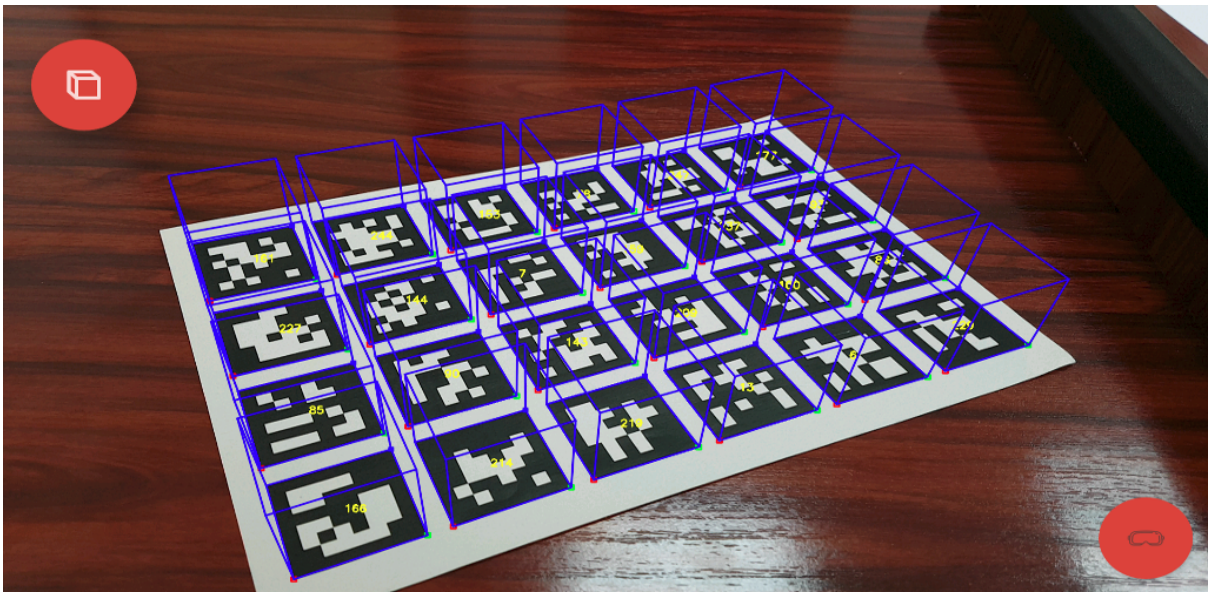
ArUco: An efficient library for detection of planar markers and camera pose estimation

News: Check out our latest project UcoSLAM
Markers + Keypoints!



Author: Rafael Muñoz Salinas

email:rmsalinas@uco.es

License and how to cite

The library is released under GPLv3. Please cite if you use it for research:

Main Paper of Aruco version 3:
"Speeded Up Detection of Squared Fiducial Markers"
https://www.sciencedirect.com/science/article/pii/S0262885618300799
[DOWNLOAD PDF]
[DOWNLOAD Bibtex]

Paper that started Aruco project:
"Automatic generation and detection of highly reliable fiducial markers under occlusion"
http://www.sciencedirect.com/science/article/pii/S0031320314000235
[DOWNLOAD PDF]
[DOWNLOAD Bibtex]

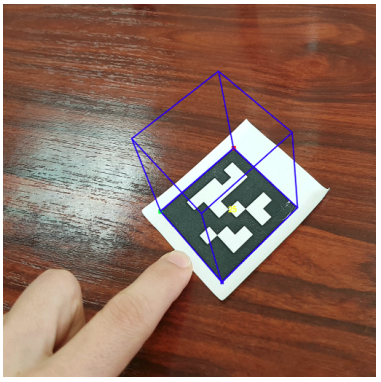
Paper explaining the default dictionary of markers:
"Generation of fiducial marker dictionaries using mixed integer linear programming"
http://www.sciencedirect.com/science/article/pii/S0031320315003544
[DOWNLOAD PDF]
[DOWNLOAD Bibtex]

Getting Support

If you need help with the library, please send an email to Rafael Muñoz Salinas: rmsalinas@uco.es

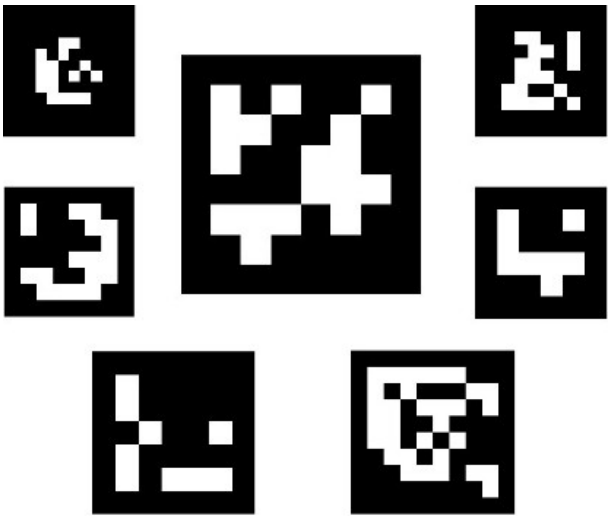
Introduction

ArUco is an OpenSource library for detecting squared fiducial markers in images. Additionally, if the camera is calibrated, you can estimate the pose of the camera with respect to the markers.



The library is written in C++, but there tools for using the library without programming. You can download both the library sources and precompiled binaries for windows.

There are several type of markers, each of them belonging to a *dictionary*. Each library has proposed its own set of markers. So, we have ArToolKit+, Chilitags, AprilTags, and of course, ArUco dictionary. The design of a dictionary is important since the idea is that their markers should be as different as possible to avoid confusions. We propose to use the dictionary *ARUCO_MIP_36h12* which we developed in a research paper. However, our library can detect the dictionaries of other libraries such as: Chiltags, AprilTags, ARToolKit+.



So, the first step is to download our dictionary, print one of two markers in a piece of paper and take a picture of it with your camera. Alternatively, you can just show them in your screen and take a picture.

