Detection of ArUco Markers

Next Tutorial: Detection of ArUco Boards

Pose estimation is of great importance in many computer vision applications: robot navigation, augmented reality, and many more. This process is based on finding correspondences between points in the real environment and their 2d image projection. This is usually a difficult step, and thus it is common to use synthetic or fiducial markers to make it easier.

One of the most popular approaches is the use of binary square fiducial markers. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques.

The aruco module is based on the ArUco library, a popular library for detection of square fiducial markers developed by Rafael Muñoz and Sergio Garrido **[87]**.

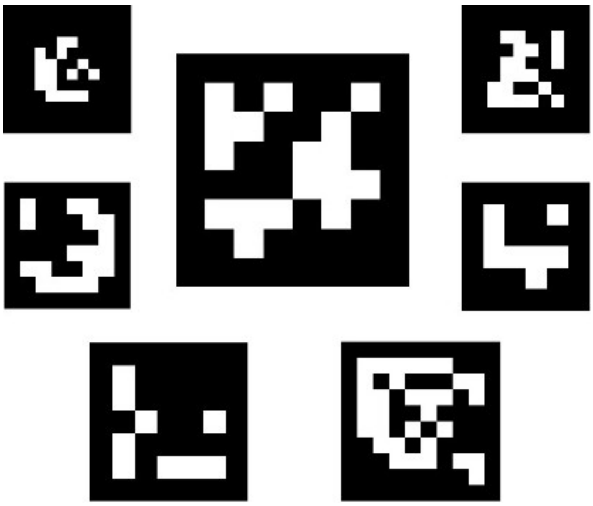
The aruco functionalities are included in:

```
#include <opencv2/aruco.hpp>
```

Markers and Dictionaries

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.

Some examples of ArUco markers:



Example of markers images

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.

More information about dictionaries is provided in the "Selecting a dictionary" section.

Marker Creation

Before their detection, markers need to be printed in order to be placed in the environment. Marker images can be generated using the `generateImageMarker()` function.

For example, lets analyze the following call:

```
cv::Mat markerImage;
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
```

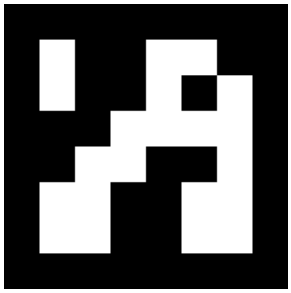
```
cv::aruco::generateImageMarker(dictionary, 23, 200, markerImage, 1);
cv::imwrite("marker23.png", markerImage);
```

First, the `Dictionary` object is created by choosing one of the predefined dictionaries in the aruco module. Concretely, this dictionary is composed of 250 markers and a marker size of 6x6 bits (`cv::aruco::DICT_6X6_250`).

The parameters of `generateImageMarker` are:

- The first parameter is the `Dictionary` object previously created.
- The second parameter is the marker id, in this case the marker 23 of the dictionary `cv::aruco::DICT_6X6_250` . Note that each dictionary is composed of a different number of markers. In this case, the valid ids go from 0 to 249. Any specific id out of the valid range will produce an exception.
- The third parameter, 200, is the size of the output marker image. In this case, the output image will have a size of 200x200 pixels. Note that this parameter should be large enough to store the number of bits for the specific dictionary. So, for instance, you cannot generate an image of 5x5 pixels for a marker size of 6x6 bits (and that is without considering the marker border). Furthermore, to avoid deformations, this parameter should be proportional to the number of bits + border size, or at least much higher than the marker size (like 200 in the example), so that deformations are insignificant.
- The fourth parameter is the output image.
- Finally, the last parameter is an optional parameter to specify the width of the marker black border. The size is specified proportional to the number of bits. For instance a value of 2 means that the border will have a width equivalent to the size of two internal bits. The default value is 1.

The generated image is:



Generated marker

A full working example is included in the `create_marker.cpp` inside the `modules/aruco/samples/` .

Note: The samples now take input from the command line using **cv::CommandLineParser**. For this file the example parameters will look like:

```
"marker23.png" -d=10 -id=23
```

Parameters for `create_marker.cpp` :

```
const char* keys  =
        "{@outfile |<none> | Output image }"
        "{d        |       | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"
        "DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "
        "DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"
        "DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16}"
        "{cd       |       | Input file with custom dictionary }"
        "{id       |       | Marker id in the dictionary }"
        "{ms       | 200   | Marker size in pixels }"
        "{bb       | 1     | Number of bits in marker borders }"
        "{si       | false | show generated image }";
}
```

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.

Consider the following image:

