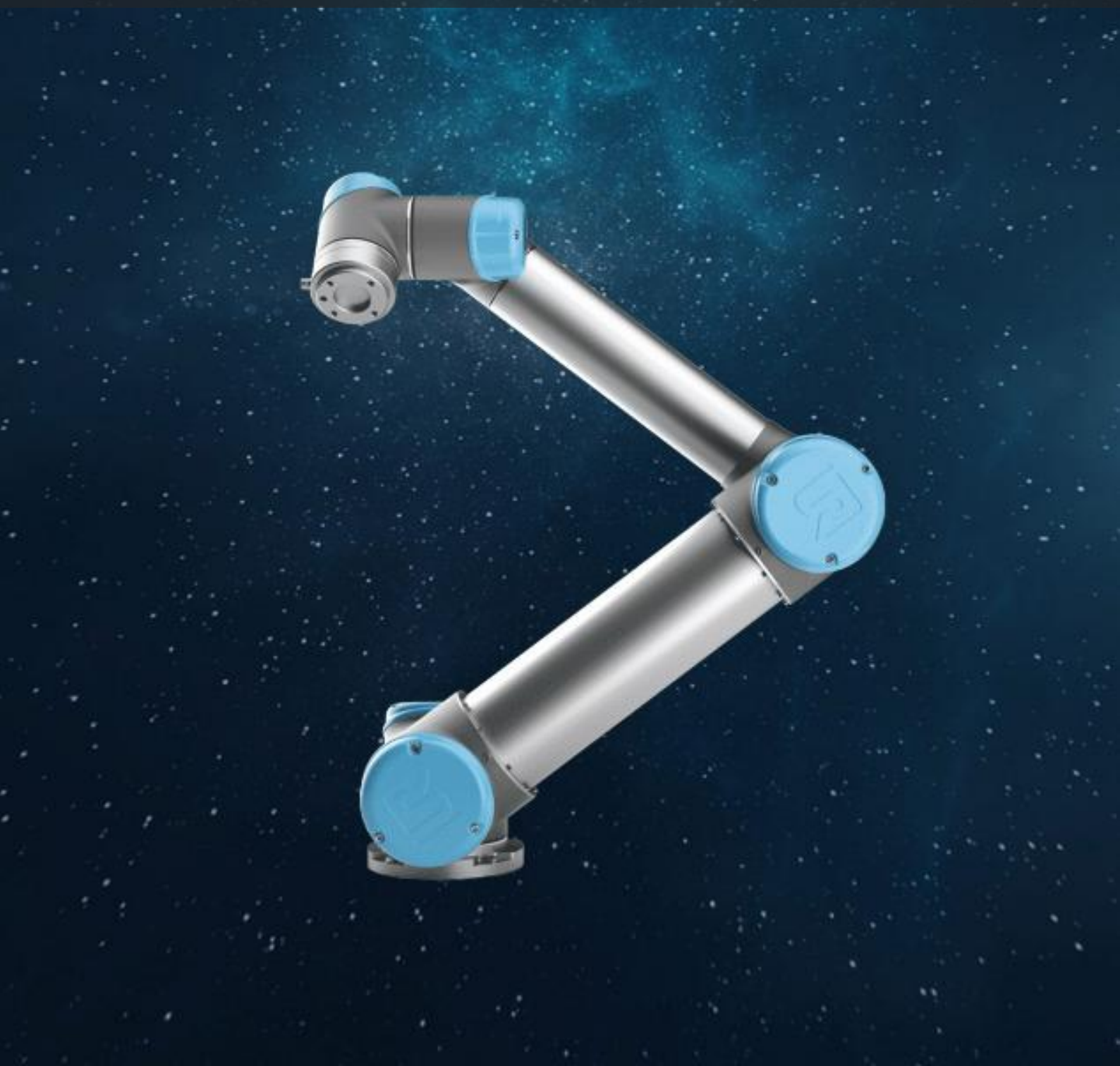
Collision Detection System for UR5

Setup Documentation



Contents

1

Hardware Setup

This system uses the Intel RealSense D455 camera to detect collision. It is independent of the collision avoidance system in MoveIt. This system will simply detect collision and stop the robot allowing for MoveIt to re-plan its trajectory and avoid new obstacles. It is possible to use the collision detection code without MoveIt, as the code simply posts a ”stop” or ”clear” message on a topic.