As already indicated, the library is written in C++ and requires OpenCv. Let us see the most simple example in which the camera is not calibrated:

```
#include "aruco.h"
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
 int main(int argc,char **argv){
   if (argc != 2 ){ std::cerr<<"Usage: inimage"<<std::endl;return -1;}
   cv::Mat image=cv::imread(argv[1]);
   aruco::MarkerDetector MDetector;
   //detect
   std::vector<aruco::Marker> markers=MDetector.detect(image);
   //print info to console
   for(size_t i=0;i<markers.size();i++){
       std::cout<<markers[i]<<std::endl;
      //draw in the image
       markers[i].draw(image);
```

```
    }
    cv::imshow("image",image);
    cv::waitKey(0);
}
```

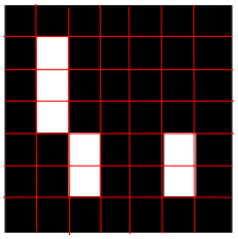
ArUco uses the class *Marker,* representing a marker observed in the image. Each marker is a vector of 4 points (representing the corners in the image), a unique id, its size (in meters), and the translation and rotation that relates the center of the marker and the camera location. In the above example, you can see that the task of detecting the markers is the class *MarkerDetector*. It is prepared to detect markers of any of the Dictionaries allowed. By default, the *MarkerDetector* will look for squares and then will analyze the binary code inside. For the code extracted, it will compare against all the markers in all the available dictionaries. This is however not the best approach. Ideally, you should explicitly indicate the type of dictionary you are using. The list of possible dictionaries is in Dictionary::DICT_TYPES.

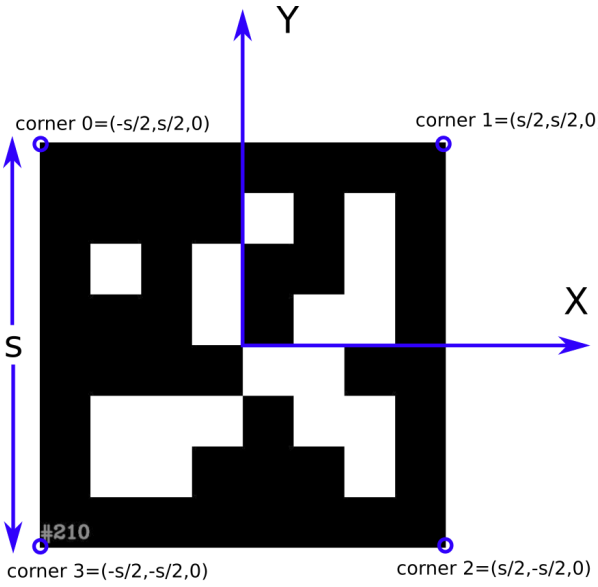
As previously indicated, we recommend the *ARUCO_MIP_36h12* dictionary. So, if you explicitly indicate the dictionary you are using, you avoid undesired errors and confusions with other Dictionaries. Thus, before calling detect, you should call:

```
MDetector.setDictionary("ARUCO_MIP_36h12");
```

Markers

Markers are composed by an external black border and an inner region that encodes a binary pattern. The binary pattern is unique and identifies each marker. Depending on the dictionary, there are markers with more or fewer bits. The more bits, the more words in the dictionary, and the smaller the chance of confusion. However, more bits means that more resolution is required for correct detection. The reader interested in knowing how marker is created, please check the following research paper.





s= marker size

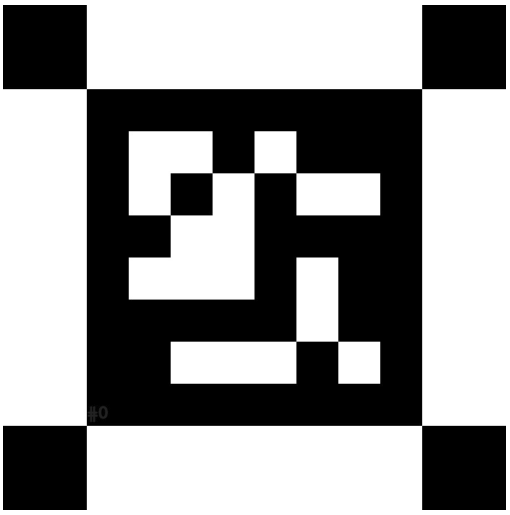
Markers can be used as 3D landmarks for camera pose estimation. We denote *s* the size of the marker once it is printed on a piece of paper. The following image shows the coordinate system employed in our library.

Enclosed Markers

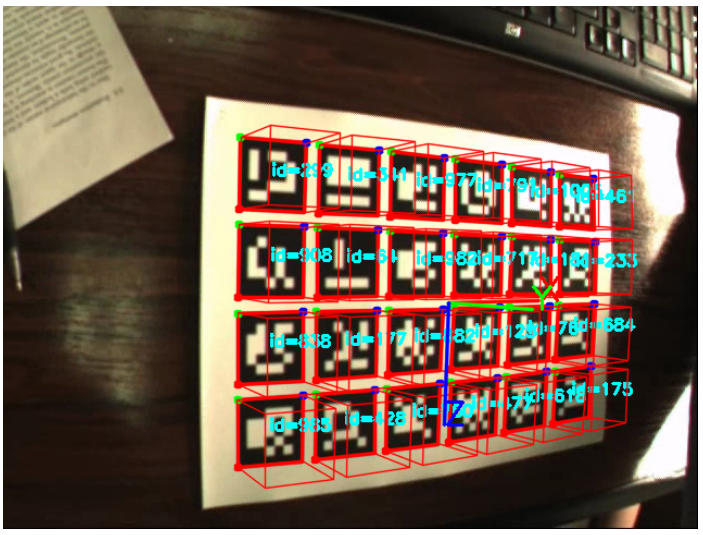
A precise corner localization is essential for camera pose estimation purposes. The final step in the detection of markers is to accurately estimate the corner with subpixel accuracy. Some algorithms for subpixel

accuracy (such as the one implemented in the OpenCv library) works better when the corner is enclosed as shown in the right image.
The library calls these, enclosed markers, and its detection is slightly more difficult than for normal markers.
Enclosed markers can be generated using the program in *utils/aruco_print_marker* using the option -e. Then, when using the library, make sure you enable the option

```
MarkerDetector::getParameters().detectEnclosedMarkers(true);
```



Marker Maps



Detecting a single marker might fail for different reasons such as poor lighting conditions, fast camera movement, occlusions, etc. To overcome that problem, ArUco allows the use of maps of markers. Marker map is composed by several markers in known locations. Marker maps present two main advantages. First, since there have more than one markers, it is less likely to lose them all at the same time. Second, the more markers are detected, the more points are available for computing the camera extrinsics. As a consequence, the accuracy obtained increases.

