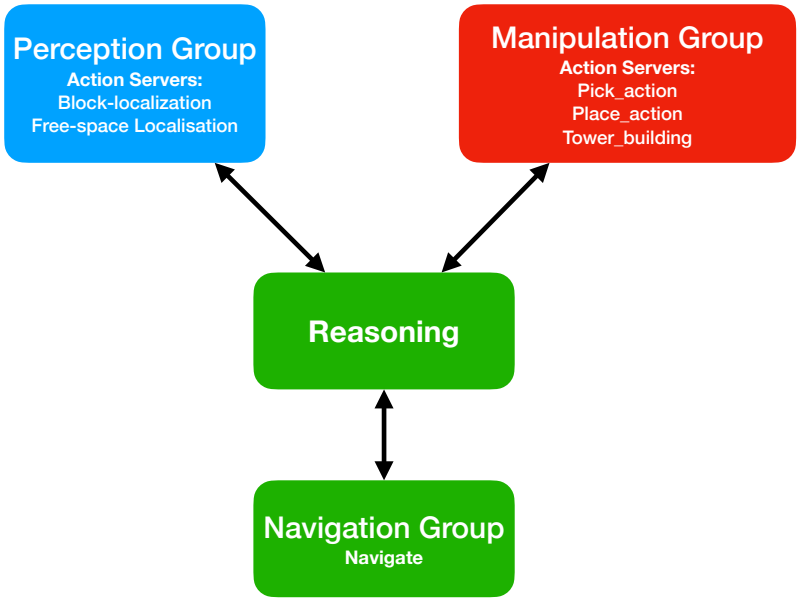

Figure 3: Perception, manipulation and navigation actions are coordinated by the Reasoning process. As indicated by the block color, the Navigation team will coordinate the development of the Reasoning process, but through close collaboration with the other two teams.

A proto-typical but over-simplified sequence of actions in a fetch and bring task could be:

1. Trigger drive to location 1
2. Trigger perception
3. Trigger picking
4. Trigger drive to goal location G
5. Trigger detection of free space
6. Trigger placing

It is over-simplified because in most situations, something will be going wrong, and the reasoning block will not only have to trigger the actions but also show some reasoning capabilities if something goes not according to expectation.

1.3 Code Management

You will find repositories on a *Gitlab* server hosted by the CS department. Here you find:

1. all the necessary code to run the robot
   https://coursegit.cs.lth.se/ias-course/heron

2. the message definitions for this course - *ias_msgs*
   https://coursegit.cs.lth.se/ias-course/heron/ias_msgs

3. repositories for the team set A and B
   https://coursegit.cs.lth.se/ias-course/groups

You will not need to push to the repositories of 1) and 2), but feel free to send merge requests if you find something that you want to see improved.

The code of your individual team will be stored in your team's repository in 3). This is especially important for the integration of the project, but also for TAs to be able to help you. You will need to be able to push and pull from this repository. If you are not familiar with Git and Gitlab this is the time to learn about these important tools for software engineering. An introduction on how to commit code in Git can be found here.

We can only add you to the respective groups in the Gitlab server after you signed in by using the Lund University credentials at least once. You can do this by going to the server and clicking at "Lund University Login" button at the lower-right corner of the login page. Let us know when you completed this.

2 Navigation + Reasoning

There are two main goals:

- Equip the robot with the capability of collision free navigation in the environment.
- Provide the robot with a *brain*, i.e., the reasoning capability that coordinates all the modules to achieve the goal of this project.

2.1 Navigation

In the navigation module, you would use the knowledge gained in the course *ROS Navigation in 5 days*. The **Navigation Stack** is a readily available collection of programs that allows the robot to move in an environment while avoiding obstacles. It takes **odometry** and **sensor data** as input and sends **velocity commands** as output to the mobile base. The Navigation Stack requires ROS, tf transform tree and correct message types publishing sensor data as pre-requisites.

The first thing a robot requires is a **map**. The map allows the robot to localize itself and also provides it with location information of objects in an environment. For this module, a map of 4th floor of the E-building will be provided to you.