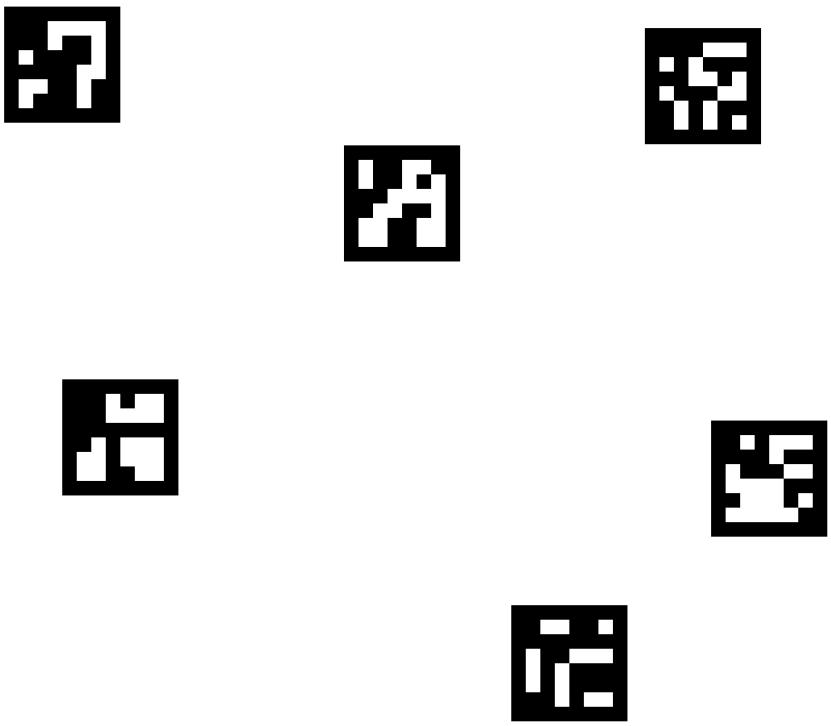
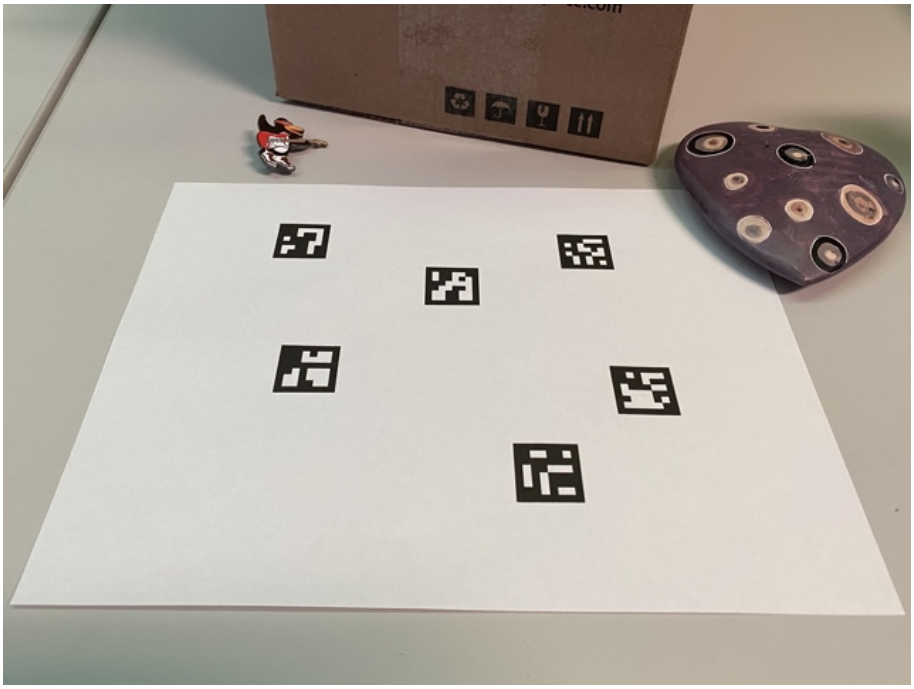


Image with an assortment of markers

And a printout of this image in a photo:



Original image with markers

These are the detected markers (in green). Note that some markers are rotated. The small red square indicates the marker’s top left corner.:

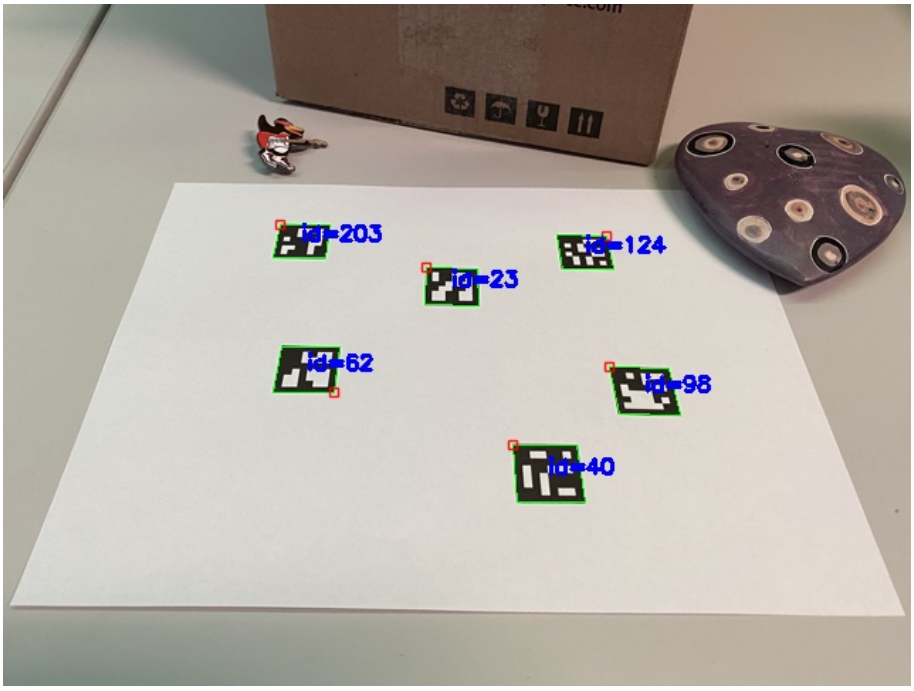


Image with detected markers

And these are the marker candidates that have been rejected during the identification step (in pink):