Before setting up the system, it is optimal to initiate the code without the robot present in the workspace. When starting the code for the first time, the user must save the static environment around the robot, this is used for reducing the number of points in the data and thus reduction in false collision detection due to light-reflections (specifically on walls/floor). It is optimal and highly recommended to save the static environment without the robot present, but can also be done with the robot preset in the workspace. An explanation of how to save the PointCloud data of the static environment/workspace can be found in chapter 3.3.

The main camera has to be mounted directly above (or close to) the middle of the base of the robot, facing straight down on the robot. It may be possible to offset the camera horizontally, but this has not been tested and the effects of this is unknown. It might be advantageous to offset the camera horizontally in some cases and could be attempted. The camera has to always face down towards the floor and can not be mounted at an angle. If it is offset horizontally, it must still face straight down.

Secondary cameras can be mounted anywhere, although it is important to remember the maximum range of the D455 cameras which is .6 to 6 meters.

For optimal performance, this should be taken into account:

- One main camera should be mounted directly above the base of the robot, and has to face straight down, it can not be angled diagonally.
- This code supports the use of additional ”secondary” cameras, these cameras will only be used for generating an occupancy-grid, and will not be used for collision detection, as it is impractical to use other cameras for this task because of how this code works. Each secondary camera used has a performance impact that depends on the resolution of the cameras, FPS, etc... which is important to keep in mind.
- The main camera is set to a resolution of 640x480 at 30 FPS. Higher resolution will force the camera into a lower FPS mode, while lower resolution will allow the use of higher FPS. The resolution of the RGB feed and PointCloud data from the main camera has to be identical.
- The code will wait until it receives a new frame from the camera before proceeding, this means lower FPS will increase the detection time of the system substantially. This also applies to secondary cameras as the code will wait for a frame from all secondary cameras before proceeding. 30 FPS is the recommended FPS for the main camera, secondary cameras should have 30 FPS

or higher. Processing of each frame is computer-intensive, much higher FPS may also result in high processing times, and thus longer detection times.

- Optimal vertical distance between main camera and robot is 3 meters. It can be mounted higher (up-to 6 meters), but higher distance will make the PointCloud data less stable and may cause other issues.
- A higher distance between the camera and robot (or camera and floor of the room) will increase the area the camera can perceive. Since the robot detects collision when objects enter a ”stop-zone” around itself, perceiving a much larger environment than the workspace of the robot is pointless and will only reduce the resolution of objects in the workspace.
- The system uses a combination of the RGB data and PointCloud data to detect collision. It is important the environment does not change lighting conditions quickly, and that sufficient lighting is provided for the camera to see the environment. The system will adjust its RGB frame of reference such that gradual change in lighting conditions will not cause issues. Blinking lights may trigger false collision, but this may not always be the case.
- Objects that are $\tilde{<}$5cm in size may not be successfully avoided by MoveIt. The detection system will successfully detect collision and stop the robot, but as the object gets closer to the robot (such as a hand placed on the robot) will turn ”invisible” to MoveIt due to self-filtering. When the robot begins to move again, it may collide with the hand. Self-filtering parameters can be tweaked in the ”sensors_3d.yaml” file. Less aggressive filtering will be safer, but may trigger false collisions regularly.
- A static object of what the robot is mounted on has to be defined either in the URDF or posted as a static object to MoveIt such that MoveIt can filter away the occupancy grid of the table. If this is not done, there may be large occupancy-voxels on the surface of the table which force the robot to take unreasonable trajectories to avoid them.

2

General Information

This system can be used in any ROS workspace. The ”CollisionDetection.py” file is the main file for the system. It is dependent on the ”cameracfg.json” file and ”data.npy” files that must remain the the same folder as the python-code.

Some things need to be taken into account before setting up the system:

- The ”camera/depth/color/points” topic responsible for publishing the PointCloud data for generation of the occupancy-grid is ”projected” down in z-direction such that ”blindzones” are filled with an occupancy grid. This PointCloud data may seem to be wrong, but this is an intended feature.
- This system does not stop the robot directly, but rather posts a ”stop” or ”clear” message on a topic. This message is posted on the ”collision_detection” topic as a string. It is up to the user to implement a stop function into the code that is responsible for movement of the robot.
- The raw D455 camera output can be visualized in the ”realsense-viewer” application and settings can be tuned in the application before applying them to the code if needed. Configuration of camera parameters and other settings are pre-configured for best performance and no additional adjustments should are needed. Changes made in the ”realsense-viewer” application will not be applied to the camera after closing the application and is only used for visualization.
- The system (python code) publishes PointCloud data for each individual camera, each PointCloud needs to be used as an input for the self-filtering in MoveIt (Through the .yaml file used for self-filtering). An example of a .yaml file for self-filtering that supports the use of several cameras can be found in A.
- If false collisions are triggered while the robot is running, higher ”padding_offset” or ”padding_scale” may be needed in the self-filtering ”sensors_3d.yaml” file.
- The camera can not be directly connected to ROS, this will ”lock” the camera feed and the python code will not be able to use (connect to) it. Make sure nothing else is using the camera. The python code will publish the pointcloud (and RGB) data on a predefined topic such that it can be used by ROS for other purposes. See 3.7 for more information on the topics.
- The names of some of the joints / links (frames) of the robot needs to have specific names for the code to be able to look up the transfer data. The names of these frames can be viewed in 3.8

3

Software Config

The main code ”CollisionDetection.py” is dependant on ”data.npy” and ”cameracfg.json”, make sure to have all 3 files in the same folder.

In order to configure the code, simply open up ”CollisionDetection.py”, the top section of the code is used for configuration.

Before starting the code, make sure the master node is running (roscore), MoveIt with perception is working (To generate an occupancy grid with an input from a PointCloud topic), the necessary nodes to control and connect to the robot (and the robot itself) is running.

The code needs to be correctly configured before starting it, but after configuration it can be started with the following command (command needs to be run from the same folder the code and its dependencies are located in):

```
python3 CollisionDetection.py
```

3.1. Workspace Setup

In order to use this system, the TF data is needed from the robot. This usually means a ROS workspace needs to be setup and configured for the UR5. MoveIt is needed for generating an occupancy-grid for collision avoidance and Rviz is recommended for configuring and setting up the collision detection system.

3.2. Installing Libraries

The collision detection code uses several python libraries. Make sure to install all libraries used by the python code, the required libraries can be viewed by opening the ”CollisionDetection.py” code and looking at the imports on top.

3.3. Main Config

Before being able to use the code, the code needs to save one frame of PointCloud data (static initial frame) to filter away the static environment such as the floor, walls and the table. This step is crucial and can not be skipped. Moving around objects that were in the frame when it is saved is not a problem, the goal is to save the floor and walls such that these do not trigger collision due to shadows/reflections.

First the serial number of the main camera must be found, it be found on the underside of the camera as a 12 digit number, for example ”031422250715”.

Input the serial number in the configuration of the python code:

```
main_camera_serial = "031422250715"
```

Optimally, the robot should be removed from the workspace / environment when saving the ”static initial frame”. If removing the robot from the workspace is impractical, it is still possible to save an initial frame. This can be done by orienting the robot to face straight up towards the main camera (assuming the main camera is correctly mounted above the base of the robot, facing down), such that the robot is in a straight ”vertical” position with the tool-end pointing towards the camera.

If the robot is removed from the workspace, or is pointing vertically towards the camera, the static initial frame can be saved by changing the ”startup” variable in ”CollisionDetection.py” to ”True” as shown:

```
startup = True
```

Note: The ”static initial frame” is stored in a file and has to only be saved once. Whenever the main camera is moved to a new location, a new ”static initial frame” should be created in a similar matter.

Afterwards, the code can be run with the command:

```
python3 CollisionDetection.py
```

After saving the ”static initial frame”, the code will shut down and a message confirming this should be shown in the console. If the frame was successfully saved, the ”startup” variable has to be set to ”False”:

```
startup = False
```

After setting startup to ”False”, the robot must be turned on and the tf-data from the robot must be available! In addition, the master node (roscore) must also be running.

Next, 3d perception needs to be enabled in MoveIt. This can be done by adding this section to the ”sensors_3d.yaml” file (or the equivalent file used in the ROS workspace).

”padding_offset” can be changed if false occupancy-voxels are generated because of incorrect self-filtering, this will result in the robot trying to avoid it self (or refuse to move in some cases). This value should be the lowest possible for safety reasons, if this value is too large, objects that move very close to the robot will not be avoided.

```
sensors:
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
    point_cloud_topic: camera/depth/color/points
    max_range: 5
    point_subsample: 1
    max_update_rate: 10
    padding_offset: 0.1
    padding_scale: 1.2
    filtered_cloud_topic: filtered_cloud
```

Next, some variables must be tuned:

The ”buffer_desity” defined how many spherical regions along the robot arm should be used to check for collision. Must be 4 or more. 6 is recommended.

