COMP0129: Robot Sensing, Manipulation and Interaction

Coursework 3: Pick and Place, Object Detection and Localization

Due Date: 14th April 2023 at 16:00 (UK time)

Worth: 35% of your final grade

Setup:

For the coursework you will need the comp0129_s23_robot repository (as we did in the lab sessions), here is a recap:

```
> git clone https://github.com/COMP0129-UCL/comp0129_s23_robot.git --recurse-
submodules
```

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

```
> cd comp0129_s23_robot

> git pull –recurse-submodules
```

The coursework is set up in a package called **cw3_world_spawner**. This needs to be added to the comp0129_s23_robot folder on your machine. We will **add this as a submodule**:

```
> cd comp0129_s23_robot/src

> git submodule add https://github.com/COMP0129-UCL/cw3_world_spawner.git
```

The next step is to **create the package which will contain your solution**. Each team will submit one folder called **`cw3_team_<team number>`** where `<team number>` is your team number. For example, team 3 would submit cw3_team_3, team 11 would submit cw3_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: **`cw3_team_x`**. This package contains template code for the coursework, usage is recommended but not mandatory.

```
> cd comp0129_s23_robot/src/cw3_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 `cw3_team_x` by `cw3_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 ~/comp0129_s23_robot

> catkin build

> source devel/setup.bash

> roslaunch cw3_team_<team number> run_solution.launch
```

Important information:

The only directory you will submit is **cw3_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 cw3_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 **`cw3_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/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/world_spawner.py* to help adjust tasks during testing. However, you will not submit your version of the `cw3_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 `cw3_world_spawner`.

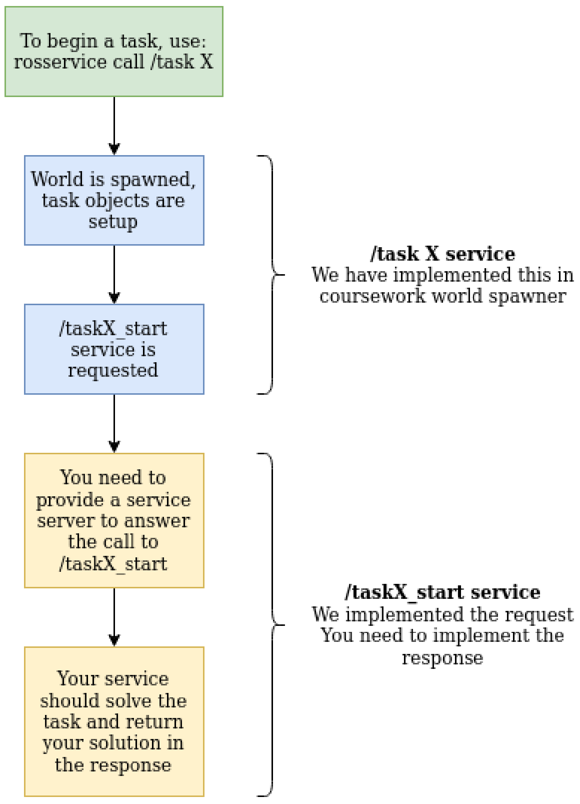
Coursework overview:

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 *cw3_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:

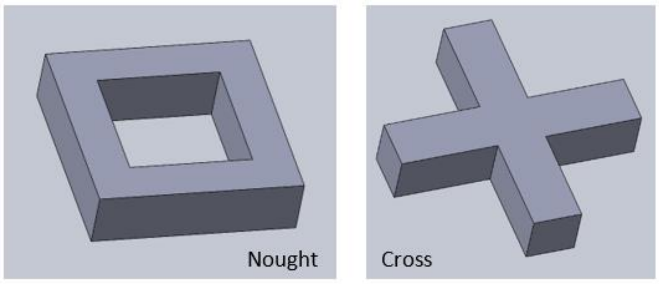
- **tweak internal robot parameters** for speed and acceleration,
- **access the location of the spawned models** through any means other than provided by the service calls or using the RGB-D input topics

Submission:

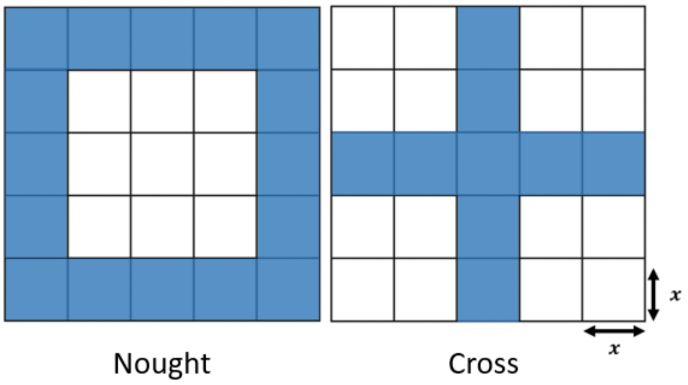
1. Submit your folder **cw3_team_<team number>**.
2. **README** file: make sure you include licenses, authors, how to build and run the package, and importantly include for each task below how much total time (roughly number of hours) and the percentage each student of the team worked on the tasks. We expect roughly equal time of contribution by each student.