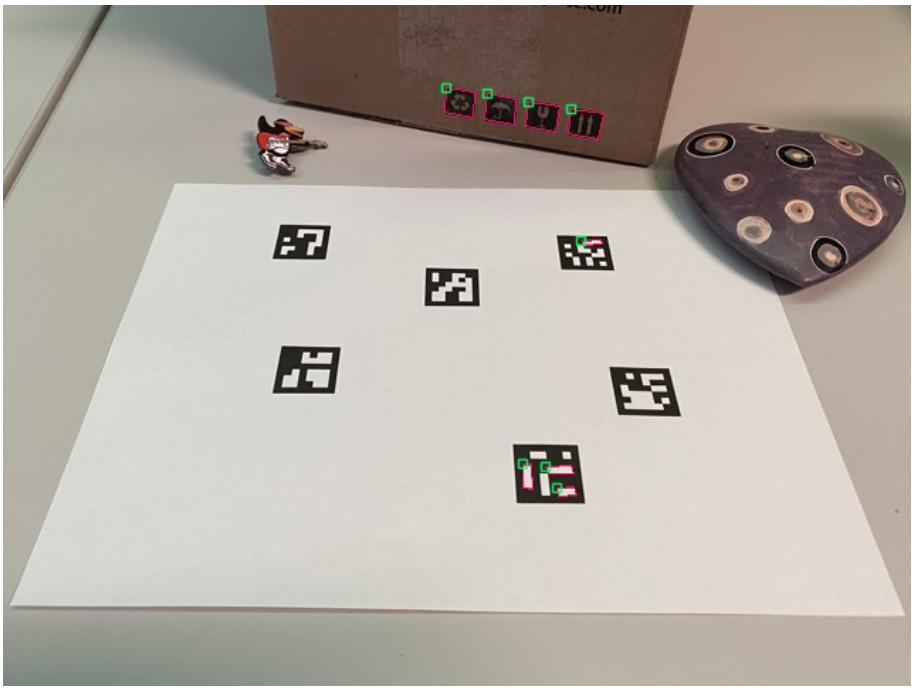


Image with rejected candidates

In the aruco module, the detection is performed in the `detectMarkers()` function. This function is the most important in the module, since all the rest of the functionality is based on the detected markers returned by `detectMarkers()` .

An example of marker detection:

```
cv::Mat inputImage;
...
std::vector<int> markerIds;
std::vector<std::vector<cv::Point2f>> markerCorners, rejectedCandidates;
cv::aruco::DetectorParameters detectorParams = cv::aruco::DetectorParameters();
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
cv::aruco::ArucoDetector detector(dictionary, detectorParams);
detector.detectMarkers(inputImage, markerCorners, markerIds, rejectedCandidates);
```

When you create an `cv::aruco::ArucoDetector` object, you need to pass the following parameters to the constructor:

- A dictionary object, in this case one of the predefined dictionaries (`cv::aruco::DICT_6X6_250`).
- Object of type `cv::aruco::DetectorParameters` . This object includes all parameters that can be customized during the detection process. These parameters will be explained in the next section.

The parameters of `detectMarkers` are:

- The first parameter is the image containing the markers to be detected.
- The detected markers are stored in the `markerCorners` and `markerIds` structures:
  - `markerCorners` is the list of corners of the detected markers. For each marker, its four corners are returned in their original order (which is clockwise starting with top left). So, the first corner is the top left corner, followed by the top right, bottom right and bottom left.
  - `markerIds` is the list of ids of each of the detected markers in `markerCorners` . Note that the returned `markerCorners` and `markerIds` vectors have the same size.
- The final parameter, `rejectedCandidates` , is a returned list of marker candidates, i.e. shapes that were found and considered but did not contain a valid marker. Each candidate is also defined by its four corners, and its format is the same as the `markerCorners` parameter. This parameter can be omitted and is only useful for debugging purposes and for 'refind' strategies (see `refineDetectedMarkers()`).

The next thing you probably want to do after `detectMarkers()` is check that your markers have been correctly detected. Fortunately, the aruco module provides a function to draw the detected markers in the input image, this function is `drawDetectedMarkers()` . For example:

```
cv::Mat outputImage = inputImage.clone();
cv::aruco::drawDetectedMarkers(outputImage, markerCorners, markerIds);
```

- `outputImage` is the input/output image where the markers will be drawn (it will normally be the same as the image where the markers were detected).
- `markerCorners` and `markerIds` are the structures of the detected markers returned by the `detectMarkers()` function.

