B. Aditya Naidu

BITS PILANI

GUIDE: SUMIT VEERAWAL

MAPPING AND LOCALIZATION BASED ON Aruco markers

PROJECT REPORT

**ACKNOWLEDGEMENTS**

This was my first exposure to one of the premier research establishments in the country and the last one month that I spent here has been quite an enlightening experience and has helped in broadening my horizon. I have been able to closely observe the way research laboratories in our country function and also got an opportunity to work alongside the best scientific brains in the country.

I am deeply indebted to my guide Shri. Sumit Veerawal for providing me with an opportunity to do my internship in this premier research institute.

I would like to thank a few other people in CAIR for constantly helping me out with any problems that I might have faced.

# **Problem Statement**

Designing and testing algorithms for mapping and localization based on ArUco markers.

# **Introduction**

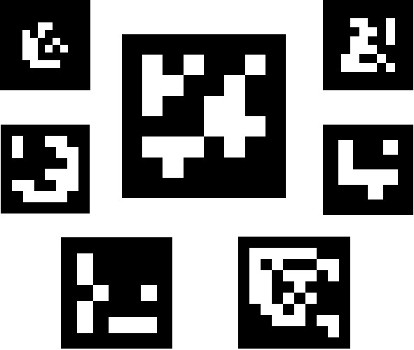
Applications that use code markers (barcodes, QR codes, etc.) have become ubiquitous following the popularity of such codes and our need to connect physical objects with corresponding digital identifiers. By learning how to implement one of these algorithms, one gains a deeper understanding of the considerations that go into designing these code markers and the algorithms that detect them.

Aruco markers consist of a seven-by-seven binary grid. Each cell is called a bit. A dictionary of Aruco markers consists of several such markers. To detect a marker in a photo or video stream, one must first find the corners of the marker, correct perspective distortion to obtain an image that looks as if the marker were being seen from above, divide the resulting image into a grid, and compare this grid of white and black cells with the given dictionary to find out if there is a match.

When building marker dictionaries, the goal is to make the markers distinguishable from each other to decrease the risk of misidentifying a marker in the dictionary for another. Unlike some other markers, Aruco markers are designed with an algorithm which can be mathematically shown to optimize the distance between markers, meaning it minimizes the chance of misidentifying a marker for another if one or a few bits are not properly recognized. Furthermore, the algorithm avoids markers that are either close to all black or close to all white, as such markers can be confused for other objects in the image.

# **ArUco Markers**

An ArUco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques. The marker size determines the size of the internal matrix. For instance, a marker size of 4x4 is composed by 16 bits.



It must be noted that a marker can be found rotated in the environment, however, the detection process needs to be able to determine its original rotation, so that each corner is identified unequivocally. This is also done based on the binary codification.

A dictionary of markers is the set of markers that are considered in a specific application. It is simply the list of binary codifications of each of its markers.

The main properties of a dictionary are the dictionary size and the marker size.

* The dictionary size is the number of markers that compose the dictionary.
* The marker size is the size of those markers (the number of bits).

The aruco module includes some predefined dictionaries covering a range of different dictionary sizes and marker sizes.

One may think that the marker id is the number obtained from converting the binary codification to a decimal base number. However, this is not possible since for high marker sizes the number of bits is too high and managing such huge numbers is not practical. Instead, a marker id is simply the marker index within the dictionary it belongs to. For instance, the first 5 markers in a dictionary have the ids: 0, 1, 2, 3 and 4.

# **Marker Detection**

Given an image containing ArUco markers, the detection process has to return a list of detected markers. Each detected marker includes:

* The position of its four corners in the image (in their original order).
* The id of the marker.