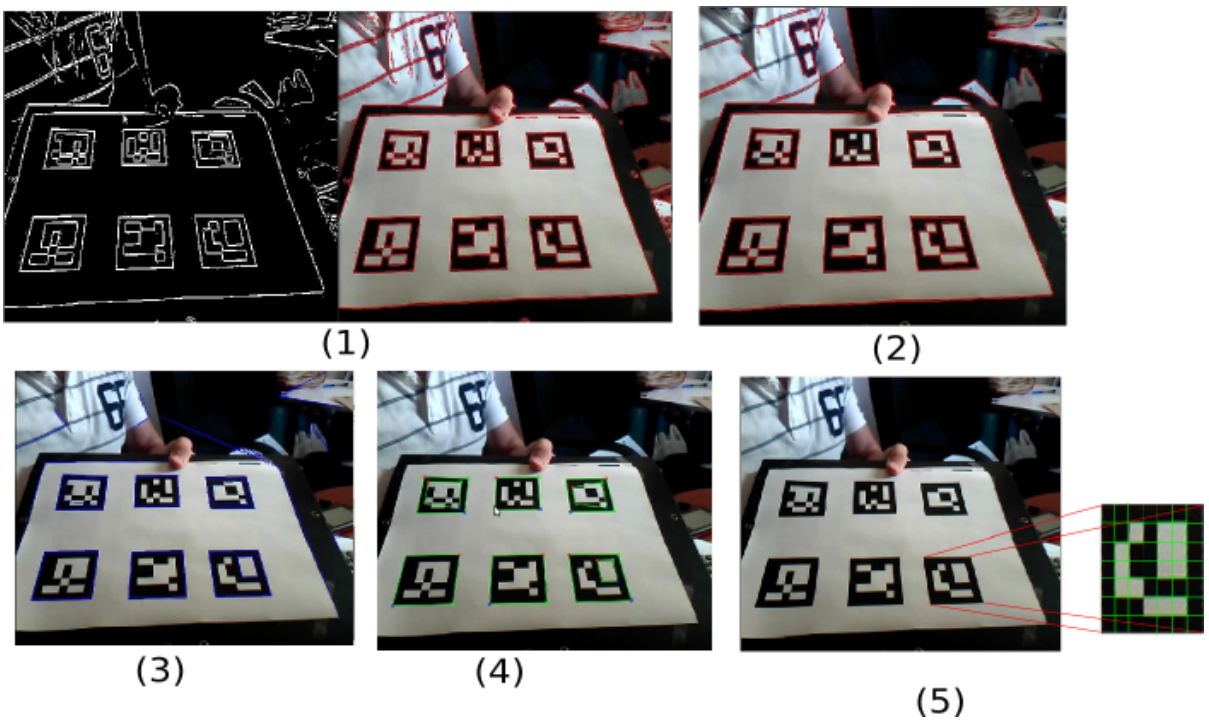
Detection Process in ArUco

The marker detection process of ArUco is as follows:

- Apply an Adaptive Thresholding to obtain borders (Figure 1)
- Find contours. After that, not only the real markers are detected but also a lot of undesired borders. The rest of the process aims to filter out unwanted borders.

- Remove borders with a small number of points (Figure 2)
- Polygonal approximation of contour and keep the concave contours with exactly 4 corners (i.e., rectangles) (Figure 3)
- Sort corners in an anti-clockwise direction.
- Remove too close rectangles. This is required because the adaptive threshold normally detects the internal an external part of the marker's border. At this stage, we keep the most external border. (Figure 4)
- Marker Identification
    - Remove the projection perspective so as to obtain a frontal view of the rectangle area using a homography (Figure 5)
    - Threshold the area using Otsu. Otsu's algorithms assume a bimodal distribution and find the threshold that maximizes the extra-class variance while keeping a low intra-class variance.
    - Identification of the internal code. If it is a marker, then it has an internal code. The marker is divided in a 6x6 grid, of which the internal 5x5 cells contains id information. The rest corresponds to the external black border. Here, we first check that the external black border is present. Afterward, we read the internal 5x5 cells and check if they provide a valid code (it might be required to rotate the code to get the valid one).
    - For the valid markers, refine corners using subpixel interpolation

Finally, if camera parameters are provided, it is computed the extrinsics of the markers to the camera.



Camera Calibration

Concepts

Camera calibration is the process of obtaining the fundamental parameters of a camera. This parameter allows us to determine where a 3D point in the space projects in the camera sensor. The camera parameters can be divided into *intrinsics* and *extrinsics*.

Intrinsic parameters are

- fx,fy: Focal length of the camera lens in both axes. These are normally expressed in pixels
- cx,cy: Optical center of the sensor(expressed in pixels)
- k1,k2,p1,p2,k3: distortion coefficients.

In a ideal camera, a 3D point (X,Y,Z) in the space would project in the pixel:

x= (X fx /Z) +cx ; y= (Y fy/Z)+cy.

However, camera lenses normally distorts the scene making the points far from the center to be even farther. Thus, the vertical stripes near the image borders appear slightly bent. As a consequence, if we want to know a pixel's projection, then we must consider the distortion components. Merely to say that are two type of distortions (radial and tangential) and these are represented by the parameters p1,p2,k1,k2,k3.

Camera Distortion: Straight lines are seen curved because of distortion of the lens

The above assumes that you know the 3D location of the point in relation to the camera reference system. If you want to know the projection of a point referred to an arbitrary reference system, then you must use the extrinsic parameters. The extrinsic parameters are basically the 3D rotations (Rvec={Rx,Ry,Rz})and 3D translations (Tvec={Tx,Ty,Tz}) required to translate the camera reference system to the arbitrary one. The rotation elements are expressed in Rodrigues formula, so you can obtain the equivalent 3x3 rotation matrix using the opencv function cv::Rodrigues().

A very popular alternative to manage both rotation and translation is using Homogeneous coordinates. You can easily create a 4x4 matrix that comprises both rotation and translation. Use the following piece of code to create the 4x4 matrix from rvec and tvec:

```
cv::Mat getRTMatrix ( const cv::Mat &R_,const cv::Mat &T_ ,int forceType ) {
   cv::Mat M;
   cv::Mat R,T;
   R_.copyTo ( R );
   T_.copyTo ( T );
   if ( R.type() ==CV_64F ) {
       assert ( T.type() ==CV_64F );
       cv::Mat Matrix=cv::Mat::eye ( 4,4,CV_64FC1 );
```

```
        cv::Mat R33=cv::Mat ( Matrix,cv::Rect ( 0,0,3,3 ) );
        if ( R.total() ==3 ) {
            cv::Rodrigues ( R,R33 );
        } else if ( R.total() ==9 ) {
            cv::Mat R64;
            R.convertTo ( R64,CV_64F );
            R.copyTo ( R33 );
        }
        for ( int i=0; i<3; i++ )
            Matrix.at<double> ( i,3 ) =T.ptr<double> ( 0 ) [i];
        M=Matrix;
    } else if ( R.depth() ==CV_32F ) {
        cv::Mat Matrix=cv::Mat::eye ( 4,4,CV_32FC1 );
        cv::Mat R33=cv::Mat ( Matrix,cv::Rect ( 0,0,3,3 ) );
        if ( R.total() ==3 ) {
            cv::Rodrigues ( R,R33 );
        } else if ( R.total() ==9 ) {
            cv::Mat R32;
            R.convertTo ( R32,CV_32F );
            R.copyTo ( R33 );
        }
        for ( int i=0; i<3; i++ )
            Matrix.at<float> ( i,3 ) =T.ptr<float> ( 0 ) [i];
        M=Matrix;
    }
    if ( forceType==-1 ) return M;
    else {
        cv::Mat MTyped;
        M.convertTo ( MTyped,forceType );
        return MTyped;
    }
}
```

This matrix allows you to transform a point from one reference system to the other. You can also invert the 4x4 matrix to obtain the opposite transform.