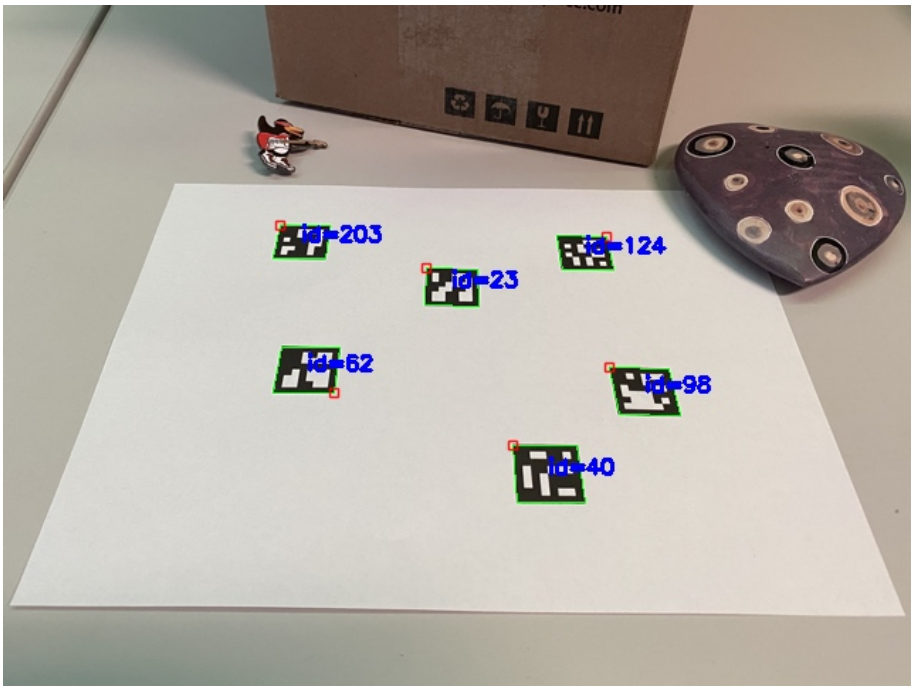


Image with detected markers

Note that this function is only provided for visualization and its use can be omitted.

With these two functions we can create a basic marker detection loop to detect markers from our camera:

```
cv::VideoCapture inputVideo;
inputVideo.open(0);

cv::aruco::DetectorParameters detectorParams = cv::aruco::DetectorParameters();
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
cv::aruco::ArucoDetector detector(dictionary, detectorParams);

while (inputVideo.grab()) {
    cv::Mat image, imageCopy;
    inputVideo.retrieve(image);
    image.copyTo(imageCopy);

    std::vector<int> ids;
    std::vector<std::vector<cv::Point2f>> corners, rejected;
    detector.detectMarkers(image, corners, ids, rejected);

    // if at least one marker detected
    if (ids.size() > 0)
        cv::aruco::drawDetectedMarkers(imageCopy, corners, ids);

    cv::imshow("out", imageCopy);
    char key = (char) cv::waitKey(waitTime);
    if (key == 27)
        break;
}
```

Note that some of the optional parameters have been omitted, like the detection parameter object and the output vector of rejected candidates.

A full working example is included in the `detect_markers.cpp` inside the `modules/aruco/samples/` .

Note: The samples now take input from the command line using **cv::CommandLineParser**. For this file the example parameters will look like

```
-v=/path_to_aruco_tutorials/aruco_detection/images/singlemarkersoriginal.jpg -d=10
```

Parameters for `detect_markers.cpp` :

```
const char* keys  =
        "{d        |       | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"
        "DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "
        "DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"
        "DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16,"
        "DICT_APRILTAG_16h5=17, DICT_APRILTAG_25h9=18, DICT_APRILTAG_36h10=19, DICT_APRILTAG_36h11=20}"
        "{cd       |       | Input file with custom dictionary }"
        "{v        |       | Input from video or image file, if ommited, input comes from camera }"
        "{ci       | 0     | Camera id if input doesnt come from video (-v) }"
        "{c        |       | Camera intrinsic parameters. Needed for camera pose }"
        "{l        | 0.1   | Marker side length (in meters). Needed for correct scale in camera pose }"
        "{dp       |       | File of marker detector parameters }"
        "{r        |       | show rejected candidates too }"
        "{refine   |       | Corner refinement: CORNER_REFINE_NONE=0, CORNER_REFINE_SUBPIX=1,"
        "CORNER_REFINE_CONTOUR=2, CORNER_REFINE_APRILTAG=3}";
}
```

Pose Estimation

The next thing you'll probably want to do after detecting the markers is to use them to get the camera pose.

To perform camera pose estimation, you need to know your camera's calibration parameters. 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()` 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).

As a result of the calibration, you get a 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 relative to the marker is a 3d transformation from the marker coordinate system to the camera coordinate system. It is specified by rotation and translation vectors (see `cv::solvePnP()` function for more information).

```
cv::Mat cameraMatrix, distCoeffs;
// You can read camera parameters from tutorial_camera_params.yml
readCameraParameters(cameraParamsFilename, cameraMatrix, distCoeffs); // This function is implemented in aruco_samples_utility.hpp

std::vector<cv::Vec3d> rvecs, tvecs;

// Set coordinate system
cv::Mat objPoints(4, 1, CV_32FC3);
...

// Calculate pose for each marker
for (int i = 0; i < nMarkers; i++) {
    solvePnP(objPoints, corners.at(i), cameraMatrix, distCoeffs, rvecs.at(i), tvecs.at(i));
}
```

- The `markerCorners` parameter is the vector of marker corners returned by the `detectMarkers()` function.
- The second parameter is the size of the marker side in meters or in any other unit. Note that the translation vectors of the estimated poses will be in the same unit
- `cameraMatrix` and `distCoeffs` are the camera calibration parameters that were created during the camera calibration process.
- The output parameters `rvecs` and `tvecs` are the rotation and translation vectors respectively, for each of the markers in `markerCorners` .

The marker coordinate system that is assumed by this function is placed in the center (by default) or in the top left corner of the marker with the Z axis pointing out, as in the following image. Axis-color correspondences are X: red, Y: green, Z: blue. Note the axis directions of the rotated markers in this image.