The marker detection process is comprised of two main steps:

1. Detection of marker candidates. In this step the image is analyzed in order to find square shapes that are candidates to be markers. It begins with an adaptive thresholding to segment the markers, then contours are extracted from the thresholded image and those that are not convex or do not approximate to a square shape are discarded. Some extra filtering is also applied (removing contours that are too small or too big, removing contours too close to each other, etc).
2. After the candidate detection, it is necessary to determine if they are actually markers by analyzing their inner codification. This step starts by extracting the marker bits of each marker. To do so, a perspective transformation is first applied to obtain the marker in its canonical form. Then, the canonical image is thresholded using Otsu to separate white and black bits. The image is divided into different cells according to the marker size and the border size. Then the number of black or white pixels in each cell is counted to determine if it is a white or a black bit. Finally, the bits are analyzed to determine if the marker belongs to the specific dictionary. Error correction techniques are employed when necessary.

## Thresholding

* One of the first steps of the marker detection process is an adaptive thresholding of the input image.

A board game on a table

Description automatically generated with low confidence

* For instance, the thresholded image for the sample image used above is:

A picture containing diagram

Description automatically generated

## Contour filtering

* After thresholding, contours are detected. However, not all contours are considered as marker candidates. They are filtered out in different steps so that contours that are very unlikely to be markers are discarded. The parameters in this section customize this filtering process.
* It must be noted that in most cases it is a question of balance between detection capacity and performance. All the considered contours will be processed in the following stages, which usually have a higher computational cost. So, it is preferred to discard invalid candidates in this stage than in the later stages.
* On the other hand, if the filtering conditions are too strict, the real marker contours could be discarded and, hence, not detected.

## Bits Extraction

* After candidate detection, the bits of each candidate are analyzed in order to determine if they are markers or not.
* Before analyzing the binary code, itself, the bits need to be extracted. To do so, the perspective distortion is removed, and the resulting image is thresholded using Otsu threshold to separate black and white pixels.

## Marker identification

* After the bits have been extracted, the next step is checking whether the extracted code belongs to the marker dictionary and, if necessary, error correction can be performed.

## Corner Refinement

* After markers have been detected and identified, the last step is performing subpixel refinement of the corner positions
* Note that this step is optional and it only makes sense if the positions of the marker corners have to be accurate, for instance for pose estimation. It is usually a time-consuming step and therefore is disabled by default.

# **Pose Estimation**

The next thing you probably want to do after detecting the markers is to obtain the camera pose from them.

To perform camera, pose estimation you need to know the calibration parameters of your camera. These are the camera matrix and distortion coefficients. If you do not know how to calibrate your camera, you can take a look at the [**calibrateCamera()**](https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga3207604e4b1a1758aa66acb6ed5aa65d) function and the Calibration tutorial of OpenCV. You can also calibrate your camera using the aruco module as explained in the **Calibration with ArUco and ChArUco** tutorial. Note that this only needs to be done once unless the camera optics are modified (for instance changing its focus).

In the end, what you get after the calibration is the camera matrix: a matrix of 3x3 elements with the focal distances and the camera center coordinates (a.k.a intrinsic parameters), and the distortion coefficients: a vector of 5 or more elements that models the distortion produced by your camera.

When you estimate the pose with ArUco markers, you can estimate the pose of each marker individually. If you want to estimate one pose from a set of markers, use ArUco Boards (see the **Detection of ArUco Boards** tutorial). Using ArUco boards instead of single markers allows some markers to be occluded.

The camera pose with respect to a marker is the 3d transformation from the marker coordinate system to the camera coordinate system. It is specified by rotation and translation vectors.

# **Selecting a dictionary**

The aruco module provides the Dictionary class to represent a dictionary of markers.

In addition to the marker size and the number of markers in the dictionary, there is another important dictionary parameter, the inter-marker distance. The inter-marker distance is the minimum distance among its markers and it determines the error detection and correction capabilities of the dictionary.

In general, lower dictionary sizes and higher marker sizes increase the inter-marker distance and vice-versa. However, the detection of markers with higher sizes is more complex, due to the higher number of bits that need to be extracted from the image.

For instance, if you need only 10 markers in your application, it is better to use a dictionary composed only of those 10 markers than using a dictionary composed of 1000 markers. The reason is that the dictionary composed of 10 markers will have a higher inter-marker distance and, thus, it will be more robust to errors.