As you know, there three ways to communicate in ROS: topics, services and actions.By far the most common way is to use topics. You can send direct motor commands to the robot by publishing a motor command messages on the topic */cmd_vel*. Generally, these messages are handled directly by the Navigation Stack.

Once the Navigation Stack is set up the next step is to have a system in place that tells the robot *where* to go and *how* to go there. This is done by performing *Path planning*. Path planning basically takes as input the current location of the robot and the location where the robot should go. As output it gives the best and fastest path.

2.1.1 Steps

Necessary commands to get the robot up and running are. Shell 1:

```
$ roscore
```

This would start the roscore.

Shell 2:

```
$ roslaunch heron_robot base_bringup.launch sim:=true
```

Alternatively, if you want to work with the real robot run the following command:

```
$ roslaunch heron_robot base_bringup.launch sim:=false
```

The above command launches *gazebo* and spawns Heron in the 4th floor of E-building environment Fig 1. You can use *rviz* to send the movement commands to the robot. To do that, run in Shell 3:

```
$ rviz -d $(rospack find mir_navigation)/rviz/navigation.rviz
```

You can manually *localize* the robot by using the *2D Pose Estimate* button at the top: To estimate the pose you must click on the robot to set the initial position and then drag the mouse into the viewing direction of the robot.

You can also send a goal to the robot in Rviz by using the *2D Nav Goal* button: Just select a goal point and drag the mouse in the direction you want the robot to be facing at the goal location. This sends the goal location to the path planner that then automatically finds a path for the robot to follow while avoiding obstacles. You can see the robot moving in gazebo and rviz.

You can also send goal manually to the robot by publishing on topic

```
    /move_base_simple/goal
```

This is the topic to which *2D Nav Goal* button sends the goal vector. Before sending goals directly please refer to actionlib or the tutorials on action servers for more information. Basically, you have to start the action client "move_base" which accepts the message type "move_base_msgs.msg.MoveBaseAction". You can find a simple example script at send_mir_goal.py.

You can find the base pose ground truth of the robot using the following topic

```
    rostopic echo -n1 /base_pose_ground_truth
```

Please check the rostopic list: you will find many potentially useful topics which may proof useful for your project.

2.2 The Reasoning Modules: The System's Brain

The way you do it is by calling the necessary actions servers introduced by the other modules whenever required. Please refer to Fig 3 for the big picture. You also need to be aware of the individual integration specifications of the other modules. Since the navigation team is also the integrating team, we are not providing a dedicated API for navigation.

The simplest solution to program such a task is to simply run one skill after another, similarly to normal desktop computer programs that execute one function after another. On robots, however, failures can easily happen: The robot might not reach the drive-to location, the picking might fail, etc. If every command has a success probability of, let's say 0.95, the overall success probability for collecting the above three objects using the above 18 commands is as small as $0.95^{18} \approx 0.4$. In the lecture on reasoning and knowledge representation, we have provided you with some techniques on how to handle such situations systematically. Keywords are *finite-state-machine*, *behaviour trees* and planners.

Another aspect is the reliability of the different modules: Do they function as expected? Are you sure the messages contain the information you expect from a particular topic? It is important that definitions are agreed with the

other teams. Also, it is very important to discuss what kind of functionalities a module could ideally provide.

3 Perception

The perception module provides vision capabilities to the robot. It allows the robot to find the location of the blocks for picking. This module will generally only interact with the Manipulation module.

In the perception team you will create action servers to provide the necessary capabilities of *block localization*. The Reasoning module will call these action servers whenever needed.

If you want to use the camera with ROS code on HERON, execute the following commands.

RobotMind2:

```
$ roscore &
$ roslaunch heron_robot heron_bringup.launch\
    realsense_static_tf_publisher:=true sim:=false
```

This starts all components of the robot. Technically you could start only the arm and the camera, but it is much more convenient to be able to move the robot around. That way you can move the robot to e.g. a table with blocks, and then execute your perception logic.