Projection of a 3D point on the camera

Given a calibrated camera, and its extrinsics wrt a global reference system, it is possible to project a 3D point (X,Y,Z)g as follows.

1. **Transform the point (X,Y,Z)g from the global to the camera reference system.** This is the purpose of the extrinsics parameters. Given the 4x4 Extrinsic Matrix M, we obtain the point (X,Y,Z)c in the camera reference system as:

   $$(X,Y,Z,1)c=M*(X,Y,Z,1)g$$

   The Matrix M moves a point from the global coordinate system g to the camera reference system c and can be obtained from the Rvec and Tvec vectors using the code shown above.

2. **Project the point**. Use the camera parameters to project the 3D point (X,Y,Z)c in the camera using the pin-hole camera equations. It will give you the ideal camera coordinates.

   Basically,

   $$x= (Xc*fx /Zc) +cx ;\ \ y= (Yc* fy/Zc)+cy.$$

   Where x and y are the pixels in the image.

3. **Add distortion.** As explained, the real projection will not be in the ideal coordinates because of distortion. Then, we must apply the distortion model to the point (x,y), obtaining (x',y').

This pipeline is used by the cv::projectPoints functions in OpenCV.

For further information on this topic please read the section “Detailed Description” of the OpenCv Documentation on this topic.

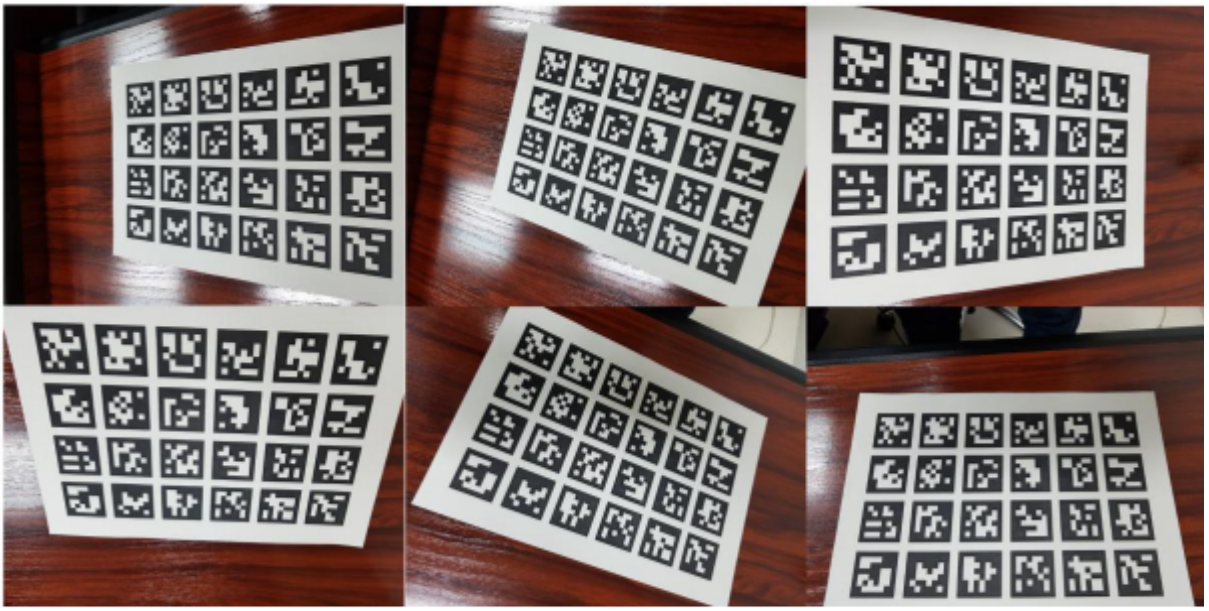
Calibration with Aruco

Aruco has its own calibration board that has an advantage over the classical opencv chessboard: since it is composed by several markers, you do not need to see it completely in order to obtain calibration points. Instead, you can see if partially and still works. This is a very convenient feature in many applications. You have programs in utils_calibration to use calibrate using our chessboard.

Aruco calibration board

To calibrate you should do the following. First, download the calibration board and print it in a piece of paper. Use a tape to measure the size of the markers. Please, be very precise. Annotate the size in meters or in your prefered metric (e.g. inches). Notice that when you compute the distance of your camera to the markers, it will be in the metric you use. So, if you measure the size of the markers in meters, then, you camera position will be expressed in meters too.

Once printed and measured, take photos of the board from different locations and perspectives. Take at least 15 photos, and try to see the whole board in all of them. Here are some examples



Example of images employed for calibration with aruco

Then, use the program in utils_calibration/aruco_calibration_fromimages as

```
aruco_calibration_fromimages mycalibrationfile.yml pathToDirWithImage -size 0.03
```

the parameter mycalibrationfile.yml will be the output of the process.. pathToDirWithImage is the directory containing the images used for calibration. The parameter 0.036 is the size of each marker (you measured earlier), and then the images used for calibration.

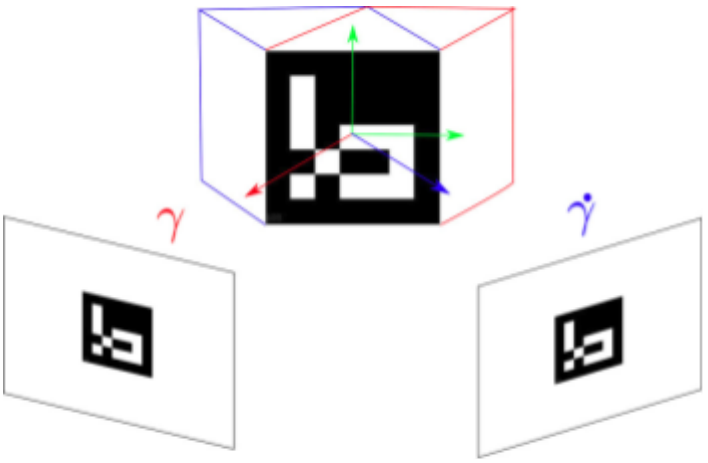
Alternatively, you can calibrate a camera connected to your computer using the program aruco_calibration, which is an interactive calibration program.