Shapes used in this coursework:

This coursework is about identifying shapes, the shapes will be a 'nought' and a 'cross', as shown:

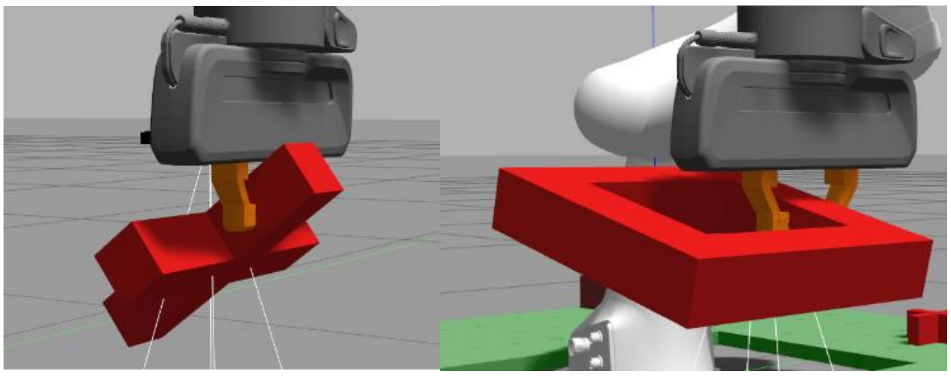


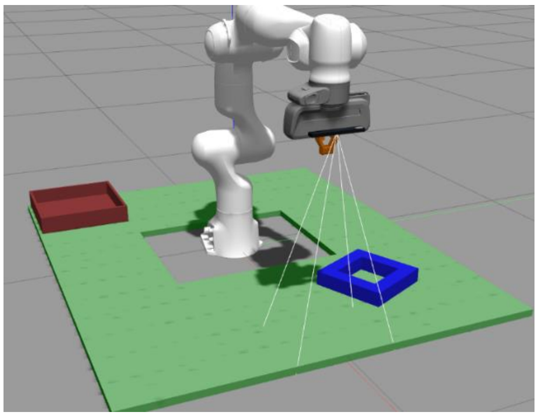
Both shapes are constructed from adding squares together on a 5x5 grid:



- Shapes will always have a height of 40mm in the vertical (z) direction
- The value of x (in above Figure) can either be 20, 30, or 40mm
    - $x = 20mm$ means the whole shape fits inside a 100mm square
    - $x = 30mm$ means the whole shape fits inside a 150mm square
    - $x = 40mm$ means the whole shape fits inside a 200mm square

Grasping of these shapes is possible in Gazebo, but the friction behaviour of the Panda gripper in simulation is a little unrealistic. These objects rotate in the grasp but do not fall, as shown below:





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

***Grade:* 20%-Correctness; 5%-Style; 5%-Structure/Efficiency**

The environment for all tasks is represented by a grid of tiles which go all the way around the robot. Shapes in all tasks are spawned within reachable distance on a grass tile.

The task is to pick up the shape and place it into the brown basket.

To begin the task, just call:

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

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

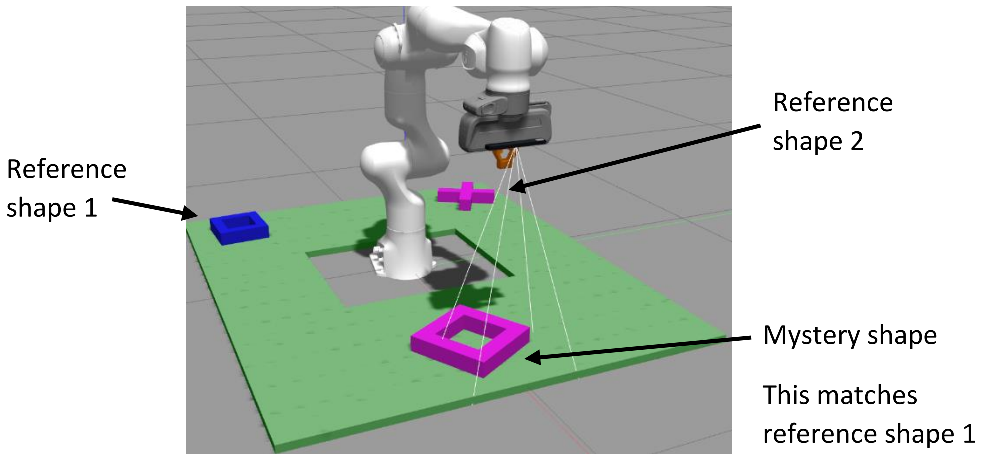
geometry_msgs/PoseStamped **object_point**
geometry_msgs/PointStamped **goal_point**
string **shape_type**

A shape is located at **object_ point,** where **object_ point** is the centroid of that shape. The goal located at **goal_point** is the position of a basket with size [0.35, 0.35, 0.05] and RGB = [0.5, 0.2, 0.2] placed on the ground. The basket will always be spawned with the same orientation (q = [0,0,0,1]) in the world frame. This is true for all tasks. The shape of the object is given by **shape_type** which can only either be "nought" or "cross".

