Login (https://app.kth.se/kpm/auth/login?nextUrl=https://canvas.kth.se/courses/45204/pages/boot-camp-part-2?module_item_id=818048)

Boot Camp Part 2

Objective

For true autonomous operation, the robot needs to be able to localize itself. That is to say, starting from somewhere in the map, the robot should be able to find out where it actually is in the map frame using the ArUco markers that it has found given that the robot knows a where the markers are in the **map** frame, b) where the detected markers are in the **camera_link** frame, and c) where the camera is relative to the robot's **base_link**. Then, the transform between **map** and **odom** frames is simply the difference between the current pose in **map** and **odom** frame.

Prerequisite

Install Linux and ROS following the instructions here (https://canvas.kth.se/courses/45204/pages/linux-installation) .

Rosbag

In this part, we will use real sensor data from a robot similar to that you will have later in the course. **You can read more about recording and playing back data (rosbags) here (https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html)** .

Download the rosbag here (https://canvas.kth.se/courses/45204/files/7489639?wrap=1) (https://canvas.kth.se/courses/45204/files/7489639/download?download_frd=1) .

Unzip the file in the `~/dd2419_ws/bags` folder.

```
unzip <PATH_TO_FILE>/boot_camp.zip -d ~/dd2419_ws/bags/
```