Camera pose estimation with ArUco

If the employed camera is calibrated, aruco can be used to estimate its pose (i.e., its 3D position in space with respect to the marker).

We will first explain the case of using independent markers. The following image shows the 3D reference system of each marker. The detection of the four corners of a marker, allows to apply planar pose estimators. They estimate the relative pose of the camera wrt to the center of the marker.

It is extremely important to remark that the estimation of the pose using only 4 coplanar points is subject to ambiguity. As shown in the next figure, a marker could project at the same pixels on two different camera locations. In general, the ambiguity can be solved, if the camera is near to the marker. However, as the marker becomes small, the errors in the corner estimation grows and ambiguity comes as a problem.

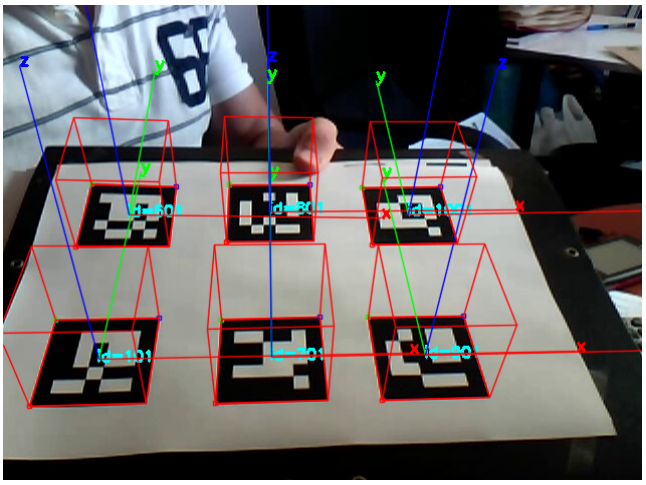


The ambiguity problem. The same marker projection could come from two poses, the two cubes shown in red and blue.

Estimating the pose of markers

We will assume at this point that no ambiguity is present in the captured images.

Using the following code, you'll detect the markers in a image along with the pose of each marker wrt to the camera.



```
#include <opencv2/highgui.hpp>
#include "aruco.h"
using namespace std;
int main(int argc,char **argv)
{
   cv::Mat im=cv::imread(argv[1]);
   aruco::CameraParameters camera;
   camera.readFromXMLFile(argv[2]);
   aruco::MarkerDetector Detector;

Detector.setDictionary("ARUCO_MIP_36h12");
   auto markers=Detector.detect(im,camera,0.05);//0.05 is the marker size
   for(auto m:markers){
        aruco::CvDrawingUtils::draw3dAxis(im,m,camera);
        cout<<m.Rvec<<" "<<m.Tvec<<endl;
   }
   cv::imshow("image",im);
   cv::waitKey(0);
}
```

When you know the camera intrinsics (camera matrix and distortion coefficients obtained by calibration), and you specify the size of the marker, the library can tell you the relative position of the markers and the camera. It is given by the *Rvec* and *Tvec* vectors. They represent the transform from the marker to the camera. You can now use these vectors in the functions cv::projectPoints() to project points in the reference system of the marker. For instance,if you project the point (0,0,0), you'll get the image pixel where the center of the marker is. If you project the point (-0.05/2,0.05/2,0), you'll get the pixel of the first corner (corner 0) Actually, you can check the code of function `aruco::CvDrawingUtils::draw3dAxis` to see how it is done.

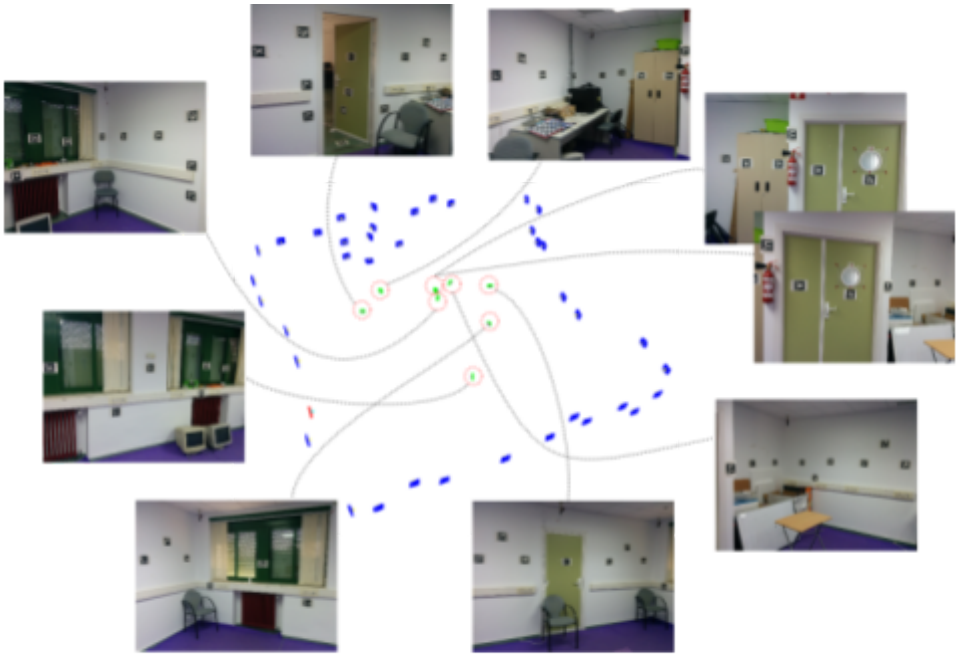
Tracking the position of a marker

Inevitably, the pose of a single marker is subject to the *ambiguity* problem, i.e., two valid solutions can explain the observed projection of the marker. It generally happens when the marker is small in relation to the image, or in very inclined positions. When tracking a marker, you see this effect as a strange change flip of the z-axis of the marker. To alleviate that problem, the library has the possibility of *tracking* the marker position. If you start from a good position, for the next frame, you can fo an optimization looking for the best pose near from the last one. Then, the ambiguity problem can be alleviated. To do so, use the MarkerPoseTracker class. You can see an example in the example program *utils/aruco_tracker.cpp*

Estimate the pose with marker maps

Example of map of markers placed for camera pose localization for AR and Robotics applications

In the previous case, each marker is treated as independent. In marker maps, the position of markers are fixed with respect to each other. They all form a unique reference system wrt which the camera can be localized.