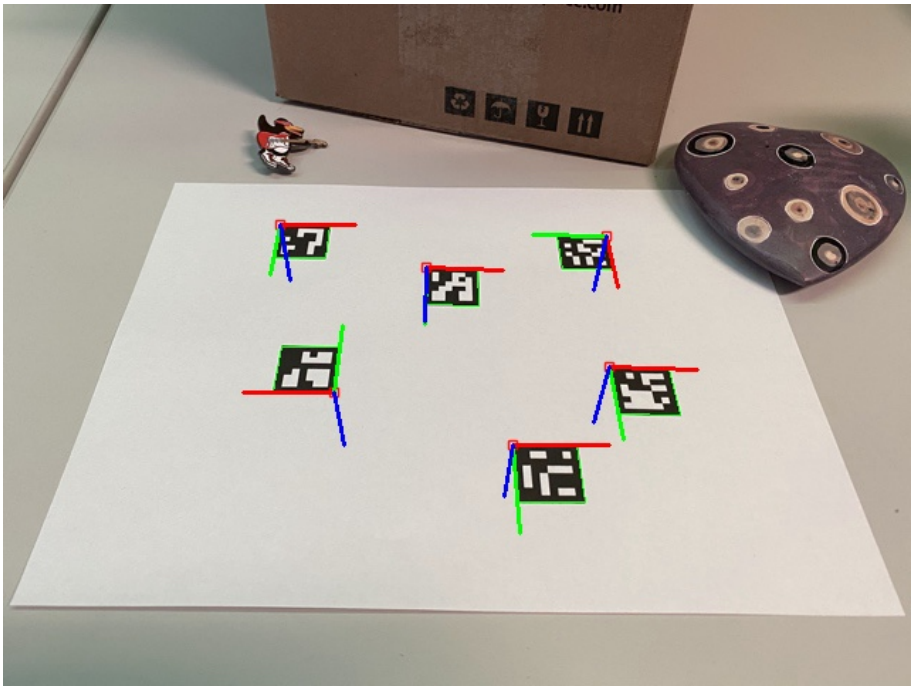


Image with axes drawn

The aruco module provides a function to draw the axis as in the image above, so pose estimation can be checked:

```
inputImage.copyTo(outputImage);
for (int i = 0; i < rvecs.size(); ++i) {
    auto rvec = rvecs[i];
    auto tvec = tvecs[i];
    cv::drawFrameAxes(outputImage, cameraMatrix, distCoeffs, rvec, tvec, 0.1);
}
```

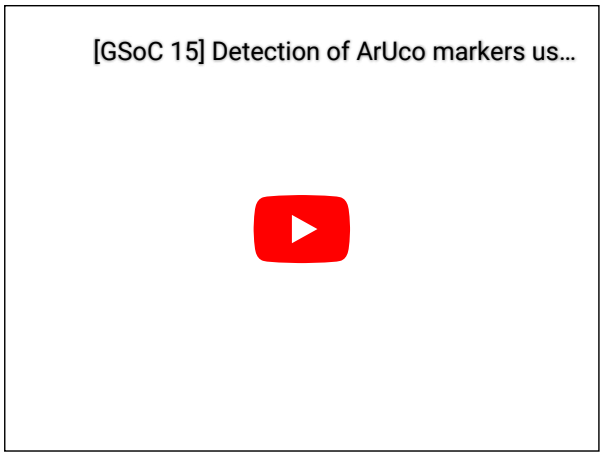
- `outputImage` is the input/output image where the markers will be drawn (it will normally be the same image where the markers were detected).
- `cameraMatrix` and `distCoeffs` are the camera calibration parameters.
- `rvec` and `tvec` are the pose parameters for the marker whose axis is to be drawn.
- The last parameter is the length of the axis, in the same unit as tvec (usually meters).

A basic full example for pose estimation from single markers:

```
cv::VideoCapture inputVideo;
inputVideo.open(0);

cv::Mat cameraMatrix, distCoeffs;
float markerLength = 0.05;

// You can read camera parameters from tutorial_camera_params.yml
readCameraParameters(cameraParamsFilename, cameraMatrix, distCoeffs); // This function is implemented in aruco_samples_utility.hpp

// Set coordinate system
cv::Mat objPoints(4, 1, CV_32FC3);
objPoints.ptr<cv::Vec3f>(0)[0] = cv::Vec3f(-markerLength/2.f, markerLength/2.f, 0);
objPoints.ptr<cv::Vec3f>(0)[1] = cv::Vec3f(markerLength/2.f, markerLength/2.f, 0);
objPoints.ptr<cv::Vec3f>(0)[2] = cv::Vec3f(markerLength/2.f, -markerLength/2.f, 0);
objPoints.ptr<cv::Vec3f>(0)[3] = cv::Vec3f(-markerLength/2.f, -markerLength/2.f, 0);

cv::aruco::DetectorParameters detectorParams = cv::aruco::DetectorParameters();
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
aruco::ArucoDetector detector(dictionary, detectorParams);

while (inputVideo.grab()) {
    cv::Mat image, imageCopy;
    inputVideo.retrieve(image);
    image.copyTo(imageCopy);

    std::vector<int> ids;
    std::vector<std::vector<cv::Point2f>> corners;
    detector.detectMarkers(image, corners, ids);

    // If at least one marker detected
    if (ids.size() > 0) {
        cv::aruco::drawDetectedMarkers(imageCopy, corners, ids);

        int nMarkers = corners.size();
        std::vector<cv::Vec3d> rvecs(nMarkers), tvecs(nMarkers);

        // Calculate pose for each marker
        for (int i = 0; i < nMarkers; i++) {
            solvePnP(objPoints, corners.at(i), cameraMatrix, distCoeffs, rvecs.at(i), tvecs.at(i));
        }
```

```
        // Draw axis for each marker
        for(unsigned int i = 0; i < ids.size(); i++) {
            cv::drawFrameAxes(imageCopy, cameraMatrix, distCoeffs, rvecs[i], tvecs[i], 0.1);
        }
    }

    // Show resulting image and close window
    cv::imshow("out", imageCopy);
    char key = (char) cv::waitKey(waitTime);
    if (key == 27)
        break;
}
```

Sample video:



A full working example is included in the `detect_markers.cpp` inside the `modules/aruco/samples/` .