The ”stop_radius” defines the radius (in meters) of the spherical regions. Bigger is safer, but increases the chance of false detection. Should be minimum 0.7.

The ”stop_radius” should be as large as practically possible for safety reasons, it should be increased after testing the code and making sure it works correctly with the default value.

The ”filter_radius” is the radius of spherical regions for self-filtering (detection only, not avoidance), default is 0.65 . Should not be higher than default unless false collision is detected when the robot is moving, should be the lowest possible value without triggering false collision.

The ”object_filter_radius” is an optional setting and can be left as default for now. More documentation and explanation on how to set this up can be found in 3.5.

```
buffer_density = 6

stop_radius = 0.7

filter_radius = 0.65

object_filter_radius = 0.3
```

Next, the position relative to the ”world” has to be defined in the code, this position will be set by a tf-broadcaster such that ROS has the correct tf-data.

Rviz should be used to visualize the ”/collision_detection_raw_point_cloud” topic combined with a visual representation of the robot (MotionPlanning by MoveIt in Rviz), to align the PointCloud data with the robot for correct self-filtering. Note that the values for camera position define the position of the camera relative to the ”world” frame, not the ”robot” frame.

The ”camera_x” and ”camera_y” is the horizontal position of the camera. The ”camera_height” is the distance between the camera and the floor, not the distance between the camera and the robot. The ”camera_rot” variable is used to define the rotation:

```
camera_height = 2.73
camera_x = 0.07
camera_y = -0.04

camera_rot = (0, math.pi, 0)
```

Similarly, the static position of the robots base relative to the ”world” frame has to be defined. (If a robot that can walk/drive around in the workspace is used (like the Tiago), see 3.6 and ignore this step). The ”world” is the environment (workspace of the robot), with the origin of the ”world” frame being the same as the cameras x, y -origin, but the height of the robots origins (z) may differ:

```
robot_height = 0.74

robot_rot = (0, 0, -0.7854)
```

The code should now run with one main camera and detect collision successfully. If collision is detected, a ”stop” message should be printed in the console. Rviz should be used while subscribing to the different topics listed in 3.7 for a better understanding of how the system detects collision.

It is possible to connect multiple cameras to the system, but this is optional, how this can be done is explained in 3.4.

3.4. Secondary Cameras

Multiple cameras can be used for collision avoidance (generating an occupancy-grid), but only the main camera is used for collision detection.

Only one secondary camera should be added at a time, do not attempt to add multiple cameras at once, this will lead to confusion. After one secondary camera has been successfully connected and tested, more can be added to the system.

The ”secondary_cameras” is a list of serial numbers similar to the one used for setting up the main camera. This should be in the form of a list, for example: [”031422250497”, ”031422250498”, ”031422250499”] for 3 secondary cameras.

The ”secondary_cameras_tf” is a list of the position of the cameras relative to the ”world” frame, similar to the position defined for the main camera. This should be in the form of a list with the (x, y, z, r, p, y) values, for example: [(0, 2, 0.7, -math.pi/2, 0, math.pi)] for 1 secondary camera.

These variables need to be changed to add secondary cameras:

```
secondary_cameras = []

secondary_cameras_tf = []
```

It is also advisable to look at the ”sensors_3d.yaml” file (or the file used in a custom ROS workspace for MoveIt self-filtering (Tutorial can be found here: 4).

An example ”sensors_3d.yaml” file pre-configured to accept 1 main and 2 secondary cameras can be seen in appendix: A.

The ”padding_offset” value can be changed if false occupancy-voxels are generated because of incorrect self-filtering, this will result in the robot trying to avoid it self (or refuse to move in some cases). This value should be higher for secondary cameras than that used for the main camera.

Make sure to add (copy & paste) this section in the ”sensors_3d.yaml” file for every secondary camera used, and change the ”n” at the end in ”point_cloud_topic: secondary_camera_n” where ”n” is the number of the secondary camera starting from 0 (first secondary camera is 0) and increasing by one for every camera that is used (the second ”secondary camera” is 1, third is 2, etc...

```
- sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
  point_cloud_topic: secondary_camera_n
  max_range: 5
  point_subsample: 1
  max_update_rate: 1
  padding_offset: 0.2
  padding_scale: 1.2
```

After adding a new section to the ”sensors_3d.yaml” file, the MoveIt node has to be restarted. After restarting the node and running the code, the secondary camera should be working and generating an occupancy grid.