Grasp the object and place it in the basket to complete the task. Once the task is completed, return the ServiceResponse. For task 1, it should be empty.

Key points:

1. The spawned shape can potentially have any orientation
2. The shape will either be a nought or a cross with size $x = 40mm$
3. The shape should be placed entirely inside the basket to complete the task
4. When we test your code we may alter ONLY the following task parameters from world_spawner.py:
    a. T1_ANY_ORIENTATION (either False/True)



Task 2: Shape detection [Grade: 35%]

Grade: 25%-Correctness; 5%-Style; 5%-Structure/Efficiency

In this task you are given three points, at each of these points a shape is spawned. Two points are 'reference' points and two different shapes will spawn, one at each. The third point is the 'mystery' shape, which will be the same as one of the reference shapes.

The task is determine which reference shape the mystery shape matches.

To begin the task, just call:

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

This will automatically run the service **/task2_start**, passing a vector of two references shape points, and a mystery shape point:

geometry_msgs/PointStamped[] ref_object_points

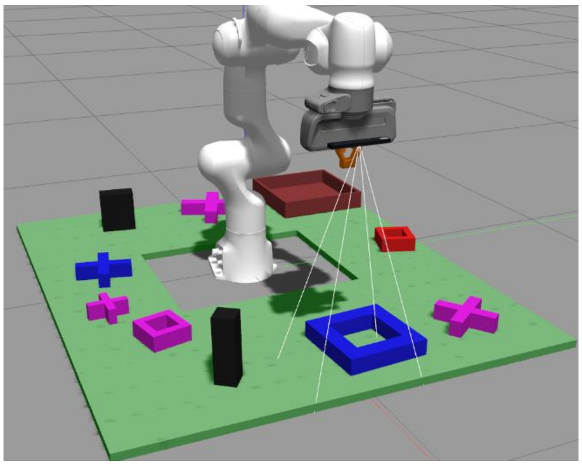
geometry_msgs/PointStamped mystery_object_point

You need to return a number in the service response, either '1' or '2' which corresponds to which reference the mystery shape matches. In the picture above the correct answer is '1'.

int64 mystery_object_num

Key points:

1. The spawned shape can potentially have any orientation
2. The green tiles (ie ground) may move vertically upwards to a maximum of $50mm$
3. The shape will either be a nought or a cross with size $x = 40mm$
4. When we test your code we may alter ONLY the following task parameters from world_spawner.py:
   a. T2_ANY_ORIENTATION (either False/True)
   b. T2_GROUND_PLANE_NOISE (either 0e-3/50e-3)



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

Grade: 25%-Correctness; 5%-Style; 5%-Structure

In this task up to 7 shapes can spawn, alongside black obstacles with RGB=[0.1, 0.1, 0.1] and a goal basket.

The task is to:

- **Count the total number of objects, excluding the black obstacles**
- **Determine which shape is more common (or if they are equal)**
- **Pick up and place one example of the most common shape into the basket whilst avoiding touching any of the obstacles**

If the number of objects present for two shapes are equal (both shapes equally common) then you may place an example of either shape into the 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 for the service.

There are two values you should set and then return in the service response message:

int64 total_num_shapes

int64 num_most_common_shape

You need to return the total number of shapes (excluding the black obstacles). You also need to return the number of the most common shape, for example in the above picture there are 7 shapes, 3 noughts and 4 crosses. So crosses are the most common shape, and you should return total_num_shapes=7 and num_most_common_shape=4.

Key points:

1. The spawned shapes can potentially have any orientation
2. The shapes will either be a nought or a cross
3. The shapes may have sizes $x = 20mm$ or $x = 30mm$ or $x = 40mm$
4. There may be up to 7 shapes and up to 4 obstacles in the scene.
5. When we test your code we may alter ONLY the following task parameters from world_spawner.py:
    a. T3_ANY_ORIENTATION (either False/True)
    b. T3_N_OBSTACLES (either 0 or 2/3/4)
    c. T3_USE_MULTIPLE_SIZES (either False/True)

Final notes on marking

In all tasks you will get credit for what you can solve.

Each of the variables we may change has two options, either an easier option (like False or 0) or a harder option (like True or 50e-3). Roughly speaking the marks for correctness will be split equally between all options. You will also get partial credit for partial solutions.

As an illustrative (approximate) example:

1. Task 1 correctness is worth 20%:
    a. ~10% for solving with T1_ANY_ORIENTATION=False
    b. ~10% for solving with T1_ANY_ORIENTATION=True
2. Task 2 correctness is worth 25%:
    a. ~12% for solving with both variables=False
    b. ~6% for solving with T2_ANY_ORIENTATION=True
    c. ~6% for solving with T2_GROUND_PLANE_NOISE=True
3. Task 3 correctness is worth 25%
    a. ~12% solving with all variables=False/0
    b. ~4% solving with T3_ANY_ORIENTATION=True
    c. ~4% solving with T3_N_OBSTACLES=2/3/4
    d. ~4% solving with T3_MULTIPLE_SIZES=True