Note: The samples now take input from the command line using **cv::CommandLineParser**. For this file the example parameters will look like

```
-v=/path_to_aruco_tutorials/aruco_detection/images/singlemarkersoriginal.jpg -d=10
-c=/path_to_aruco_samples/tutorial_camera_params.yml
```

Parameters for `detect_markers.cpp` :

```
const char* keys  =
        "{d        |       | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"
        "DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "
        "DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"
        "DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16,"
        "DICT_APRILTAG_16h5=17, DICT_APRILTAG_25h9=18, DICT_APRILTAG_36h10=19, DICT_APRILTAG_36h11=20}"
        "{cd       |       | Input file with custom dictionary }"
        "{v        |       | Input from video or image file, if ommited, input comes from camera }"
        "{ci       | 0     | Camera id if input doesnt come from video (-v) }"
        "{c        |       | Camera intrinsic parameters. Needed for camera pose }"
        "{l        | 0.1   | Marker side length (in meters). Needed for correct scale in camera pose }"
        "{dp       |       | File of marker detector parameters }"
        "{r        |       | show rejected candidates too }"
        "{refine   |       | Corner refinement: CORNER_REFINE_NONE=0, CORNER_REFINE_SUBPIX=1,"
        "CORNER_REFINE_CONTOUR=2, CORNER_REFINE_APRILTAG=3}";
}
```

Note

To work with examples from the tutorial, you can use camera parameters from `tutorial_camera_params.yml` . An example of use in `detect_markers.cpp` .

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 parameter of the dictionary - the inter-marker distance. The inter-marker distance is the minimum distance between dictionary markers that determines the dictionary's ability to detect and correct errors.

In general, smaller dictionary sizes and larger marker sizes increase the inter-marker distance and vice versa. However, the detection of markers with larger sizes is more difficult 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.

As a consequence, the aruco module includes several ways to select your dictionary of markers, so that you can increase your system robustness:

Predefined dictionaries

This is the easiest way to select a dictionary. The aruco module includes a set of predefined dictionaries in a variety of marker sizes and number of markers. For instance:

```
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
```

`cv::aruco::DICT_6X6_250` is an example of predefined dictionary of markers with 6x6 bits and a total of 250 markers.

From all the provided dictionaries, it is recommended to choose the smallest one that fits your application. For instance, if you need 200 markers of 6x6 bits, it is better to use `cv::aruco::DICT_6X6_250` than `cv::aruco::DICT_6X6_1000`. The smaller the dictionary, the higher the inter-marker distance.

The list of available predefined dictionaries can be found in the documentation for the `PredefinedDictionaryType` enum.

Automatic dictionary generation

A dictionary can be generated automatically to adjust the desired number of markers and bits to optimize the inter-marker distance:

```
cv::aruco::Dictiojary dictionary = cv::aruco::extendDictionary(36, 5);
```

This will generate a customized dictionary composed of 36 markers of 5x5 bits. The process can take several seconds, depending on the parameters (it is slower for larger dictionaries and higher numbers of bits).

Manual dictionary generation

Finally, the dictionary can be configured manually, so that any encoding can be used. To do that, the `Dictionary` object parameters need to be assigned manually. It must be noted that, unless you have a special reason to do this manually, it is preferable to use one of the previous alternatives.

The `Dictionary` parameters are:

```
class Dictionary {
    public:

    cv::Mat bytesList;      // marker code information
    int markerSize;         // number of bits per dimension
    int maxCorrectionBits;  // maximum number of bits that can be corrected

    ...

}
```

`bytesList` is the array that contains all the information about the marker codes. `markerSize` is the size of each marker dimension (for instance, 5 for markers with 5x5 bits). Finally, `maxCorrectionBits` is the maximum number of erroneous bits that can be corrected during the marker detection. If this value is too high, it can lead to a high number of false positives.

Each row in `bytesList` represents one of the dictionary markers. However, the markers are not stored in their binary form, instead they are stored in a special format to simplify their detection.

Fortunately, a marker can be easily transformed to this form using the static method `Dictionary::getByteListFromBits()`.