The central image shows in blue the 3D markers placed in the environment. Their location has been automatically extracted from the images show.

Marker maps are of special interest for several reasons. First, imagine an Augmented Reality application in which a virtual game must be drawn. We can use a marker map such as the calibration board. The good thing, is that the occlusion of a single marker will not affect to the the estimation of the camera pose. See for instance the following video. In this case, the relative position of the marker is known since we have created and printed it. Second, imagine now that you want to create a cost-effective camera localization systems based on squared planar markers you can print at home. See for instance the following images. In the left we have randomly set markers on the walls, and in the right image they are in the ceiling. If we know the relative pose of the markers wrt a global reference system, we could know the camera pose in the global reference system by just spotting at one marker.

This is the problem solved in the Marker Mapping project. Below, we show the 3D reconstruction of the marker locations from a set of nine images. As a result, of the process, we obtain a marker map, with the marker locations, that can be used with ArUco for tracking your camera in larger environments. Please notice that the camera employed for creating the marker map does not need to be the same as the one employed for localization.

Assuming that we already have a calibrated camera, and a marker map, we can estimate the pose of the camera wrt to the map reference system as:

```
int main(int argc,char **argv)
{
   cv::Mat im=cv::imread(argv[1]);
   aruco::CameraParameters camera;
   camera.readFromXMLFile(argv[2]);
   aruco::MarkerMap mmap;
   mmap.readFromFile(argv[3]);
   aruco::MarkerMapPoseTracker MMTracker;
   MMTracker.setParams(camera,mmap);

   aruco::MarkerDetector Detector;
   Detector.setDictionary("ARUCO_MIP_36h12");
   auto markers=Detector.detect(im);//0.05 is the marker size

   MMTracker.estimatePose(markers);
   if (MMTracker.isValid())
        cout<<MMTracker.getRvec()<<" "<<MMTracker.getTvec()<<endl,

   cv::imshow("image",im);
   cv::waitKey(0);
}
```

In this case, the *mmap* variable will contain the definition of the map, and *MMTracker* will estimate the pose of the camera. The example can easily be extended to work with a video camera. You can check the library directory *utils/marker_map* were you'll more advances see examples.

Multiple camera Extrinsic Calibration using Marker Maps

Another application of marker maps is finding the extrinsics of multiple cameras using them. A typical problem in many cases is that you have several cameras (static), and you need to find their extrinsics. This is the typical case for 3D reconstruction or tracking. The problem is that estimating the extrinsics using a planar calibration pattern is not very easy, since you have to take multiple images and establish a graph of connection between the cameras.

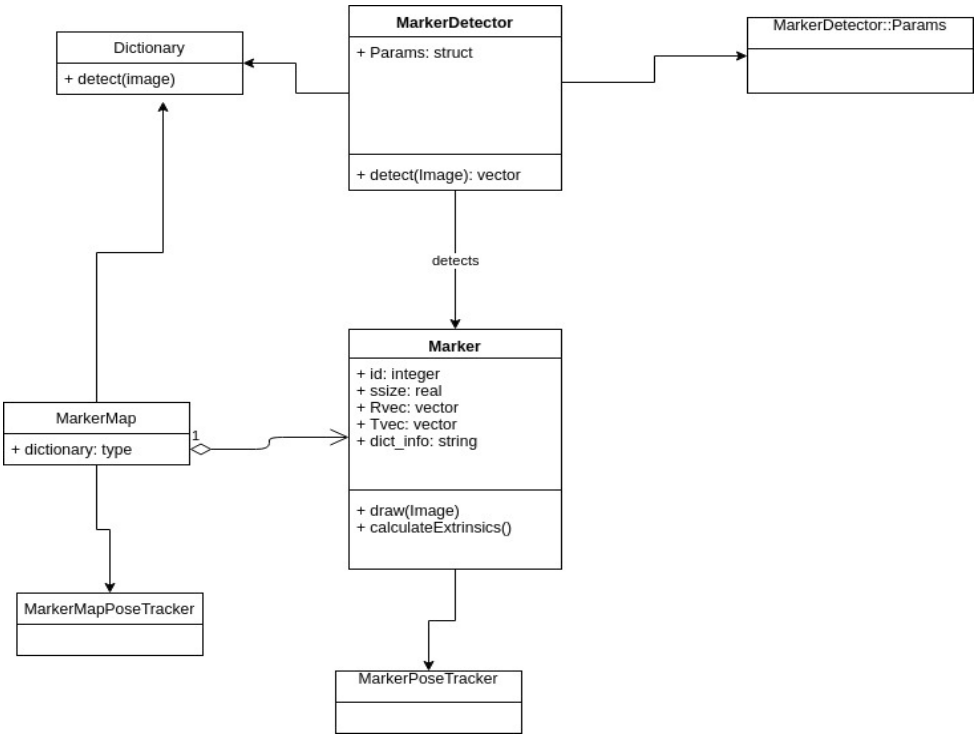
Using marker maps the solution is simpler. You can create a 3D calibration object such as in the next image. We have used a big cardboard box and attached markers to it. Then, using the marker mapping software. We have obtained the marker map with the position of the markers. Now, this 3D calibration object can be set in the middle of the room and estimate the extrinsics of all cameras simultaneously.

3D Calibration object created as a marker map.

Library description

Main classes

The following chart shows the main classes of the library.



- Marker: represents a marker.
- MarkerDetector: does the detection work.
    - MarkerDetector::Params: parameters controlling the different aspect of detection
- Dictionary: defines the set of binary patterns: ARUCO, CHILITGS, ...
- Marker map: A set of markers along with its 3D.
- MarkerPoseTracker: tracks the pose of a marker in a video sequence.
- MarkerMapPoseTracker: tracks the pose of the camera in a video sequence wrt a marker map.

Compiling the library

The ArUco library has been completely implemented in C++11 with the cmake project manager. The library depends on OpenCv, and has been tested with both version 2.4 and 3.X. So the first thing is being sure that OpenCv is in your system.
Then, take the aruco library, uncompress it and compile as:

```
mkdir build; cd  build ; cmake .. -DOpenCV_DIR=<pathTo-OpenCVConfig.cmake>; make -j
```

In Windows systems, you can download the precompiled library if you want, or to compile it yourself. I normally use QtCreator software and Microsoft Visual Studio.

