Localization Module for ROAR Autonomous Driving System

Intro & Goal

ROAR stands for Robot Open Autonomous Racing, which was founded by the FHL Vive Center for Enhanced Reality at UC Berkeley in 2019. The goal of ROAR is promoting cutting-edge machine learning, control, and autonomous driving research via an exciting and affordable ground vehicle hardware.

The reference design of the ROAR hardware (the Vehicle) includes a forward-facing Intel Realsense D435i depth camera, which will be used for image input to calculate the location of the Vehicle. Although D435i provides both imagery and IMU data input. In this Localization Module, only the imagery data will be utilized. Nevertheless, the IMU data can be still utilized by the Vehicle controllers and other modules.

Our solution to localize the Vehicle consists of two parts. The first part is a digital map standard in JSON format, which is capable of precisely describing the physical layout of a race track ("the Track"). The map utilizes AR markers to serve as physical landmarks that will be installed on the Track to indicate 3D spatial transformations between the Vehicle and the Track.

The second part describes how to calculate the Vehicle's position and orientation with one or many AR markers visible or with no AR marker visible. The goal is to provide reliable estimation of the rotation and translation matrices of the car relative to the global map.

Who is this for?

The Module develops a simplified but sufficiently fast and robust localization method for ground vehicles to rely solely on detecting AR markers from imagery input.

Given a race track, the module will first assume it consists of two types of segments: straight-lines and turns. A digital map of the track then will be created and documented in JSON format to describe the length, width, and radius, etc.

Then, with respect to each segment, one or more AR markers need to be instrumented in the track environment and their absolute rotation and translation position in the track will be also added into the track map in JSON.

Finally, a ground vehicle will rely on a single depth camera to observe the track and the installed AR markers, and can then use this module to estimate its absolute rotation and translation position in real time reliably.

Users who agree with the above race track standard and localization procedure can adopt this module to localize their vehicles.

Why build it?

Localization based on road conditions and environmental landmarks is a critical function of ground-based autonomous driving systems. Yet, commercial localization and mapping solutions are known to suffer from the complexity of the road condition and the wide variety of possible landmarks.

To mitigate these challenges, our approach with the end goal of providing a reliable and real-time software solution is not to develop an overly complicated algorithm to handle all possible road conditions, but to seek the most well-used algorithms to apply to a simplified and standardized race track. To this end, the use of AR marker as unique landmarks is well known to be reliable in estimating relative spatial transformation between the landmarks and the camera.

The localization module will be extremely light weight in taxing the computation resources on the mobile computer system onboard the 1/10 RC cars. Also because our localization algorithm only relies on imagery input from a single depth camera, it is complementary to other additional localization solutions, such as relying on lane tracking and IMU inputs. Therefore, end users may choose to combine this module with other alternative modules without much challenges.

In addition, the localization will support both an infrared and rgb image input, specified by the user elsewhere in the code, in case the user wants to use one or the other for performance enhancements.

Lastly, the localization lets a user put in coordinates in terms of x y and z and in terms of eulerian angles, and then internally converts it to a rotation matrix and translation matrix horizontally stacked together (called the config matrix) which the user can interpret in code.

What is it?

The Localization Module defines two components: a **JSON Standard** and a **Localization Class**.

The **JSON file** establishes a map of the track. Below is a template for a track. As seen below, the JSON file has three major sections:

1. *Comment*
2. *AR parameters*

3. *Segments*
4. *AR markers*

The *Comment* is provided to help users to understand the formatting of the JSON file and the information contained in each segment of the file.

The *AR parameters* provide information for the bits of information provided by the AR markers and the number of AR markers in the dictionary. The comment in this section provides more information on how to setup the AR markers.

The *Segments* describe each section of the race track. *Segments* contain an array of individual segments of the track. A segment of the track has 7 attributes which are: Angle, Radius, Length, Width, Start, End, and AR Id.

- Angle represents the angle of turn for that section of track. If the angle is 0, it can be assumed that the track is straight. Likewise an angle of 90 indicates a 90 degree turn on the track. Units: Degrees
- Radius is specific to a turn. If there is a turn then the radius represents the radius of that turn. If the segment of track is straight the radius should be set to 0. Units: Meters
- Length represents the length of the segment of track. For turns this should be equal to Angle * Radius, where the Angle is converted from degrees to radians. Units: Meters
- Width is the width of the track. Units: Meters
- Start represents the starting [x, y, z, roll, pitch, yaw] of the segment of track.
- End represents the ending [x, y, z, roll, pitch, yaw] of the segment of track.
- AR Id is an array of the ids of all the AR markers found on the selected section of the track.

