Robotics Project of the course
EDAP20: Intelligent Autonomous Systems

Robotics and Semantic Systems Group, Lund University

September 2022
Revision 1.2

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 them at a goal location. The simulation of the LUCAS corridor with 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.

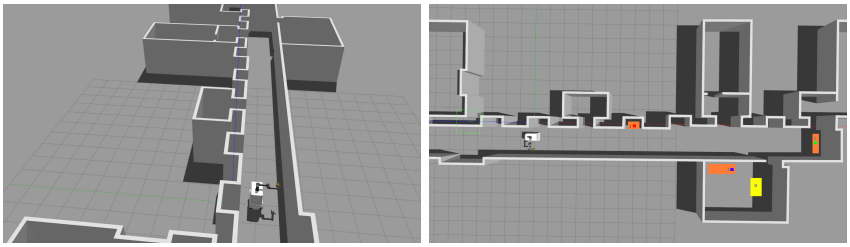

Figure 1: The location of the blocks are shown on the platforms 1,2 and 3. The yellow table shown in the image is the sample goal location where the robot has to place the objects.

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 them on a table and
- a *perception* module to help the robot to see the object it should pick.

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

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 code

skeleton to be filled out. The provided code skeletons are part of a larger framework called *SkiROS2*[1], that will automatically handle coordination of your individual modules.

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 (i.e. the wheels)
- *Universal Robots UR5e* - a robot arm
- *Intel Realsense D435* - An RGB+D camera attached to the arm, with an effective depth range of 0.1-0.8m.
- *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. For simulation on your own computer and E:Alfa use *Heron Container*. The configuration files inside heron workspace can be used with the Heron Container.

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 onboard WiFi:
SSID: `Heron Mobile WiFi 5G`
Password: `herongogo`

There is also another WiFi router in the lab room with a WiFi called `Heron Wifi 5G`. Together they form one network, so other than signal strength it does not matter to which network one is connected.

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 learned in `ROS Basics in 5 Days`.

[1] https://github.com/RVMI/skiros2

Figure 2: Our Heron robot. The blue-colored LED strip inidicated that it is in manual (joystick) driving mode.

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 `ias2022`.

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. One important difference is that it is not possible to launch graphical user interfaces with this ssh command.

1.2 The Big Picture

The overall structure of the project is depicted in Figure 3. Each team will contribute their respective functionalities in a provided code skeleton. These modules will then be able to interact with the reasoning module (SkiROS).

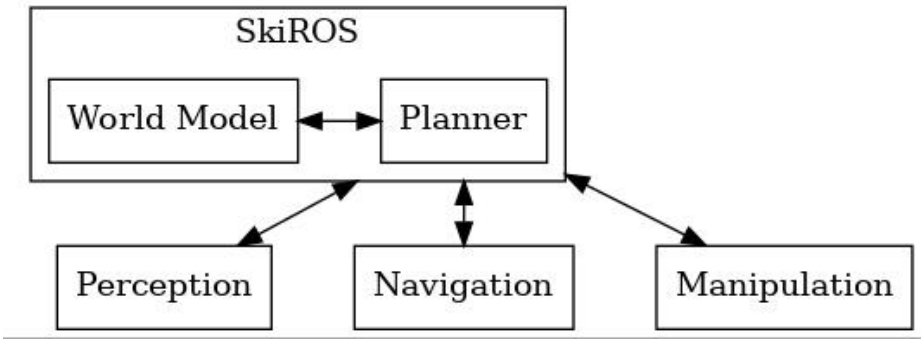

Figure 3: Perception, manipulation and navigation actions are coordinated by SkiROS.

1.2.1 SkiROS2

SkiROS2 is a skill framework for ROS. The principle behind SkiROS is that you should be able to program basic skills that can be composed into larger skills. SkiROS is made up of a skill manager, a world model that has one or more reasonsers and a planning capabilities. These two modules communicate in order to coordinate planning and execution of the skills.

For this project you will implement functionalities in 6 primitive skill:

- Drive skill,
- Block detection skill,
- Free-Space detection skill,
- Skill to move the arm into its home position,
- Skill to move the arm into a lookout position
- and finally a skill to move the arm into a (x, y, z) coordinate.

You will find more details on your specific tasks below.

When you have implemented your tasks, these basic skills will be combined in SkiROS as so:

1. **Drive** to location A,
2. **Move arm to lookout position**,
3. **Block detection**,
4. **Move arm to (x, y, z) coordinate** of the block,
5. **Grasp** the block,
6. **Move arm to home position**,
7. **Drive** to location B,
8. **Move arm to lookout position**,
9. **Free-Space detection**,

10. **Move arm to (x, y, z) coordinate** of the free-space,
11. **Release** the block,
12. **Move arm to home position**.

The **Grasp** and **Release** skills will be provided by us.