Since the entire robot is started, you will be able to see very many different ros topics. The ones related to the camera start with `/realsense`. Specifically you should focus on:

- `/realsense/rgb/image_raw`
  This topic will provide you with plain RGB images.
- `/realsense/aligned_depth_to_color/image_raw`
  This topic will provide you with depth images that have been aligned to match the RGB images.

If you want, you can visualize the entire robot, including the camera views by running:
Shell 1:

```
$ roslaunch heron_robot rviz.launch config:=heron
```

To run the camera in simulation, execute the following commands:
Shell 1:

```
$ roscore
```

Shell 2:

```
$ roslaunch heron_robot arm_bringup.launch \
realsense_static_tf_publisher:=true sim:=true
```

If you want to start working on perception before getting access to the real Heron robot, by far the easiest way is to use the static_image_publisher that we provide.

It is entirely possible to run the perception module in simulation as described above. However, due to its simplicity, we recommend the `static_image_publisher`

instead.

`static_image_publisher` is a small package that simply publishes a static RGB + Depth image in the correct topics. That way, you can launch the `static_image_publisher.py` script according to the instructions in the repo, and then program your perception logic just like you would if you were using the real Heron robot.

The image being published by `static_image_publisher` was originally taken from the real Heron robot in the Robotlab, so it is a very real test-case for you. It also includes several blocks of different colors, so it should provide a good basis for you to implement your block detection logic.

3.1 Block Localization

The primary goal of the Perception module is to detect a block and return its pose in 3D space, so that the arm can move to the block and pick it up.

You will use the wrist camera of the arm to detect the blocks. This means that the first thing the arms needs to do is to move into a configuration that brings the blocks into the field of view (FOV) of the camera. This arm movement will be handled by the manipulation group. Once a block is detected in the RGB image, its location in the camera coordinate system needs to be determined using the depth data. This location is then transformed to the arm's coordinate frame and returned as a result of the action.

ROS' tf package provides the class `tf.TransformListener`, which is helpful when transforming a coordinate into a different frame. Look especially at `tf.Transformer.lookupTransform()` and `tf.TransformerRos.transformPose()`. These functions are both available in the `tf.TransformListener` class.

The blocks used for this project look similar to Fig 4. There are several ways you can detect a block. An easy approach to detect the block is to move the camera above the table and let the camera look straight down onto the table. You will easily be able to locate the block using color segmentation. Naturally you may use any method you want, but color segmentation is most likely the easiest.

Figure 4: Example blocks.

3.1.1 Integration

The block localization runs as an action server following the conventions listed below. Whenever a goal is sent to this action server, it would return the **pose of one or more blocks** as a result of the action. You might want to consider how the `BlockLocalization` action might react if there are multiple blocks visible in the camera? Alternatively, it might also be OK to assume that only one block will be visible at a time.

- Type: Simple Action Server
- Topic: */block_localization*
- Message type: *ias_msgs/BlockLocalization.action*

```
### goal definition
---
### result definition
geometry_msgs/PoseArray poses
---
### feedback
# can be used as an optional progress indicator
# with values between 0 and 100
int32[] progress
```

Listing 1: Definition of the *BlockLocalization* action. The goal can be empty since a trigger suffices and the result is an array of possibly multiple poses.

3.2 The Depth Image

While the RGB image is nothing more than a plain-old-school RGB image, the Depth image is different from what you might have seen before.

When you subscribe to the `/realsense/aligned_depth_to_color/image_raw` topic (and run it through cv_bridge), you'll get an OpenCV image.

The first thing you might try is to run

```
cv2.imshow('depth image', depth_image)
cv2.waitKey(0)
```

in order to look at the image.

However, you'll notice that visualizing the image with `imshow()` didn't work very well. So instead you try printing some information about it:

```
print depth_image.shape, depth_image.dtype
```

Here, you'll see that the shape matches the size of the RGB image, so far so good. But the data type returns `uint16`. What? Shouldn't it be `uint8`?

So it turns out that the Depth image is made up of values from 0-65535, and not 0-255 as you might expect. This is to accommodate a distance directly in the image data.