3.5. Preventing Collision Detection With Specific Robots / Objects

In order for the system to not trigger collision with specific objects or other robots, some additional configuration is needed.

Configuring this is mostly only required if the specific objects / robots move around in the workspace. Interaction with small objects ($\sim$ 7x7x7cm) that stay still after / before interaction might not trigger collision, but MoveIt will still prevent the robot from interacting with these small objects and MoveIt needs to be configured to allow interaction.

To prevent objects / robots from triggering collision, a list of the X, Y, Z points with the locations of the objects / robots is needed. These points should be the origin of the objects.

A list of coordinates of the specific objects / robots must be created, a transform-listener can be used to get the coordinates of other robots. The coordinates must be a list with the name same format as shown:

The list of coordinates of specific objects / robots must be named ”objects”. After creating/importing such a list, the code will automatically detect that it exists and use it to filter objects, example:

```
objects = [(0, 0, 1), (2.5, 2, 0.7)]
```

Length of the list does not matter, as long as its a list with tuples: [(x, y, z), (x, y, z), (x, y, z), ...]

The list of x, y, z points is used to define the centre of spheres that are used to filter away PointCloud data, and thus prevent triggering collision with these objects.

The radius (in meters) of spheres that are used to filter away the PointCloud data of objects must be defined in the config section in ”CollisionDetection.py”. This value should be slightly bigger than the maximum width of the largest objects / robots divided by 2:

```
object_filter_radius = 0.3
```

If the specific objects are very large, it is advisable to use a smaller object_filter_radius, but use several x, y, z points as origins for spheres so there are multiple smaller filter-spheres filtering the PointCloud data of specific objects / robots.

In addition to the configuration done of the detection system, additional configuration is needed such that MoveIt filters away the occupancy-grid (voxel-data) of the objects / robots and allows the main UR5 robot to interact with them. The configuration done of the collision detection system will only prevent triggering collision, MoveIt might still refuse to move near the objects.

3.6. Using the code for other robots than the UR-series

By default, the code is configured to work with the UR-series (Specifically the UR5, but should work with UR3 / UR10 without any reconfiguration as long as link / joint names are the same). The code uses the TF data from the robot to generate ”X, Y, Z” points to use as centres of spherical zones. The zones are used for self-filtering and collision detection.

It is possible to tweak the tf-data the code receives in order to generate points on an any-shaped robot, for example a Tiago.

In order to change the tf-function, the ”tfdata()” function must be edited in ”CollisionDetection.py”:

```
def tfdata(listener):
  try:
    listener.waitForTransform('/camera_link', '/base_link', rospy.Time(0), rospy.Duration
        (0.1))
    (trans, rot) = listener.lookupTransform('/camera_link', '/base_link', rospy.Time(0))
    (transferone, rot) = listener.lookupTransform('/camera_link', '/wrist_2_link', rospy.Time
        (0))
    (transfertwo, rot) = listener.lookupTransform('/camera_link', '/forearm_link', rospy.Time
        (0))
    return trans, transferone, transfertwo
  except:
    pass
```

The ”tfdata()” function received the tf-data of 3 points for the robot, for robotic-arms, usually 3 points is enough. The 3 points of the robot is the base of the robot, the tool-mount (Or tool) and the ”elbow” joint of the robot (between base and tool-mount). The code generates several points along a line between tf-data point 1 and 2, and between tf-data point 2 and 3. The ”tfdata()” function looks up the tf-data relative to the ”camera_link” frame, which is the position of the camera.

In order to re-configure the function for use with other robots, 3 tf-frames need to be changed in the ”tfdata()” function. The tf-listener should always find the tf-data relative to the ”camera_link” frame, this frame is created by the code when receiving the PointCloud data from the camera.