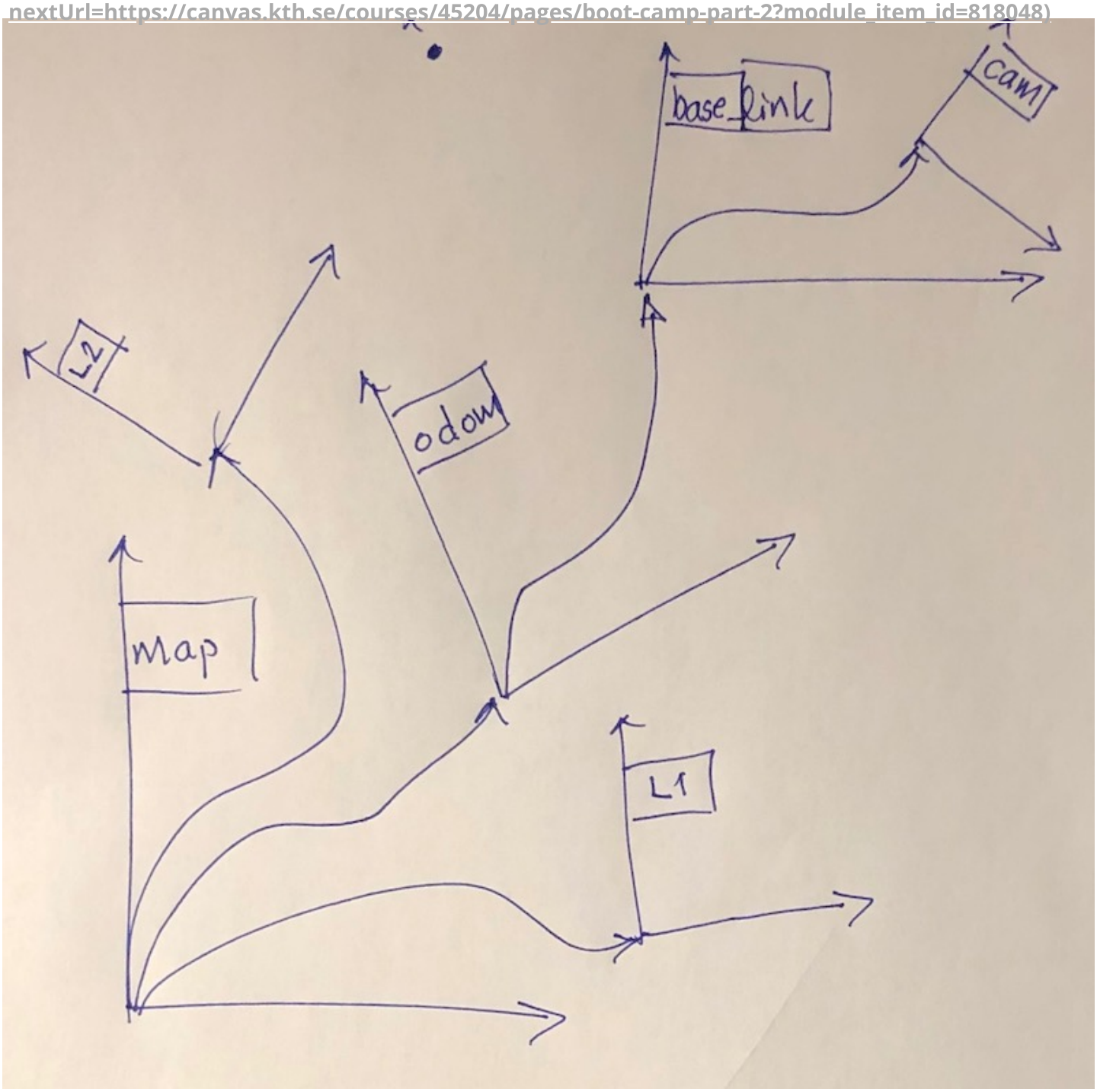
Transforms and Frames — TF

Let's take a look at how to deal with coordinate systems in ROS. A coordinate system is defined by a frame of reference. You might for example say that your computer is 50 cm from the left edge of your table, and 40 cm from the bottom edge. This coordinate system uses the table's lower left corner as its frame of reference (its origin), but clearly, you could choose any arbitrary reference frame. Perhaps it would make more sense to locate your laptop with respect to the northeast corner of the room you're in. If you then know where the table is in the room, it should be trivial to calculate the position of your laptop in the room -- this is exactly what TF does for you. More information about the

with tutorials here (https://docs.ros.org/en/humble/Tutorials/Intermediate/TF2/TF2-Main.html).

In mobile robotics there are three important *frames* that (almost) always are present when working with ROS: the **map** frame, considered the "absolute" coordinate system in that it doesn't change over time; the **odom** frame, for **odom**etry frame, whose origin typically is where the robot was powered on; and the **base_link** frame, which is the robot frame -- in other words, the robot is always at the origin of the **base_link** frame. REP 105 (http://www.ros.org/reps/rep-0105.html) defines these, and also gives some naming conventions and semantics, as well as some additional common frames. Coordinate frames convention can be found in REP 103 (https://www.ros.org/reps/rep-0103.html#coordinate-frame-conventions).

The picture below illustrates the relationship between the three frames mentioned above and some additional ones. A position x=(x, y, z) or a pose (x, y, z and three rotations about the body axes) can be expressed in any of the frames. However, one of them is usually more natural than the other. For example, your laptop is easier to locate with respect to the table than the room in the example above. In the image below, the location of the landmarks **L1** and **L2** are easier to express in the map frame, whereas the position of the **camera_link** is defined with respect to **base_link** (i.e. relative to the robot.) We can see from the graph that in order to know where the camera is in the map frame we also need to know where the robot is in the **odom** frame and the relation between the **odom** frame and the **map** frame. This requires a localization system, which estimates the pose of the robot in the map frame and therefore calculates the transform between map and odometry. We will see later how TF allows us to seamlessly move from frame to frame and thus make it possible to, for example, express the location of the landmarks in the **base_link** frame.



Question: What can you say about the transform from **map** and **odom** for a robot that has perfect odometry?

Odometry

Create a package called **odometry** and a node called **odometry** in it:

```
cd ~/dd2419_ws/src
ros2 pkg create odometry --build-type ament_python --node-name odometry --dependencies geometry_
msgs nav_msgs python3-numpy robp_interfaces rclpy tf_transformations tf2_ros --license MIT
```

```
cd ~/dd2419_ws
rosdep install --from-paths src -y --ignore-src --as-root pip:false
```

Build the new package

```
cd ~/dd2419_ws
colcon build --symlink-install
```

Source the workspace so the new package is included in your path

```
source ~/dd2419_ws/install/local_setup.bash
```