Let's finally get to what you need to know: Each pixel value in the Depth image is the distance from the camera in *millimeters*, which means that a pixel value of 548 indicates that something is 548mm (or 54.8cm, or 0.548m) away from the camera.

The value 0 is special and means that the camera failed to detect a distance, which usually means that the object there is either too close or too far away.

If you want to visualize the Depth image, you'll have more luck with:

```
scale = 255.0 / np.max(depth_image)
depth_image_scaled = cv2.convertScaleAbs(depth_image,
                         alpha=scale)
depth_image_colormap = cv2.applyColorMap(depth_image_scaled,
                         cv2.COLORMAP_JET)
cv2.imshow('depth image', depth_image_colormap)
cv2.waitKey(0)
```

3.3 Calibration

Before you attempt to compute coordinates in the world frame you need to know how to transform image pixel coordinates into physical distances. Immediately, you may think that you need to perform a full camera calibration. However, that is not necessary in this project, and some simple measurements will do! Keyword: *Orthographic Projection*. Your TA will help you with this in the lab.

3.4 Grasping the Block

When running your code on the real Heron robot, you may find that you don't get an exact 3D position on the first try, especially when the block is closer to the edge of the image. As such, it is wise to coordinate with the Manipulation group to create a method that can move closer to the block in several steps (i.e. in a loop).

Hint: the translation to 3D coordinates will generally be most exact when the block is centered in the image.

Another hint: Remember the video from the class.

3.5 Table Detection

An *optional* extension to the perception module is a tabletop detector. At some point, the Heron robot will have to put the block somewhere. While it is OK to simply hard-code the arm position used when releasing the block, eager students are welcome to extend the perception module by dynamically detecting a table surface where the block can be placed.

4 Manipulation

Manipulation module equips the robot with the capability to manipulate blocks in the environment by automatically generating arm movement sequences. The simplest form of manipulation would be *picking* and *placing* the blocks on the table. To be able to generate those movement sequences, we use motion planning. As in the ConstructSim, we will be using MoveIt.

As you know, MoveIt is a ready made set of packages and tools that allow you to perform manipulation with ROS. MoveIt provides software and tools in order to do Motion Planning, Manipulation, Perception, Kinematics, Collision Checking, and Control. We have set up the launch files for Heron robot to launch Rviz and Gazebo simulator.

You can run the launch file using the command as follows:

Shell 1 (First run the ros master):

```
$ roscore
```

Shell 2 (Launch the Heron Simulation):

```
$ roslaunch heron_robot heron_bringup.launch
```

To run the launch file on the real robot, you would run

```
$ roslaunch heron_robot heron_bringup.launch sim:=false
```

This command will start Rviz and the Gazebo window (containing the Heron robot model). In Rviz, you can play with the robot by giving different goal poses and then plan with the different motion algorithms (RRT*, PRM e.t.c.) provided by the OMPL library as shown in Figure 5.

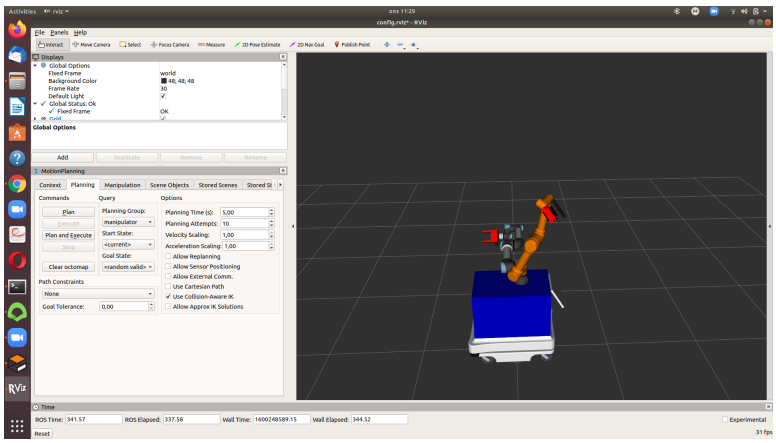

Figure 5: Using OMPL motion planning library to plan for a trajectory from an initial pose to the goal pose and simulating the motion plan in Rviz.