Library Programs

The library comes with several applications that will help you to learn how to use the library. In the folder *utils_xx*, you can find applications focused on different parts of the library. Here we provide a description of them:

`utils`: contains basic programs of the library

- `utils/aruco_print_marker`: which creates marker and saves it in a jpg file you can print.
- `utils/aruco_print_dictionary`: saves to a dictionary all the markers of the dictionary indicated(ARUCO,APRILTAGS,ARTOOLKIT+,etc).
- `utils/aruco_simple` : simple test application that detects the markers in an image
- **`utils/aruco_test`**: this is the main application for detection and profiling. It reads images either from the camera of from a video and detect markers. You have a menu to play with the different parameters of the library. Additionally, if you provide the intrinsics of the camera(obtained by OpenCv calibration) and the size of the marker in meters, the library calculates the marker intrinsics so that you can easily create your AR applications. Also, you can press 'f' to save the configured parameters and reuse later in your applications. See the video for instructions
- `utils/aruco_tracker`: example showing how to use the tracker.

`utils_markermap`: programs that help to learn the use of markermaps

- `utils_markermap/aruco_create_markermap`: creation of simple marker maps (the old boards). It creates a grid of markers that can be printed in a piece of paper. The result of this program are two files (.png and .yml) The png file is an image of the marker you can print. The .yml file is the configuration file that you'll need to pass to the rest of the programs, so they know how the map is. The .yml contains the location of the markers in pixels. Since we do not know in advance how large the printed marker will be, we use pixels here. However, in order to obtain the camera, w need to know the real size of the marker. For this purpose, you can either use the program utils_markermap/aruco_markermap_pix2meters, that creates a new .yml, with the map information in meters. Alternatively, all the test programs allows you to indicate the markersize in the command line.
- `utils_markermap/aruco_markermap_pix2meters:` converts a markermap configuration file from pixels to meters
- `utils_markermap/aruco_simple_markermap` : simple example showing how to determine the camera pose using the marker maps
- `utils_markermap/aruco_test_markermap` : a bit more elaborated example showing how to determine the camera pose using the marker maps with 3D visualization.

`Utils_calibration: programs to calibrate your camera using ArUco`

- `utils_calibration/aruco_calibration` : a program to calibrate a camera using a marker set comprised by aruco markers. It is a marker map,you can download at here
- `utils_calibration/aruco_calibration_fromimages` the same as above, but from images saved in a file
- `utils_gl/aruco_test_gl` simple example showing how to combine aruco with OpenGL

NOTE ON OPENGL: The library supports the integration with OpenGL. In order to compile with support for OpenGL, you just have installed in your system the develop packages for GL and glut (or freeglut).

Creating Marker maps

As already explained, marker maps are useful in some applications since it allow to estimate the pose of the camera very robustly since the occlusion of a marker does not affect the process. Marker maps can be created in two ways. First, using the application utils_markermap/aruco_create_markermap. In that case, all the markers lay in the same plane and the map can be printed in a piece of paper. Second, using the marker mapper. In that case, the markers can be in arbitrary 3D locations, not necessarily on the same plane.

The marker map is expressed as a yml file indicating the markers of the map, along with the 3D location of their corner in the global reference system. When the map is created using the utility *aruco_create_markermap*, the 3D position of the corners is expressed in pixels because it is impossible to know its position in meters (or inches) until you print it. So, once you print it, you must use the application *aruco_markermap_pix2meters* that will create a new yml file where the corner locations are expressed in meters. You can use meters instead of inches without loss of generality.

Main parameters for detection

The detection of markers is controlled by the parameters defined in: *MarkerDetector::Params*. See the following example:

```
MarkerDetector MDetector;
MarkerDetector::Params &params= MDetector.getParameters();
```

The variable params is now a reference to the detector params. You can call its functions to adapt its different values. It is also possible to do the adjustments using the application in utils/aruco_test, and save the configuration to a file use it in your production application.

Next we explain the main elements that control the detection of markers.

Minimum Marker Size and Detection Mode

```
MarkerDetector::Params::setDetectionMode( DetectionMode dm,float minMarkerSize);
```

ArUco version 3 main changes are related to speed and to ease of usage. To do so, we define two main concepts: Minimum Marker Size and Detection Mode.

Minimum Marker Size

In most cases when detecting markers, we are interested in markers of a minimum size. Either we know that markers will have a minimum size in the image, or we are not interested in very small markers because they are not reliable for camera pose estimation. Thus, ArUco 3 defines a method to increase the speed by using images of smaller size for the sake of detection. However, please notice that the precision is not affected by this fact, only computing time is reduced. In order to be general and to adapt to any image size, the minimum marker size is expressed as a normalized value (0,1) indicating the minimum area that a marker must occupy in the image to consider it valid. The value 0 indicates that all markers are considered as valid. As the minimum size increases towards 1, only big markers will be detected.

Which value should I use? Well, you should try in your own video sequences to get the best results. However, we have found that setting this value to 0.02 (2% of the image area) has a tremendous impact in the speed.
If you are processing video and you want maximum speed, you can use the Detection mode DM_VIDEO_FAST (see below) and it will automatically compute the minimum marker by considering the information in the previous frame.

Detection Mode

Refers to three basic use cases that we can normally find among the ArUco users:

1. DM_NORMAL: this is the case of requiring to detect markers in images, and not taking much care about the computing time. This is normally the case of a batch processing in which computing time is not a problem. In this case, we apply a local adaptive threshold approach which is very robust. This is the approach used in ArUco version 2.
2. DM_FAST: in this case, you are concern about speed. Then, a global threshold approach is employed, that randomly searches the best threshold. It works fine in most cases.
3. DM_VIDEO_FAST: this is specially designed for processing video sequences. In this mode, a global threshold is automatically determined at each frame, and the minimum marker size is also automatically determined in order achieve the maximum speed. If the marker is seen very big in one image, in the next frame, only markers of similar size are searched.

By default, the MarkerDetector is configured to be conservative, i.e., with minimum marker size to zero, and in DM_NORMAL mode. If you want to change this behaviour, you need to call

```
void     MarkerDetector::Params::setDetectionMode(     DetectionMode     dm,float
minMarkerSize=0);
```

before detecting markers.

Corner refinement method

```
void MarkerDetector::Params::setCornerRefinementMethod()
```

As previously indicated, the estimation of the marker corners is of special importance when for camera pose estimation. The library defines three refinement methods.

- CORNER_SUBPIX: uses the opencv subpixel function. As previously indicated, in some cases the use of enclosed markers can achieve better results. However, in my experience there is not significant difference.
- CORNER_LINES: this method uses all the pixels of corner border to analytically estimate the 4 lines of the square. Then the corners are estimated by intersecting the lines. This method is more robust to noise. However, it only works if input image is not resized. So, the value minMarkerSize will be set to 0 if you use this corner refinement method.
- CORNER_NONE: Does not corner refinement. In some problems, camera pose does not need to be estimated, so you can use it. Also, in some very noisy environments, this can be a better option.