- - **tf-point 1:**
  This is the ”base” of the robot, should be the bottom part of any robot.
  In the ”tfdata()” function in ”listener.waitForTransform()” and ”listener.lookupTransform()” change:

  ”/base_link” -> **”/your_robot_frame_base”**

- - **tf-point 2:**
  This is the ”elbow” of the robot, should be the ”middle” / ”elbow” part of any robot.
  In the ”tfdata()” function in ”listener.lookupTransform()” change:

  ”/wrist_2_link” -> **”/your_robot_frame_wrist”**

- - **tf-point 3:**
  This is the ”toold-end” (or tool) of the robot, should be the ”endpoint” of any robot.
  In the ”tfdata()” function in ”listener.lookupTransform()” change:

  ”/forearm_link” -> **”/your_robot_frame_tool-end”**

In addition, the tf-publisher of the base of the robot needs to be changed. By default, the code publishes the position of the ”base_link” of the robot relative to the ”world” frame. If the user uses a moving robot such as a Tiago, a custom tf-publisher must be created to publish the position of the ”base_link” or (”/your_robot_frame_base”). The static tf-publisher for the robot in the code needs to be deleted.

Delete the static tf-broadcaster in ”CollisionDetection.py” TWO INSTANCES, (There are two broadcasters, delete both):

```
br.sendTransform((0,0,robot_height), quaternionrobot, tfnow, "base_link", "world")
```

```
    br.sendTransform((0,0,robot_height), quaternionrobot, tfnow, "base_link", "world")
```

No more changes are needed in the code, but a custom tf-broadcaster is required and needs to be created by the user:

The user needs to create a tf-broadcaster to publish tf-data (position) between the ”base_link” (or ”/your_robot_frame_base” if another name is used) of the moving robot and the ”world” frame. The world frame is simply the ”environment”, the origin of the ”world” frame is what is used to define the position of the camera, (usually 0, 0, 0), it is basically the point on the floor beneath the main camera. Alternatively, it is possible to publish the tf-data relative to the ”camera_link” instead, but this can result in two unconnected tf-trees (use ”roslaunch rqt_tf_tree rqt_tf_tree” to check if it is connected.)

When creating the custom tf-broadcaster, it is advisable to create this in a separate code, not in ”CollisionDetection.py”.

3.7. ROS-topics

.

- **Stop / Clear - Message publisher**: Publishes a string ”Stop” when collision is detected, ”Clear” if no collision is detected.
  Topic: ”/collision_detection”
  Message type: ”std_msgs.msg.String”

- **Main Camera PointCloud Publisher**: Publishes processed and compressed PointCloud data before self-filtering is done. This data is projected in z-direction (vertically) to fill blind-zones with PointCloud data.
  Topic: ”/camera/depth/color/points”
  Message type: ”sensor_msgs.msg.PointCloud2”

- **Secondary Camera(s) PointCloud Publisher(s)**: Publishes processed and compressed PointCloud data to MoveIt.
  Topic: ”/secondary_camera_x”
  X = number of secondary camera, **First Is 0, Second Is 1, Etc...**
  Message type: ”sensor_msgs.msg.PointCloud2”

- **Detected Points Publisher**: Publishes the points that can trigger collision detection. Used for visualization and trouble-shooting.
  Topic: ”/camera/depth/color/detectedpoints”
  Message type: ”sensor_msgs.msg.PointCloud2”

- **Raw Points Publisher**: Publishes the PointCloud data after compression and filtering, this is the PointCloud data used before ”projection”, ”self-filtering” and ”RGB overlaying” is performed. This is not truly the ”raw” PointCloud data, but is filtered and processed for maximum precision, this is the topic that should be used for other applications if needed.
  Topic: ”/collision_detection_raw_point_cloud”
  Message type: ”sensor_msgs.msg.PointCloud2”

- **Image Publisher**: Publishes the RGB data from the camera. This is the raw image from the camera, this image is highly saturated and has a lot of contrast, which is required for better detection using the RGB data. The saturation and contrast are due to settings pre-tuned and uploaded to the camera on code-start.
  Topic: ”/img”
  Message type: ”sensor_msgs.msg.Image”

- **Diff Image Publisher**: Publishes the detected ”change in RGB”, this is the data overlayed over the PointCloud data to detect ”new” obstacles.
  Topic: ”/filtered_img”
  Message type: ”sensor_msgs.msg.Image”

- **Stop-Zone MarkerArray**: Publishes a ”MarkerArray” that displays spheres representing the ”stop-zones” around the robot arm, great for visualizing where collision can be detected, or what triggers collision in combination with the **Detected Points Publisher**.
  Topic: ”/visualization_marker_array”
  Message type: ”MarkerArray”

3.8. Frame Names

The links / joins (frames) need specific names to be used by the code, since the code looks up the tf-data using these names. By looking through the tfdata() function in the code, the code can instead be changed to accept different link/join names. This is a list of the only required frame-names, the other joints / links names are irrelevant as they are not used.