Edit the **odometry** node by editing the file `~/dd2419_ws/src/odometry/odometry/odometry.py`, below is a skeleton that you can use

```
#!/usr/bin/env python

import math

import numpy as np

import rclpy
from rclpy.node import Node

from tf2_ros import TransformBroadcaster
from tf_transformations import quaternion_from_euler, euler_from_quaternion

from geometry_msgs.msg import TransformStamped
from robp_interfaces.msg import Encoders
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped


class Odometry(Node):

    def __init__(self):
        super().__init__('odometry')

        # Initialize the transform broadcaster
        self._tf_broadcaster = TransformBroadcaster(self)

        # Initialize the path publisher
        self._path_pub = self.create_publisher(Path, 'path', 10)
        # Store the path here
        self._path = Path()

        # Subscribe to encoder topic and call callback function on each recieved message
        self.create_subscription(
            Encoders, '/motor/encoders', self.encoder_callback, 10)

        # 2D pose
        self._x = 0.0
        self._y = 0.0
        self._yaw = 0.0

    def encoder_callback(self, msg: Encoders):
        """Takes encoder readings and updates the odometry.

        This function is called every time the encoders are updated (i.e., when a message is pub
```

```
        Your task is to update the odometry based on the encoder data in `msg`. You are allowed
to add/change things outside this function.

        Keyword arguments:
        msg -- An encoders ROS message. To see more information about it
        run 'ros2 interface show robp_interfaces/msg/Encoders' in a terminal.
        """

        # The kinematic parameters for the differential configuration
        dt = 50 / 1000
        ticks_per_rev = 48 * 64
        wheel_radius = 0.0  # TODO: Fill in
        base = 0.0  # TODO: Fill in

        # Ticks since last message
        delta_ticks_left = msg.delta_encoder_left
        delta_ticks_right = msg.delta_encoder_right

        # TODO: Fill in

        self._x = self._x  # TODO: Fill in
        self._y = self._y  # TODO: Fill in
        self._yaw = self._yaw  # TODO: Fill in

        stamp = None # TODO: Fill in

        self.broadcast_transform(stamp, self._x, self._y, self._yaw)
        self.publish_path(stamp, self._x, self._y, self._yaw)

    def broadcast_transform(self, stamp, x, y, yaw):
        """Takes a 2D pose and broadcasts it as a ROS transform.

        Broadcasts a 3D transform with z, roll, and pitch all zero.
        The transform is stamped with the current time and is between the frames 'odom' -> 'base
_link'.

        Keyword arguments:
        stamp -- timestamp of the transform
        x -- x coordinate of the 2D pose
        y -- y coordinate of the 2D pose
        yaw -- yaw of the 2D pose (in radians)
        """

        t = TransformStamped()
        t.header.stamp = stamp
        t.header.frame_id = 'odom'
        t.child_frame_id = 'base_link'

        # The robot only exists in 2D, thus we set x and y translation
        # coordinates and set the z coordinate to 0
        t.transform.translation.x = x
        t.transform.translation.y = y
        t.transform.translation.z = 0.0

        # For the same reason, the robot can only rotate around one axis
        # and this why we set rotation in x and y to 0 and obtain
        # rotation in z axis from the message
        q = quaternion_from_euler(0.0, 0.0, yaw)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        # Send the transformation
        self._tf_broadcaster.sendTransform(t)
```

```
        Keyword arguments:
        stamp -- timestamp of the transform
        x -- x coordinate of the 2D pose
        y -- y coordinate of the 2D pose
        yaw -- yaw of the 2D pose (in radians)
        """

        self._path.header.stamp = stamp
        self._path.header.frame_id = 'odom'

        pose = PoseStamped()
        pose.header = self._path.header

        pose.pose.position.x = x
        pose.pose.position.y = y
        pose.pose.position.z = 0.01  # 1 cm up so it will be above ground level

        q = quaternion_from_euler(0.0, 0.0, yaw)
        pose.pose.orientation.x = q[0]
        pose.pose.orientation.y = q[1]
        pose.pose.orientation.z = q[2]
        pose.pose.orientation.w = q[3]

        self._path.poses.append(pose)

        self._path_pub.publish(self._path)


def main():
    rclpy.init()
    node = Odometry()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

For the boot camp, you can put the robot at some known starting location so that the transformation between **map** and **odom** is known in advance. This is obviously a brittle approach and one that will only work for a short time until the transformation between map and odom has changed too much due to drift, but it will suffice for demonstrating how to work with the different frames. In our case, the robot starts at (0, 0), with a yaw of 0 radians. Thus, the transform between the two frames is (0, 0) and yaw 0. You can publish this transform statically using the `static_transform_publisher` too directly on the command line. Open a terminal and run

