COMP0250: Robot Sensing, Manipulation and Interaction

Coursework 1: Pick and Place, Object Detection and Localization

Due Date: 10 March 2024 at 16:00 (UK time)

Worth: 40% of your final grade

Setup:

For the coursework you will need the comp0250_s25_labs repository (as we did in the lab sessions), here is a recap:

```
> git clone https://github.com/surgical-vision/comp0250_s25_labs.git --recurse-
submodules
```

If you already have this repository, make sure to pull new updates:

```
> cd comp0250_s25_labs

> git pull
```

The coursework is set up in a package called **cw1_world_spawner**.

The next step is to **create the package which will contain your solution**. Each team will submit one folder called **`cw1_team_<team number>`** where `<team number>` is your team number. For example, team 3 would submit cw1_team_3, team 11 would submit cw1_team_11.

When you pull the new changes (or download a fresh clone of the repo) it will now have a new package in the `src` folder called: **`cw1_team_x`**. This package contains template code for the coursework, usage is recommended but not mandatory.

```
> cd comp0250_s25_robot/src/cw1_team_x
```

A key point is that the folder includes template code for the **launch** file, used to launch the solution of the coursework.

Important: If you use the template code as a starting point make sure to change the name of your team wherever needed. So, replace `cw1_team_x` by `cw1_team_ <team number>` in the following locations:

1. The name of the package folder

2. In `launch/run_solution.launch` line 17
3. In `package.xml` line 3
4. In `CMakeLists.txt` line 2

Now, you can run the coursework with:

```
> cd ~/comp0250_s25_labs

> catkin build

> source devel/setup.bash

> roslaunch cw1_team_<team number> run_solution.launch
```

Important information:

The only directory you will submit is **cw1_team_<team number>**. All your solution code must be inside this directory.

For your solution, all necessary nodes must be launched from **run_solution.launch**. During our evaluation, no other files from that directory will be launched. We will only run:

```
> roslaunch cw1_team_<team number> run_solution.launch
```

You are allowed to add new dependencies to ROS **standard** libraries. These include sensor_msgs, geometry_msgs, moveit, cv2, pcl. If you are unsure post on the discussion forum or contact the TA team for more information. In order to simplify using MoveIt! and PCL, we recommend you use C++, but it is **not** mandatory.

You are free to edit the **`cw1_world_spawner`** package. This can help you to make tasks easier during development or to make them harder to test your solution. There are two important places to look in this package:

1. *scripts/courswork_world_spawner.py* – this file sets up the tasks, here is where you can edit them, there is an explanation of how at the top of the file
2. *srv/TaskXService.srv* – there are three service files (one per task), look at them to see what data is in the request (which you receive) and the response (which you send).

We recommend you edit *scripts/courswork_world_spawner.py* to help adjust tasks during testing. However, you will not submit your version of the `cw1_world_spawner` package, so **any changes you make to it will not be submitted**. We will mark your work with our own, clean version of `cw1_world_spawner`.

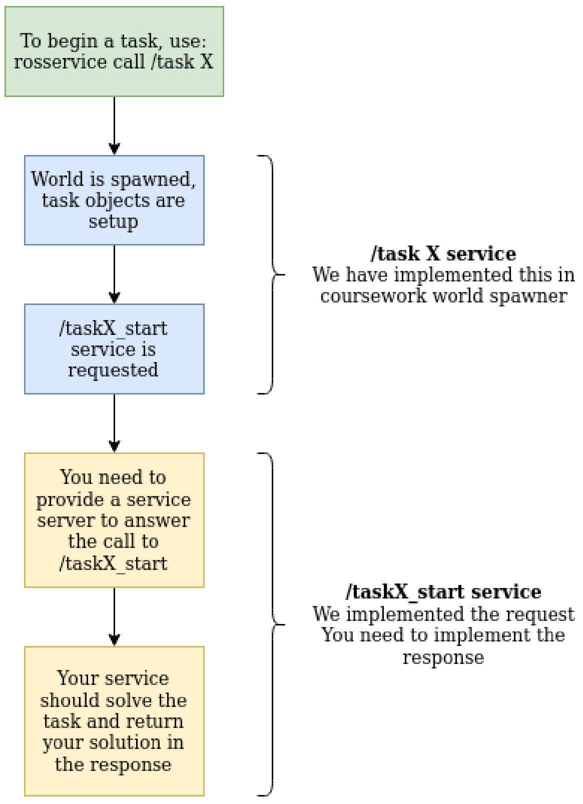
Coursework overview:

The objective of this coursework is to perform pick and place tasks in Gazebo, using MoveIt! to move the robot and PCL to detect object positions and colours. The coursework is split into **three tasks**, detailed later in this document.

To start each task, you will call the service '**rosservice call /task X**', where **X** is the number of the task. When this runs, it will automatically set up the task environment in Gazebo and the service '**/taskX_start**' will be requested. Each task takes the form of a service request that you need to respond to. The request message contains the information needed to solve the task, for example the position of an object to pick up. Your job is to receive the task request, solve the task, and fill in the solution in the task response message.

Each task's service request and response arguments are defined in *coursework_world_spawner/srv*. Thus, you will need to implement methods that receive those requests and fill in the response arguments as specified in the task descriptions later in this document.

In summary, solving each of the three tasks will look like this:


To begin a task, use: rosservice call /task X
World is spawned, task objects are setup
/taskX_start service is requested
/task X service
We have implemented this in coursework world spawner
You need to provide a service server to answer the call to /taskX_start
Your service should solve the task and return your solution in the response
/taskX_start service
We implemented the request
You need to implement the response

Evaluation:

Your submission will be evaluated on:

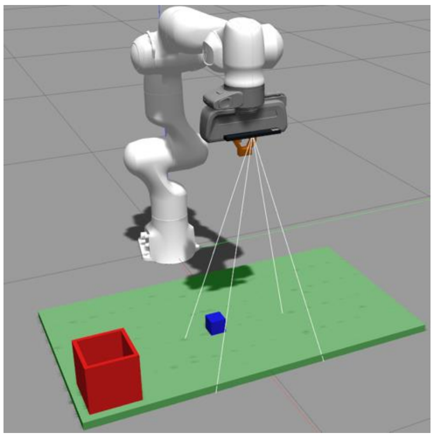
- the **correctness** of your outputs and behaviour of the robot - the robot should not collide into the ground, obstacles or itself
- the **performance** of your code – the solutions should not take too long, the design choices made should be reasonable and efficient.
- the **clarity and presentation** of your code - repetitive code should be structured into functions, comments should be used well, a README containing information about your code, authors, how it should be built and run, etc. If we fail to build, we will follow whatever instructions you have provided in the README and **will not attempt to fix anything major that is broken.**

For all these tasks, you are not allowed to:

- **tweak internal robot parameters** for speed and acceleration,
- **access the location of the spawned models** through any means other than provided by the service calls or using the RGB-D input topics

Submission:

1. Submit your folder **cw1_team_<team number>**.
2. **README** file: make sure you include licenses, authors, how to build and run the package, and importantly include for each task below how much total time (roughly number of hours) and the percentage each student of the team worked on the tasks. We expect roughly equal time of contribution by each student.



Task 1: MoveIt! - Pick and Place at given positions [Grade: 30%]

***Grade:* 15%-Correctness; 7.5%-Style; 7.5%-Structure/Efficiency**

The environment for this task is represented by a grid of tiles. A cube is spawned within reachable distance, on a grass tile.

The task is to pick up the cube and place it in the basket without collision.