CAUTION (While working on the real robot) :

- Remember that you should always have your hands on the big red (STOP) button in case there is something in the way or anything unexpected happens.
- BEFORE you start moving the arm through the teaching pendant, set the speed to 10
- If you are using Rviz to move the arm then there is a button on the motion planning tab where you can reduce the speed.
- NEVER change the arm speed to a higher velocity.

You can use the following command to get the last joint state configuration.

```
$ rostopic echo -n1 /joint_states
```

Before you do anything, make sure you have done the ROS manipulation course in 5 days.

4.1 Overall Task Description

With your manipulation team you will create action servers that should provide the necessary capabilities of

1. going into a camera overview pose so that the camera can see the block you want to pick
2. picking a block from a table
3. *placing* a block at a specified location.

This module will always be triggered by the Reasoning module.

An optional goal for this module is building a **tower of blocks**. This can be done once all the blocks are collected at the specified rectangular space. The "tower of blocks" is defined to have > 3 **blocks** stacked on top of each other.

4.2 Going to a Camera Overview Pose

The perception group needs the blocks on the table to be in the field of view of the camera. Since the camera is mounted to the wrist of the arm, you need to move the robot arm appropriately.

4.2.1 Integration

This capability should run as an action server with the topic name ***"/goto"***. The goal sent to the action server are the joint states for the camera lookout pose. You need to call the MoveIt commander to plan for the motion.

- Type: Simple Action Server

- Topic: */goto*
- Message type: *ias_msgs/GoTo.action*

```
### goal definition
float32[] joint_states
---
### result definition
---
### feedback
# can be used as an optional progress indicator
# with values between 0 and 100
int32[] progress
```

Listing 2: The definition of *GoTo.action*.

4.3 Picking

The goal of this capability is to allow the robot to pick blocks from a table.

You will use the robotic gripper attached to *UR5e* arm to grasp or pick the block on a table. This means that the arm needs to be moved into a configuration that allows the gripper to pick the block. The gripper is a *parallel gripper* with two parallel fingers that can be closed to pick the block. Once the gripper holds the object you can move the arm to lift the object. Gripper can be opened or closed with a single command.

Tor safety reasons, the robot can navigate *only* if the arm is in its *home position*. Therefore, to be able to let the robot move to a new location, you need to move the arm into its home position.

CAUTION: Be careful when moving the arm when attempting to pick: you may break the fingers of the gripper if the gripper hits the table surface. Make sure your hand is close to the red emergency button and ready to press it once the robot arm moves.

4.3.1 Integration

Picking should run as an action server named ***"/pick"***. The goal sent to the action server is the 3D location of a block on the table. This pose was previously computed by the perception module (BlockLocalization).

- Type: Simple Action Server
- Topic: */pick*
- Message type: *ias_msgs/Pick.action*

```
### goal definition
geometry_msgs/Pose block
---
### result definition
---
### feedback
# can be used as an optional progress indicator
# with values between 0 and 100
int32[] progress
```

Listing 3: Pick.action

4.4 Placing

The goal of this capability is to allow the robot to place blocks on a table.

For the placing action the arm needs to be moved to a configuration that allows the gripper to place the block at the specified location. Once the arm has moved the gripper to the right location the gripper has to be opened to release the block. Once the block is released the arm needs to move back to the home configuration.

CAUTION: Again, make sure your hand is close to the red emergency button and ready to press it once the robot arm moves.

4.4.1 Integration

Placing should be again realized as an action server named ***"/place"***. The goal sent to the action server should be the **location of free space on the table**, as provided by the perception module (FreeSpaceLocalization).

- Type: Simple Action Server
- Topic: */place*
- Message type: *ias_msgs/Place.action*

```
### goal definition
geometry_msgs/PoseArray space
---
### result definition
geometry_msgs/Pose block
---
### feedback
# can be used as an optional progress indicator
# with values between 0 and 100
int32[] progress
```

Listing 4: Place.action