```
ros2 run tf2_ros static_transform_publisher --frame-id map --child-frame-id odom
```

With this transform in place, it should now be possible to visualize the robot's odometric pose in the map frame.

not what is missing?

Question: What would happen if the robot didn't start with its X-axis aligned to the map's X-axis?

Running

You will need 4 terminals/"tabs", the order in which you run these are important.

Terminal 1

```
ros2 launch robp_boot_camp_launch boot_camp_part2_launch.xml
```

This will set the parameter */use_sim_time* to true, such that the time from the rosbag is used. This is important since otherwise, you will have problems with timestamps when running the rosbag.

It will also start RViz and publish a model for the camera.

Note that we added a blue square in RViz that should be roughly where the blue square is in the real world. This is to make it easier for you to see if your odometry is working.

Terminal 2

```
ros2 run tf2_ros static_transform_publisher --frame-id map --child-frame-id odom
```

To make sure you have a transform between **map** and **odom**.

Terminal 3

```
ros2 run odometry odometry
```

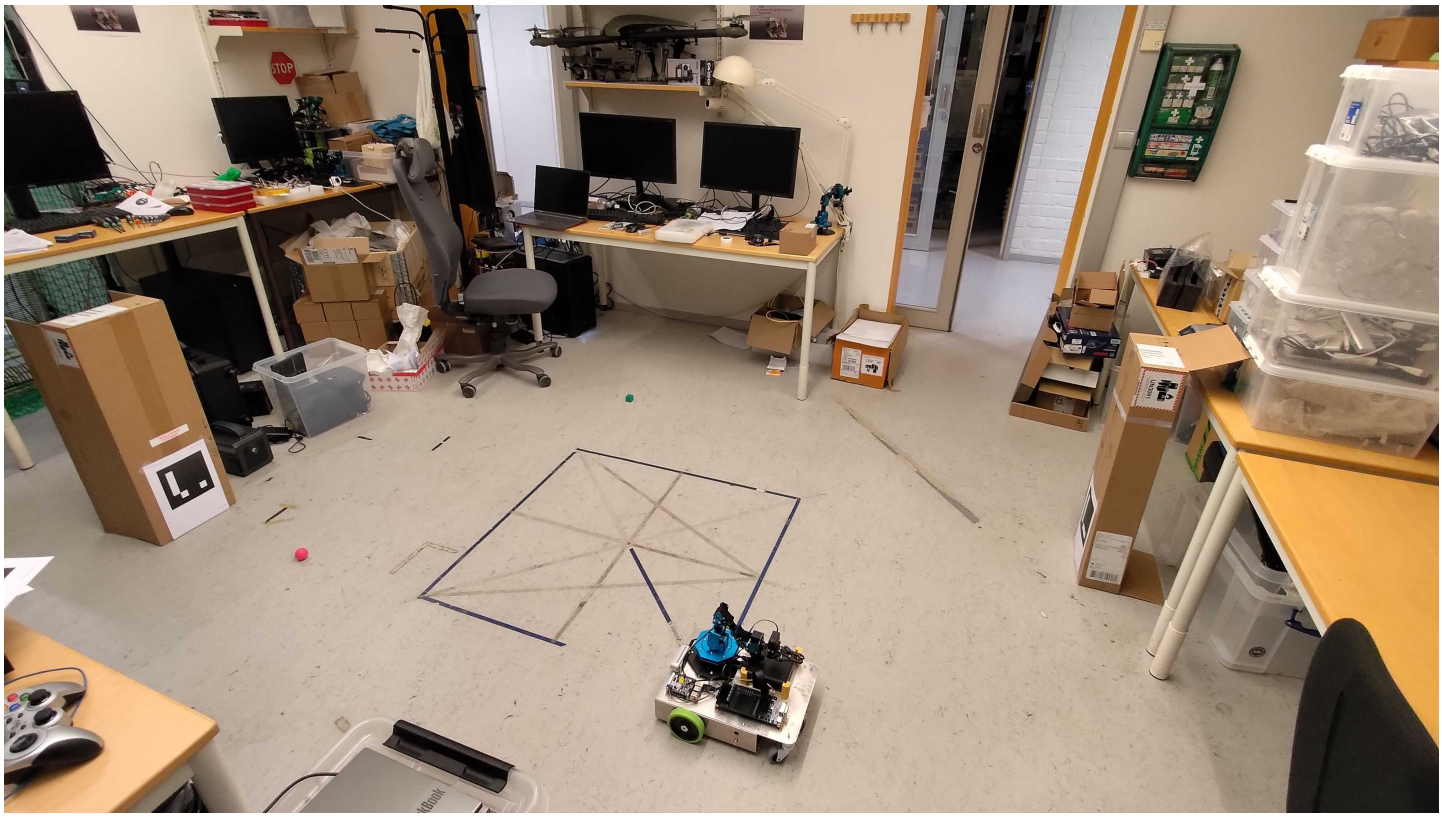
This will start your odometry node.

Terminal 4