Three ways of selecting a dictionary:

* 1. Predefined dictionaries
  2. Automatic dictionary generation
  3. Manual dictionary generation

# **Kalman Filter**

Kalman filtering uses a system's dynamic model (e.g., physical laws of motion), known control inputs to that system, and multiple sequential measurements (such as from sensors) to form an estimate of the system's varying quantities (its state) that is better than the estimate obtained by using only one measurement alone. As such, it is a common sensor fusion and data fusion algorithm.

Noisy sensor data, approximations in the equations that describe the system evolution, and external factors that are not accounted for all limit how well it is possible to determine the system's state. The Kalman filter deals effectively with the uncertainty due to noisy sensor data and, to some extent, with random external factors. The Kalman filter produces an estimate of the state of the system as an average of the system's predicted state and of the new measurement using a weighted average. The purpose of the weights is that values with better (i.e., smaller) estimated uncertainty are "trusted" more. The weights are calculated from the covariance, a measure of the estimated uncertainty of the prediction of the system's state. The result of the weighted average is a new state estimate that lies between the predicted and measured state, and has a better estimated uncertainty than either alone. This process is repeated at every time step, with the new estimate and its covariance informing the prediction used in the following iteration. This means that Kalman filter works recursively and requires only the last "best guess", rather than the entire history, of a system's state to calculate a new state.

The measurements' certainty-grading and current-state estimate are important considerations. It is common to discuss the filter's response in terms of the Kalman filter's gain. The Kalman-gain is the weight given to the measurements and current-state estimate and can be "tuned" to achieve a particular performance. With a high gain, the filter places more weight on the most recent measurements, and thus conforms to them more responsively. With a low gain, the filter conforms to the model predictions more closely. At the extremes, a high gain close to one will result in a jumpier estimated trajectory, while a low gain close to zero will smooth out noise but decrease the responsiveness.

When performing the actual calculations for the filter (as discussed below), the state estimate and covariances are coded into matrices because of the multiple dimensions involved in a single set of calculations. This allows for a representation of linear relationships between different state variables (such as position, velocity, and acceleration) in any of the transition models or covariances.

# **Dependencies**

This software is built on the Robotic Operating System (ROS), which needs to be installed first. It also requires OpenCV 3.3+.

# **Usage**

## Marker placement

* 1. first marker is identified with world´s origin
  2. second marker's position is computed with respect to first one
  3. third's marker position is computed with respect to second one

In order to achieve successful mapping, every time when a new marker is detected, previous marker needs to be visible in the actual image to allow computing new marker's position.

## Camera calibration

1. Proper camera calibration strongly affects overall system accuracy.
2. Aruco mapping expects calibration file in Videre INI format

## Mapping

* First step is to create a map:

roslaunch tf\_mapping start\_tf\_mapping.launch

* Store the tf's and the images using:

rosrun tf\_mapping tf\_create\_map.py

* Adjust full\_hd resolution (or the highes resolution for creating the map) in tf\_mapping/launch/start\_tf\_mapping.launch
* Point the camera such that only the aruco marker with id=0 is detected (This will be your initial coordinate system).
* The transformations of all other detected aruco markers will be relative to this initial system.
* Point the camera to another aruco marker e.g. with id=1
* When moving the camera consider that always 2 markers are visible on every image.

## Localization

* Navigate your camera now in this map
* Adjust to use now lower resolution e.g. 640x480 (to be realtime capable) in tf\_mapping/launch/start\_tf\_navigate.launch

roslaunch tf\_mapping start\_tf\_navigate.launch

# Conclusion

In this report I discussed the usage of ArUco markers for mapping and localization using a camera. I also discussed the process of detection of ArUco markers, pose estimation as well as saving the map. The mapping needs to be done with the utmost care so as to avoid and errors. I also discussed the basics of Kalman Filtering to predict the uncertainty in the pose estimation.

# References

1. https://pyimagesearch.com/2020/12/21/detecting-aruco-markers-with-opencv-and-python/
2. https://medium.com/@calle\_4729/using-mathematica-to-detect-aruco-markers-197410223f62
3. https://docs.opencv.org/4.x/d5/dae/tutorial\_aruco\_detection.html
4. http://wiki.ros.org/aruco\_mapping