The *AR tags* segment contains an array of the AR markers. Each AR marker has 3 attributes. The first is an integer which is the id of the AR marker. The second attribute is the location of the AR marker relative to the global map. The location has three parameters: x, y, z, roll, pitch, and yaw (angle) of the AR marker. The final attribute of an AR marker is the segment of the track (*“Segments”*) that the AR marker is found on. “Segment” is the index of *Segments* at which the AR marker is located.

```
{
  "Comment": "This JSON has 4 main components: Comment, AR parameters, Segments,
and AR Tags. The AR parameters provide information on the size, dimension, and
setup of the AR tags. Segments contain an array of segments of the track, which
are ordered by index in an array. Segment 0 is located at Segment[0]. Each segment
has 7 parts whose units are in meters and degrees. The Angle and Radius correspond
to the value of a turn, if the segment is a turn. The Length is the length of the
segment in meters. The Width of the track corresponds to the width from the
centerline of the segment to one edge of the track. The Start and End coordinates
are stored as arrays and have 6 values: [x,y,z,roll,pitch,yaw]. Units for the
values are in meters and degrees. The final value for a segment is an array called
```

```
AR Id and contains the integer ID of all the AR tags in that segment. The last
component is AR Tags which stores an array with the values of individual AR tags.
A singular AR tag has the fields: Id, Location, and Segment. The Id is an integer
corresponding to the Id of the AR tag. The Location is an array with 6 values
corresponding to the [x,y,z,roll,pitch,yaw] of the tag relative to the map.
Segment is an integer value for the index in Segments that the AR tag is found
on.",
  "AR parameters" : {
         "Comment": "The width represents the width of the AR tags in centimeters.
The margin is the square white space surrounding the AR tag in centimeters.
Dimension is marker size in bits (y). Size in the number of markers in that
dictionary (z). The corresponding aruco dictionary can be found by running the
command: aruco.Dictionary_get(aruco.DICT_<<y>>X<<y>>_<<z>>)",
         "Width": 12,
         "Margin": 0.5,
         "Dimension": 6,
         "Size": 250
  },
  "Segments": [{
                "Angle": 0,
                "Radius": 0,
                "Length": 10,
                "Width": 0.5,
                "Start": [0,0,0,0,0,0],
                "End": [10,0,0,0,0,0],
                "AR Id": [1,2]
         },{
                "Angle": 90,
                "Radius": 2.82,
                "Length": 4.429656,
                "Width": 0.5,
                "Start": [10,0,0,0,0,0],
                "End": [12,2,0,0,0,90],
                "AR Id": [3]
         }],
  "AR tags": [{
         "Id": 1,
         "Location": [4, 0.3, 0,0,0,0],
         "Segment": 0
  },
  {
         "Id": 2,
         "Location": [9, -0.3,0,0,0,0,0],
         "Segment": 0
  },
  {
         "Id": 3,
         "Location": [11.414, 0.886, 0, 0, 0, 45],
         "Segment": 1
  }]
```

}

The **Localization Class** takes in the JSON map and the D435i image of the current frame from the car and using the two find the location of the car relative to the map. This class is compatible with the DonkeyCar interface as well as BARC. This class will contain a run_threaded function which will take in the image array from the RealSense camera and return the current localization within the map based on AR markers within the image. Later, this part will have an additional fix where in the event a AR marker isn't present, we will use image keypoint stitching to find our change in position

Class Localization:

Attributes:

json_in : str

Will be the JSON file location for the map

map : dictionary

calls and saves information from json.loads(json_in)

AR_loc : array of AR markers from JSON str

Methods:

__init__(str json_in)

run_threaded(D435i image):

Threaded func that preprocesses image, calls get_position_list, then uses avg_AR to get the global weight average position estimate based on distance from camera. (AR_detected = True)

If the position_list is empty, then use keyframe_matching to see if you can establish position based on previous image to current image relation, and shift this matrix to current coordinates if possible. (AR_detected = False)

Return config matrix, AR_detected

get_position_list(D435i image)

Returns list of config matrices representing the global position of the car on the map based on each aruco marker

Returns None if no markers detected

avg_ARs(list of estimated global positions based AR markers seen and their distances from the camera)

Returns the weighted average of the global interpolated translation and global interpolated rotation

keyframe_matching(new image)

Returns the config matrix transformations between the previous image and the new image (returns null value if previous matrix isn't available)

Using class:
Assuming V is your vehicle class:
“V.add(class_instance_name, inputs=[‘cam/image_array’], outputs=[‘map/config’, ‘map/AR_detected’], threaded=True)”

Any other class that wants to use the config values can add itself to V in the same way and specify ‘map/config’ as the inputs.
e.g. “V.add(example, inputs=[‘map/config’, ‘map/AR_detected’], outputs=[‘user/example’], threaded=True)”

Implementation for the Localization Class

First, we use the CV ARUCO functions to extract the AR markers and then the rotation/translation info from them. If there are no AR Markers, we either use keyframe matching to get a relative location and angle to our previous location, or return a NULL value if this is impossible.
After this is done, we will have to reference the global config matrix of each detected AR marker and then determine our global interpolated translation and global interpolated rotation from each of those. This will use a weighted average based off of distance of the AR markers from the car.

Once we have our averaged config matrix, we will return it.

Constraints and Solutions

If the car’s camera detects one AR the implementation above can be easily followed.

If multiple AR markers are seen, the same implementation is followed. Translation is calculated using Weighted Linear Interpolation and rotation is calculated using Weighted Spherical Interpolation.

If no AR markers can be found and the car has at least seen one AR marker, keypoint matching can be used on consecutive images to figure out our location relative to that of the last seen AR marker image. This will reset once an AR marker is seen. If we have no original frame of reference to an AR marker or the image stitching fails for some reason, return NULL.