```
ros2 bag play --read-ahead-queue-size 100 -l -r 1.0 --clock 100 --start-paused ~/dd2419_ws/bags/boot_camp
```

This will play the rosbag. The two options *--clock* and *--start-paused* are very important. *--clock* enables publishing of the clock time. *--start-paused* starts the rosbag in paused mode, this is to make sure that RViz and your nodes receive all the *static transforms* that are being published in the beginning. If you do not have these you will most likely have problems with timestamps. You press SPACE in this terminal to resume/pause the playback of the rosbag. To read what the optional arguments `--read-ahead-queue-size`, `-l`, `-r`, `--clock`, and `--start-paused`, do, you can run `ros2 bag play -h`.

Fill in the missing code for the odometry. The robot moves around in a 1x1 meter square three times. You can see the setup below. The robot moves along the blue lines.



Robot Kinematics

The kinematic parameters for the differential configuration are:

- Ticks per revolution: 48 * 64 = 3072
- Wheel radius (r): 0.04921 m
- Distance between the wheels, "base" (B): 0.3 m (this is the actual base (i.e., center of the wheels), but please experiment with different values to see if you can improve the performance)
- Update interval/rate: 50 ms (20 Hz)

Translation speed of each wheel given by $\boldsymbol{r} \cdot \frac{d\varphi}{dt}$
Translation and rotation speed of the robot as a function of the wheel speeds are given by

$$v = \frac{r}{2}(\dot{\varphi}_R + \dot{\varphi}_L) = \frac{r}{2}(d\varphi_R/dt + d\varphi_L/dt)$$
$$\omega = \frac{r}{B}(\dot{\varphi}_R - \dot{\varphi}_L) = \frac{r}{B}(d\varphi_R/dt - d\varphi_L/dt)$$

with φ_L and φ_R being the wheel rotation angles.

If K is the scaling factor to transform encoder difference to angle, $\Delta\varphi = K\Delta E$

$$v = \frac{r}{2}(K\Delta E_R + K\Delta E_L)/\Delta T \rightarrow D = v\Delta T = \frac{r}{2}(K\Delta E_R + K\Delta E_L)$$
$$\omega = \frac{r}{B}(K\Delta E_R - K\Delta E_L)/\Delta T \rightarrow \Delta\theta = \omega\Delta T = \frac{r}{B}(K\Delta E_R - K\Delta E_L)$$

Odometry

The so called odometry is given by integrating encoder measurements over time based on the kinematic model. The better the kinematic model, the better the odometry

$$x_{k+1} = x_k + v \cdot dt \cdot \cos(\theta) = x_k + D \cdot \cos(\theta)$$
$$y_{k+1} = y_k + v \cdot dt \cdot \sin(\theta) = y_k + D \cdot \sin(\theta)$$
$$\theta_{k+1} = \theta_k + \omega \cdot dt = \theta_k + \Delta\theta$$
$$\mathrm{pose}_{k+1} = \mathrm{f}_1(\mathrm{pose}_k, v, \omega, dt) = \mathrm{f}_2(\mathrm{pose}_k, D, \Delta\theta)$$

Expected Output

With $B = 0.3$ you can expect output similar to that in the video below:

The above is fine for passing the boot camp. But if you want to you can try to a better B. B will most likely be different on the robot you will have in the course, it can also "change" over time, so keep that in mind! In the video below you can see our optmized B, can you beat us?

Login (https://app.kth.se/kpm/auth/login?nextUrl=https://canvas.kth.se/courses/45204/pages/boot-camp-part-2?module_item_id=818048)

Note that the rosbag is run at 5x rate in both videos above.

ArUco Marker Detection

We have placed a number of ArUco markers in the world in order for you to be able to localize yourself. The ArUco markers have an orientation and an ID. Once you have run the `boot_camp_part2.launch` you should be able to listen to a topic called `/aruco/markers` which publishes the pose and ID of the markers that it detects.

NOTE: The rosbag has to be running for all the topics to show up!

Create a package called **display_markers** and a node called **display_markers** in it:

```
cd ~/dd2419_ws/src
ros2 pkg create display_markers --build-type ament_python --node-name display_markers --dependen
cies aruco_msgs geometry_msgs rclpy tf2_geometry_msgs tf2_ros --license MIT
```

Make sure all the dependencies are installed

```
cd ~/dd2419_ws
rosdep install --from-paths src -y --ignore-src --as-root pip:false
```

Edit the display_markers node by editing the file `~/dd2419_ws/src/display_markers/display_markers/display_markers.py`:

```
#!/usr/bin/env python

import rclpy
from rclpy.node import Node

from tf2_ros import TransformException
```

```
from tf2_ros import TransformBroadcaster
import tf2_geometry_msgs

from aruco_msgs.msg import MarkerArray
from geometry_msgs.msg import TransformStamped


class DisplayMarkers(Node) :

    def __init__(self) :
    super().__init__('display_markers')

    # Initialize the transform listener and assign it a buffer

    # Initialize the transform broadcaster

    # Subscribe to aruco marker topic and call callback function on each received message


    def aruco_callback(self, msg: MarkerArray) :

        # Broadcast/publish the transform between the map frame and the detected aruco marker


def main() :
    rclpy.init()
    node = DisplayMarkers()
    try :
        rclpy.spin(node)
    except KeyboardInterrupt :
        pass

    rclpy.shutdown()


if __name__ == '__main__':
main()
```

Task 2

The **display_markers** node should take the ArUco marker detections, transforms them into the map frame in a node, and publishes a TF from the *map* frame to a */aruco/detectedX* frame, where X corresponds to the ID of the marker. To see the message type on the */aruco/markers* topic you can run `ros2 topic info /aruco/markers` when you are running the rosbag. When you know the message type you can run `ros2 interface show [MESSAGE_TYPE]` to see how the message is structured.

The camera is mounted **0.08987m** forward, **0.0175m** left, and **0.10456m** up with respect to the robot. Note that the camera coordinate system is such that z is along the optical axis (i.e., forward) and y is pointing down.

You have to run your odometry as well for the detected markers to be placed in the correct place since you have to transform them into the map frame, so you need the transformation between odom and base_link.

ones in the launch file. So you need to either rotate your markers in your code or change the orientation in the launch file. Either is fine! Remember to build your workspace in case you change the launch file to update the installed launch file!

```
cd ~/dd2419_ws
colcon build --symlink-install
```

Running

You will need the 4 terminals/"tabs" from task 1 and also.

```
ros2 run display_markers display_markers
```

Expected Output

With the odometry version $B = 0.3$ above, this is the expected output of the marker detection:

With the odometry version with optimized B you can expect the behavior shown below:

Login (https://app.kth.se/kpm/auth/login?nextUrl=https://canvas.kth.se/courses/45204/pages/boot-camp-part-2?module_item_id=818048)

Note that the rosbag is running at 4x rate in both videos.

Tips

- Use radians not degrees for the kinematic calculations.
- Look at the TF tree in rqt to see what transforms are already available for you. Remember to launch rqt before you start the rosbag. The TF tree will be populated after you have unpaused the rosbag (see this page for more info on this **/courses/45204/pages/nice-and-mixed-gott-och-blandat (https://canvas.kth.se/courses/45204/pages/nice-and-mixed-gott-och-blandat)** , scroll down to "Rosbags and static transformations").