For example:

```
cv::aruco::Dictionary dictionary;

// Markers of 6x6 bits
dictionary.markerSize = 6;

// Maximum number of bit corrections
dictionary.maxCorrectionBits = 3;

// Let's create a dictionary of 100 markers
for(int i = 0; i < 100; i++)
{
    // Assume generateMarkerBits() generates a new marker in binary format, so that
    // markerBits is a 6x6 matrix of CV_8UC1 type, only containing 0s and 1s
    cv::Mat markerBits = generateMarkerBits();
    cv::Mat markerCompressed = cv::aruco::Dictionary::getByteListFromBits(markerBits);

    // Add the marker as a new row
    dictionary.bytesList.push_back(markerCompressed);
}
```

Detector Parameters

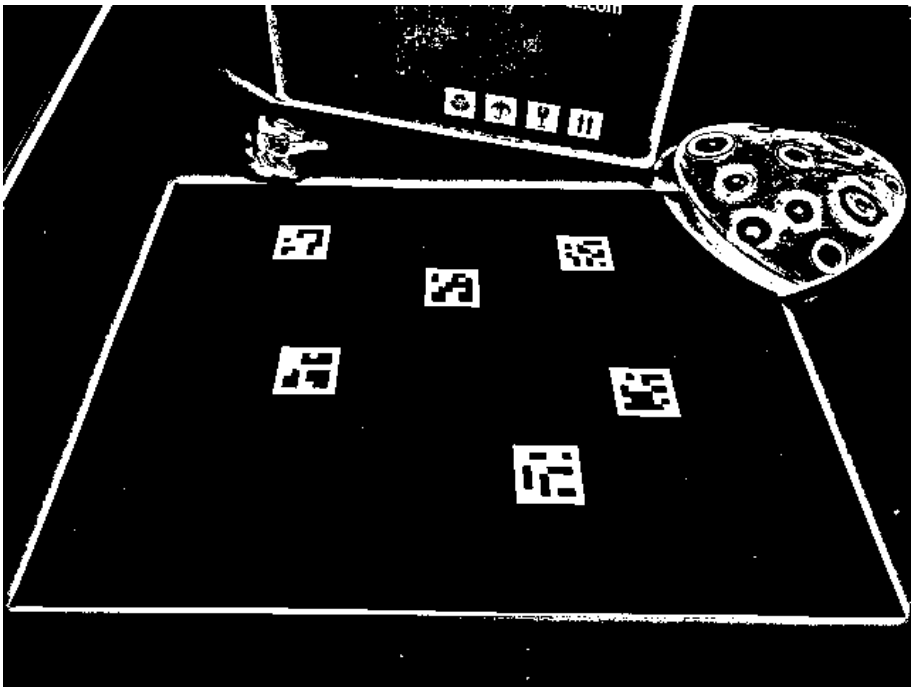
One of the parameters of `ArucoDetector` is a `DetectorParameters` object. This object includes all the options that can be customized during the marker detection process.

This section describes each detector parameter. The parameters can be classified depending on the process in which they're involved:

Thresholding

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

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



Thresholded image

This thresholding can be customized with the following parameters:

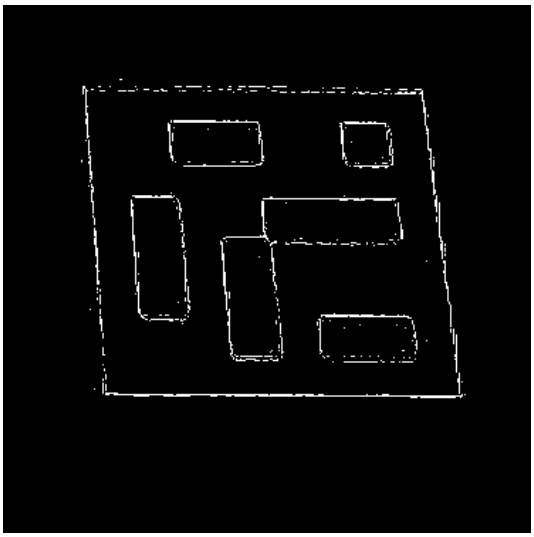
adaptiveThreshWinSizeMin, adaptiveThreshWinSizeMax, and adaptiveThreshWinSizeStep

The `adaptiveThreshWinSizeMin` and `adaptiveThreshWinSizeMax` parameters represent the interval where the thresholding window sizes (in pixels) are selected for the adaptive thresholding (see OpenCV `threshold()` function for more details).

The parameter `adaptiveThreshWinSizeStep` indicates the increments of the window size from `adaptiveThreshWinSizeMin` to `adaptiveThreshWinSizeMax`.

For instance, for the values `adaptiveThreshWinSizeMin` = 5 and `adaptiveThreshWinSizeMax` = 21 and `adaptiveThreshWinSizeStep` = 4, there will be 5 thresholding steps with window sizes 5, 9, 13, 17 and 21. On each thresholding image, marker candidates will be extracted.

Low values of window size can "break" the marker border if the marker size is too large, causing it to not be detected, as in the following image:



Broken marker image

On the other hand, too large values can produce the same effect if the markers are too small, and can also reduce the performance. Moreover the process will tend to global thresholding, resulting in a loss of adaptive benefits.

The simplest case is using the same value for `adaptiveThreshWinSizeMin` and `adaptiveThreshWinSizeMax`, which produces a single thresholding step. However, it is usually better to use a range of values for the window size, although many thresholding steps can also reduce the performance considerably.

Default values:

- `int adaptiveThreshWinSizeMin = 3`
- `int adaptiveThreshWinSizeMax = 23`
- `int adaptiveThreshWinSizeStep = 10`

adaptiveThreshConstant

The `adaptiveThreshConstant` parameter represents the constant value added in the thresholding operation (see OpenCV `threshold()` function for more details). Its default value is a good option in most cases.

Default value:

- `double adaptiveThreshConstant = 7`

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.

minMarkerPerimeterRate and maxMarkerPerimeterRate

These parameters determine the minimum and maximum size of a marker, specifically the minimum and maximum marker perimeter. They are not specified in absolute pixel values, instead they are specified relative to the maximum dimension of the input image.