While this project does not require you to write any code for communicating with SkiROS or its world model, (this code is available to you in our code skeletons) we strongly encourage you to have a look at our code if you're interested in how it's all connected.

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. Repositories for the teams
   https://coursegit.cs.lth.se/ias-course/groups

You will not need to push to the repositories of 1), 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 2). 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

The primary objective for this module is to provide Heron with capabilities to navigate the environment, while not running into things.

2.1 Navigation

In the navigation module, you will apply what you learned 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.

With the map in place, you will be able to use the `move_base` node, just like you learned in the ConstructSim course.

2.1.1 Running in Simulation

Necessary commands to get the robot up and running are:

```
$ roscore &
$ roslaunch heron_launch simulation.launch
```

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.

You can manually *localize* the robot by using the *2D Pose Estimate* button at the top of RViz: 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.

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

2.1.2 Running on the real System

For starting the setup with the real robot, execute the following commands on RobotMind2:

```
$ roscore &
$ roslaunch heron_launch real.launch
```

3 Perception

The perception module provides vision capabilities to the robot. It allows the robot to find the location of the blocks for picking, and free space for placing.

3.1 Working with the Robot

3.1.1 Working on the real Robot

If you want to use the camera with ROS code on Heron, execute the following commands on RobotMind2:

```
$ roscore &
$ roslaunch heron_launch real.launch
```

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 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 this on the laptop:

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

3.1.2 Working in Simulation

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. `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 perception logic.

3.2 Before You Start - The Depth Image

While the RGB image is nothing more than a plain-old 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.

If you look at the shape of the depth image, you'll see that the *width* and *height* matches the RGB image exactly, but it doesn't have a third dimension (i.e. it is a single-channel image).

Looking at the data type, you'll see that it is uint16 (whereas the RGB image is uint8).

So let's unpack this. The reason for using uint16 (instead of the standard uint8) is to allow distances directly in the image data. 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.

As the topic name suggests, the depth image is *aligned* to the RGB image pixel-by-pixel. So if you find an interesting position (x, y) in the RGB image, you can simply look up `depth_image[y, x]` in order to get the distance at that (x, y) position.

If you want to visualize the Depth image, you can do it as such:

```
depth_image_scaled = cv2.convertScaleAbs(depth_image,
                         alpha=0.3)
depth_image_colormap = cv2.applyColorMap(depth_image_scaled,
                         cv2.COLORMAP_JET)
cv2.imshow('depth image', depth_image_colormap)
cv2.waitKey(0)
```

3.3 Task 0 - Getting the Images

In the perception mini-project, you got to implement some logic in the `RGBListener` and `DepthListener` utility classes. You will again use those same utility classes. So your first task is to implement that logic again. This should be a mere copy-paste from your mini-project.

3.4 Task 1 - Block Detection

The first goal of the perception module is to detect a block and return its position in the image. For this task, you should be able to reuse a lot of your code from the mini-project.

Before triggering your detection module, the arm will be moved into a lookout position, looking straight down at the table. As such, you can expect a similar scenario as in the mini-project.

3.5 Task 2 - Free-Space detection

You second task is to implement free-space detection in order to determine where a block can be placed. Like the block detection task, you should be able to reuse a lot of code from the mini-project.

3.6 Task 3 - Transforming pixel coordinates into 3D space

Once you have a pixel (x, y) location of your block (or free-space), you need to transform that location into 3D space. This will allow the arm to move into that 3D location, and grab (or place) the block.

The first step to computing a 3D location is to combine your (x, y) pixel location with information from the depth camera, so that you get a (x, y, depth) location. This is done for you in the code skeleton.

In order to compute a 3D location from (x, y, depth) coordinates, we will use a basic pinhole camera model:

$$\begin{bmatrix} X_{pixel} \\ Y_{pixel} \\ 1 \end{bmatrix} \sim \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{3D} \\ Y_{3D} \\ Z_{3D} \end{bmatrix}.$$

The first thing you should note about this model is that we use a tilde ($\sim$) instead of an equals sign ($=$). If you haven't seen this notation before, $\vec{X} \sim \vec{Y}$ means $\vec{X}$ *is similar* to $\vec{Y}$, and it is equivalent to writing $c\vec{X} = \vec{Y}$ (where c is a scalar constant). Thus we have:

$$c \begin{bmatrix} X_{pixel} \\ Y_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{3D} \\ Y_{3D} \\ Z_{3D} \end{bmatrix} = \begin{bmatrix} f_x X_{3D} + c_x Z_{3D} \\ f_y Y_{3D} + c_y Z_{3D} \\ Z_{3D} \end{bmatrix}.$$

Now, solving for c, we see that $c = Z_{3D}$ (make sure you understand why this is the case). As such we get:

$$\begin{cases} X_{pixel} Z_{3D} = f_x X_{3D} + c_x Z_{3D}, \\ Y_{pixel} Z_{3D} = f_y Y_{3D} + c_y Z_{3D} \end{cases}.$$

In the code skeleton, you will be provided with f_x, f_y, c_x, and c_y. It is your job to use these in the formula above in order to calculate X_{3D} and Y_{3D}.