To begin the task, just call:

```
> rosservice call /task 1
```

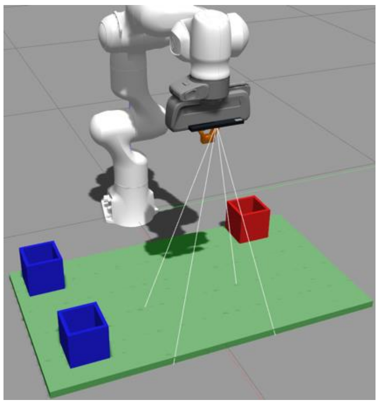
This will automatically run the service **/task1_start**, passing two inputs:

geometry_msgs/PoseStamped **object_loc**
geometry_msgs/PointStamped **goal_loc**

An object is located at **object_loc,** where **object_loc** is the centroid of that object. For this task, you can assume the shape of the object is known - a cube with size [0.04, 0.04, 0.04]. The goal located at **goal_loc** is the position of a basket with size [0.1, 0.1, 0.1] placed on the ground. Both the basket and box will always be spawned with the same orientation (q = [0,0,0,1]) in the world frame. This is true for all tasks.

Place the object in the basket without colliding with it and the ground or self-colliding.

Once the task is completed, return the ServiceResponse. For task 1, it should be empty.



Task 2: PCL - Object Detection & Colour Identification [Grade: 35%]

***Grade:* 20%-Correctness; 7.5%-Style; 7.5%-Structure/Efficiency**

The environment for this task is represented by a grid of tiles. There will be baskets (size [0.1, 0.1, 0.1]) spawned randomly. The spawned baskets will be coloured in any combination of red, purple, blue. You are given positions where baskets might be.

The task is to report the basket colours at each location or if any locations are empty.

To begin the task, just call:

```
> rosservice call /task 2
```

This will automatically run the service **/task2_start**, passing a vector of possible basket positions for you to check:

geometry_msgs/PointStamped[] basket_locs

Using the camera topics starting with '**/r200/**', you must check each point to determine 1) if a basket is there and 2) what colour is the basket. You then return a vector of strings describing what is at each point (in the same order). The **only** possible strings are:

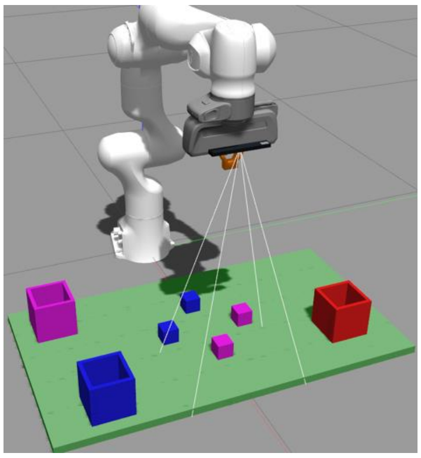
- "blue" - if a blue basket is at that point, RGB = [0.1, 0.1, 0.8]
- "red" - if a red basket is at that point, RGB = [0.8, 0.1, 0.1]
- "purple" - if a purple basket is at that point, RGB = [0.8, 0.1, 0.8]
- "none" - if there is no basket at that point (note: strings are not case-sensitive)

string[] basket_colours

For example this vector could be ["blue", "red", "purple", "none"]

Key points:

1. Match the right basket to the right string colour description
2. The vector of strings must be in the same order as the vector of points.
3. When we test your code we may alter the numbers of baskets.



Task 3: Planning and Execution [Grade: 35%]

***Grade:* 20%-Correctness; 7.5%-Style; 7.5%-Structure/Efficiency**

The environment for this task is represented by a grid of tiles. Several graspable boxes with size [0.04, 0.04, 0.04] are placed **in front** of the robot on the plane, along with coloured baskets size [0.1, 0.1, 0.1].

The task is to place each cube into a basket of the same colour. Blue cubes should go into the blue basket, red cubes into the red basket, and purple cubes into the purple basket.

To begin the task, just call:

```
> rosservice call /task 3
```

This will automatically run the service **/task3_start**, in this task there is no data provided and no data requested from the service.

Using the camera topics starting with '**/r200/**', find all objects belonging to each colour (blue, red, purple) and pick-and-place them in the basket which has the same colour. You will need to find the position of the boxes and the baskets.

Key points:

1. It is not guaranteed that there will always be a box for each present basket. If you refer to the figure, there is not a red cube.
2. It is not guaranteed that there will always be a basket for each box present. When we test your code we may alter the numbers of boxes and baskets.
3. There may also be other edge cases that can be tested.