- **base_link**: The base of the robot.
- **forearm_link**: The link in the ”middle” of the arm.
- **wrist_2_link**: The middle of the end-link.

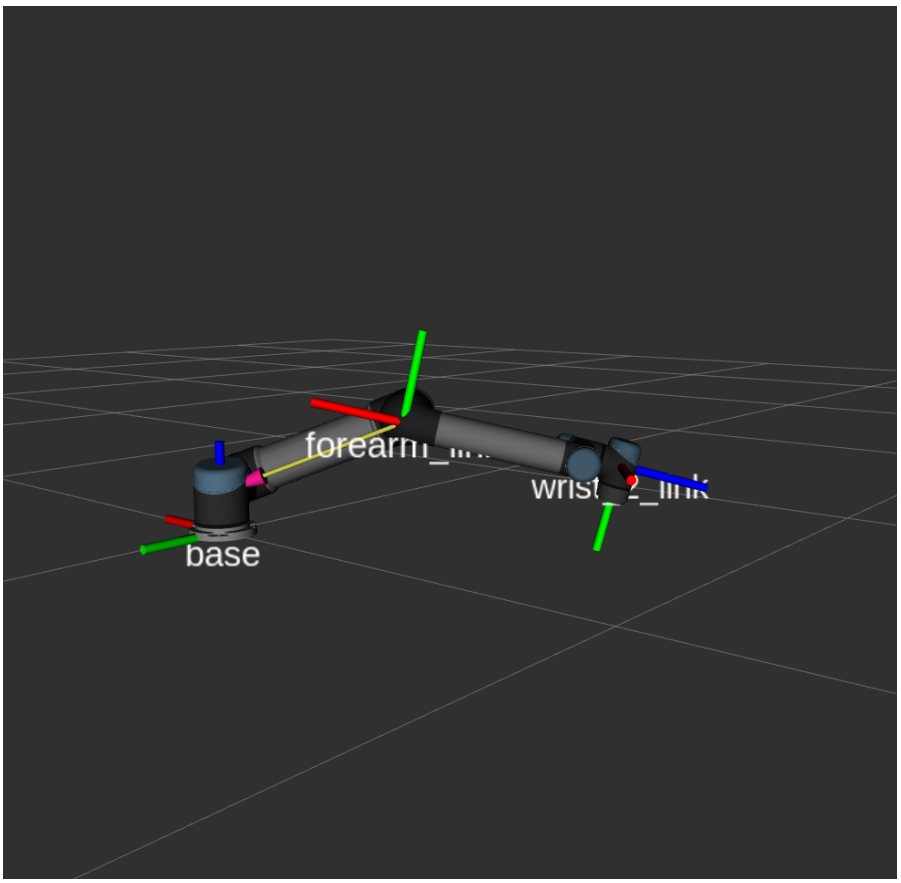


Figure 3.1: Robot transfer-frames illustration

In addition, the code itself will generate a ”camera_link”. The PointCloud data from the camera will be published on a topic, and the tf-data will be available (published) based on the configured position of the camera in the code for use in other applications if needed.

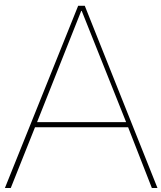
4

References

An example of a .yaml file needed for self-filtering, with predefined parameters that work with the accuracy of the D455 cameras used can be found in appendix: A.

Tutorial for setting up perception (sensors_3d.yaml) for self-filtering in MoveIt:

`http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/perception_pipeline/perception_pipeline_tutorial.html`



sensors_3d.yaml (Example)

Tutorial on setting up 3d perception can be found in 4.

This is the 3d perception file (”sensors_3d.yaml”) with the correctly defined topics for the use of 1 main and 2 secondary cameras:

```
sensors:
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
    point_cloud_topic: camera/depth/color/points
    max_range: 5
    point_subsample: 1
    max_update_rate: 10
    padding_offset: 0.1
    padding_scale: 1.2
    filtered_cloud_topic: filtered_cloud
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
    point_cloud_topic: secondary_camera_0
    max_range: 5
    point_subsample: 1
    max_update_rate: 1
    padding_offset: 0.2
    padding_scale: 1.2
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
    point_cloud_topic: secondary_camera_1
    max_range: 5
    point_subsample: 1
    max_update_rate: 1
    padding_offset: 0.2
    padding_scale: 1.2
```