Note: we haven't given you an equation for Z_{3D}, why is that?

3.7 Task 4 - Transforming a 3D position

The camera is mounted on Heron's wrist. So when you compute a 3D position, it is in relation to the camera's coordinate system. However, the block you're picking (or placing) lives in a different coordinate system. As such, you need to transform your block (or free-space) position into its respective coordinate system.

ROS has a library called *tf*, that handles coordinate frames and transformations between them. The camera and the 3D position you compute live in a coordinate frame called *realsense_rgb_optical_frame*. The block (or free-space) lives in another frame called, for example, *skiros:Workstation-1*. This frame is determined by the world model, and it will be given to you in the code skeleton.

In order to transform between these two frames, you will use the class `TransformListener` in the `tf` package. Have a look at the function `transformPose` on this page. Your TA will help you understand and use this function.

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 course, 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.

4.1 Compliant Control

For the parts that are contact-rich, we are going to use a controller that is compliant. Meaning that the robot arm would not be stiff, but reacts to the sensor data that comes from the force-torque sensor. This is similar to the robot in this video except that the UR5 would only react to the end effector.

A compliant control solution is often preferred when a workspace is shared with humans or if the environment has uncertainties such as varying poses for object. The compliant control solution on Heron utilizes this controller package. A few things are important to note:

1. The control response implements a spring-damper system between the current pose of the end effector and some reference pose. Just like a spring, the repelling force would become bigger, the bigger the distance. You can see a demonstration here.
2. As one of the inputs, it accepts a reference pose, so a pose where the end-effector "should ideally be" and this way the gripper can be moved to a new location. We typically use some trajectory generation to create smooth reference poses.
3. The controller does not do path planning and is not directly aware of kinematic constraints such as maximum joint angles. Therefore it is not suited to cover larger distances and we will use motion planning to do that.
4. This control solution is not very stable in singularities and they should therefore be avoided.

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:

```
$ roscore &
```

```
$ roslaunch heron_launch simulation.launch
```

To run the launch file on the real robot, you would run (on Robotmind2):

```
$ roscore &
$ roslaunch heron_launch real.launch
```

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 4.

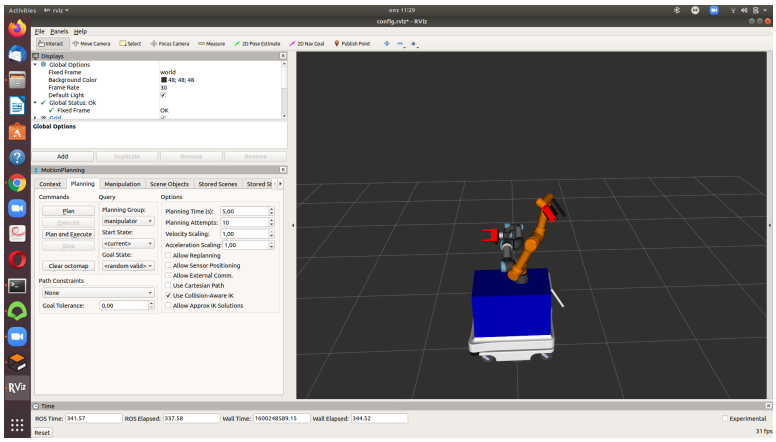

Figure 4: Using OMPL motion planning library to plan for a trajectory from an initial pose to the goal pose and simulating the motion plan in Rviz.

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.

4.2 Overall Task Description

With your manipulation team you will create skills 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,
4. going back to the arm's home position.

In order to allow for these tasks, you will implement two basic capabilities:

- moving the arm into a fixed joint state,
- moving the gripper to a specified (x, y, z) position.

4.2.1 Moving to a Fixed Joint State

The arm has a total of 6 moving joints. Each of these joints can be controlled to move into a fixed angle. Joint states are specified using a vector of angles (for example $[\frac{\pi}{2}, \pi, 0, 0, \frac{\pi}{2}, \frac{\pi}{2}]$). These can be sent directly into MoveIt, and MoveIt will handle planning and execution for you. You will **not** need to find the specific joint angles; they will be provided for you.

4.2.2 Moving the gripper to a (x, y, z) position

When you want to pick up a block, you may know the (x, y, z) position of the block, but not the corresponding joint angles. As such, you should also implement functionality to move by (x, y, z) coordinates. When implementing this funcionality, you will be provided with a full ROS `Pose` message, that also contains rotation information. These Pose messages can be sent directly to Moveit, just like the joint angles.