By default, CORNER_SUBPIX is employed, and works fine in most situations.

Detection of enclosed markers

```
void MarkerDetector::Params::detectEnclosedMarkers(true)
```

As previously indicated, the detection of enclosed markers requires an special treatment. In order to increase the detection rate, activate its detection

Selecting the best parameters for your problem

In general, the ArUco library is very robust and can detect markers in a very wide range of situations. So normally, you do not need to change the default values employed for the parameters except for the Dictionary. However, in some situations, you may want to fine tune the parameters. To do so, use the application utils/aruco_test. It allows you to modify all the library parameters and to save them to a configuration file named *arucoConfig.yml* by pressing 'f'. You can rename the file if you need it later.
The configuration file can be used to configure the marker detector as:

```
        MarkerDetector MDetector;
        MDetector.loadParamsFromFile("arucoConfig.yml");
```

In the following video, you can see how to use the program and fine tune the params.

Using ArUco in your project

Here we show how to create your own project using ArUco and cmake. This example can be downloaded at SourceForge.

First, create a dir for your project (e.g. aruco_testproject). Go in and create the CMakeLists.txt as

```
cmake_minimum_required(VERSION 2.8)
project(aruco_testproject)
find_package(aruco REQUIRED )
add_executable(aruco_simple aruco_simple.cpp)
target_link_libraries(aruco_simple  aruco)
```

Then, create the program file aruco_simple.cpp :

```
#include <iostream>
#include <aruco/aruco.h>
#include <opencv2/highgui.hpp>
int main(int argc,char **argv)
{
  try
  {
      if (argc!=2) throw std::runtime_error("Usage: inimage");
      aruco::MarkerDetector MDetector;
      //read the input image
      cv::Mat InImage=cv::imread(argv[1]);
    //Ok, let's detect
      MDetector.setDictionary("ARUCO_MIP_36h12");
      //detect markers and for each one, draw info and its boundaries in the image
      for(auto m:MDetector.detect(InImage)){
          std::cout<<m<<std::endl;
          m.draw(InImage);
      }
      cv::imshow("in",InImage);
      cv::waitKey(0);//wait for key to be pressed
  } catch (std::exception &ex)
  {
      std::cout<<"Exception :"<<ex.what()<<std::endl;
  }
}
```

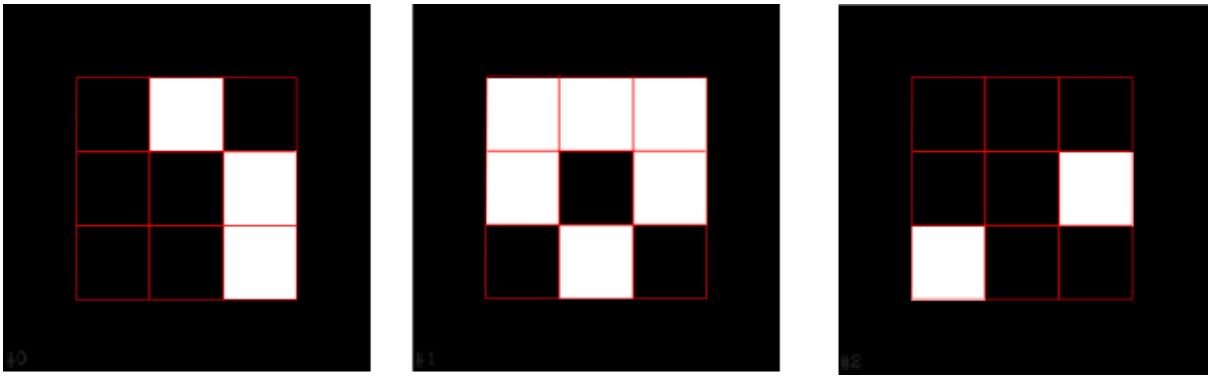
Finally, create a directory for building (e.g. build), go in, compile and execute

```
mkdir build
cd build
cmake .. -Daruco_DIR=<path2arucoConfig.cmake> -DOpenCV_DIR=<path2OpenCVConfig.cmake>
make
./aruco_simple image_withmarkers.jpg
```

Custom Dictionaries

Aruco allows to use your own dictionaries. To do so, first write the definition of your dictionary. Imagine you want to create a dictionary of 3 markers with size 3x3 bits each

shown in the image. (The red lines are just set for the sake of clearly see the bits of the markers.)



You must create then the following file (A copy of this file is in utils/myown.dict)

```
name MYOWN
nbits  9
010001001
111101010
000001100
```

Please notice that the bits are expressed from top left corner to bottom right corner.

Then, you might want to print your dictionary. So, use the application utils/aruco_print_dictionary as:
./utilts/aruco_print_dictionary <pathToSaveAllImages> <pathto/myown.dict>

In the directory indicated as first parameter, the images will be saved. Then, print them and take a picture. To test if it works, use utils/aruco_test. In that case, pass the parameter -d <pathto/myown.dict>. It will automatically load your dict and use it.

Note: this feature is not supported for MarkerMaps.

Note2: If you use aruco_test to generate the configuration file, please consider that it will contain the path to the dictionary. Here is an example of configuration file generated with a custom dir.

```
%YAML:1.0
---
aruco-dictionary: "/home/salinas/Libraries/aruco/trunk/utils/myown.dict"
aruco-detectMode: DM_NORMAL
aruco-cornerRefinementM: CORNER_SUBPIX
aruco-thresMethod: THRES_ADAPTIVE
aruco-maxThreads: 1
aruco-borderDistThres: 1.4999999664723873e-02
aruco-lowResMarkerSize: 20
aruco-minSize: 0.
aruco-minSize_pix: -1
aruco-enclosedMarker: 0
aruco-NAttemptsAutoThresFix: 3
aruco-AdaptiveThresWindowSize: -1
aruco-ThresHold: 7
aruco-AdaptiveThresWindowSize_range: 0
aruco-markerWarpPixSize: 5
```

```
aruco-autoSize: 0
aruco-ts: 2.5000000000000000e-01
aruco-pyrfactor: 2.
aruco-error_correction_rate: 0.
```

Please notice that path. You must be sure when loading the configuration file that the file is in its path.

Frequently Asked Questions

1. **When I draw the axis of a marker, the z axis appears flipped or inverted on some occasions**

   Answer: This is because of the ambiguity problem. It normally happens because the marker is small, or the corners are estimated without precision. Possible ways to deal with ambiguity are:

   - Try using a different corner refinement method.
   - Try using the MarkerPoseTracker.

https://drive.google.com/open?id=14w52PO76d9MIA8tebGpVdCXY8S1qC8zD