For instance, a image with size 640x480 and a minimum relative marker perimeter of 0.05 will lead to a minimum marker perimeter of 640x0.05 = 32 pixels, since 640 is the maximum dimension of the image. The same applies for the `maxMarkerPerimeterRate` parameter.

If the `minMarkerPerimeterRate` is too low, detection performance can be significantly reduced, as many more contours will be considered for future stages. This penalization is not so noticeable for the `maxMarkerPerimeterRate` parameter, since there are usually many more small contours than big contours. A `minMarkerPerimeterRate` value of 0 and a `maxMarkerPerimeterRate` value of 4 (or more) will be equivalent to consider all the contours in the image, however this is not recommended for performance reasons.

Default values:

- `double minMarkerPerimeterRate = 0.03`
- `double maxMarkerPerimeterRate = 4.0`

polygonalApproxAccuracyRate

A polygonal approximation is applied to each candidate and only those that approximate to a square shape are accepted. This value determines the maximum error that the polygonal approximation can produce (see `approxPolyDP()` function for more information).

This parameter is relative to the candidate length (in pixels). So if the candidate has a perimeter of 100 pixels and the value of `polygonalApproxAccuracyRate` is 0.04, the maximum error would be 100x0.04=5.4 pixels.

In most cases, the default value works fine, but higher error values could be necessary for highly distorted images.

Default value:

- `double polygonalApproxAccuracyRate = 0.05`

minCornerDistanceRate

Minimum distance between any pair of corners in the same marker. It is expressed relative to the marker perimeter. Minimum distance in pixels is Perimeter * minCornerDistanceRate.

Default value:

- `double minCornerDistanceRate = 0.05`

minMarkerDistanceRate

Minimum distance between any pair of corners from two different markers. It is expressed relative to the minimum marker perimeter of the two markers. If two candidates are too close, the smaller one is ignored.

Default value:

- `double minMarkerDistanceRate = 0.05`

minDistanceToBorder

Minimum distance to any of the marker corners to the image border (in pixels). Markers partially occluded by the image border can be correctly detected if the occlusion is small. However, if one of the corners is occluded, the returned corner is usually placed in a wrong position near the image border.

If the position of marker corners is important, for instance if you want to do pose estimation, it is better to discard any markers whose corners are too close to the image border. Elsewhere, it is not necessary.

Default value:

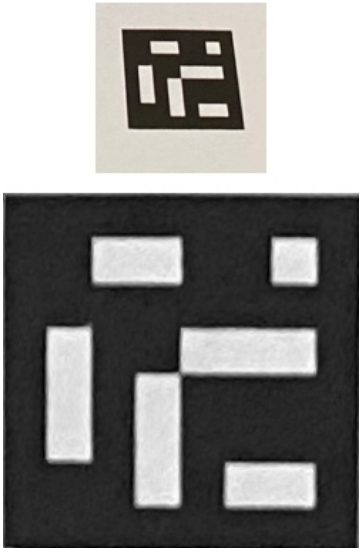
- `int minDistanceToBorder = 3`

Bits Extraction

After candidate detection, the bits of each candidate are analyzed in order to determine if they are markers or not.

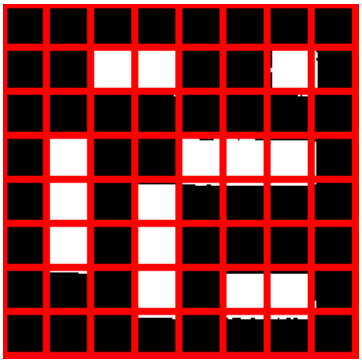
Before analyzing the binary code itself, the bits need to be extracted. To do this, perspective distortion is corrected and the resulting image is thresholded using Otsu threshold to separate black and white pixels.

This is an example of the image obtained after removing the perspective distortion of a marker:



Perspective removing

Then, the image is divided into a grid with the same number of cells as the number of bits in the marker. In each cell, the number of black and white pixels are counted to determine the bit value assigned to the cell (from the majority value):



Marker cells

There are several parameters that can customize this process:

markerBorderBits

This parameter indicates the width of the marker border. It is relative to the size of each bit. So, a value of 2 indicates the border has the width of two internal bits.

This parameter needs to coincide with the border size of the markers you are using. The border size can be configured in the marker drawing functions such as `generateImageMarker()` .

Default value:

- `int markerBorderBits = 1`

minOtsuStdDev

This value determines the minimum standard deviation of the pixel values to perform Otsu thresholding. If the deviation is low, it probably means that all the square is black (or white) and applying Otsu does not make sense. If this is the case, all the bits are set to 0 (or 1) depending on whether the mean value is higher or lower than 128.

Default value:

- `double minOtsuStdDev = 5.0`

perspectiveRemovePixelPerCell

This parameter determines the number of pixels (per cell) in the obtained image after correcting perspective distortion (including the border). This is the size of the red squares in the image above.

For instance, let's assume we are dealing with markers of 5x5 bits and border size of 1 bit (see `markerBorderBits`). Then, the total number of cells/bits per dimension is 5 + 2*1 = 7 (the border has to be counted twice). The total number of cells is 7x7.

If the value of `perspectiveRemovePixelPerCell` is 10, then the size of the obtained image will be 10*7 = 70 -> 70x70 pixels.

A higher value of this parameter can improve the bits extraction process (up to some degree), however it can penalize the performance.

Default value:

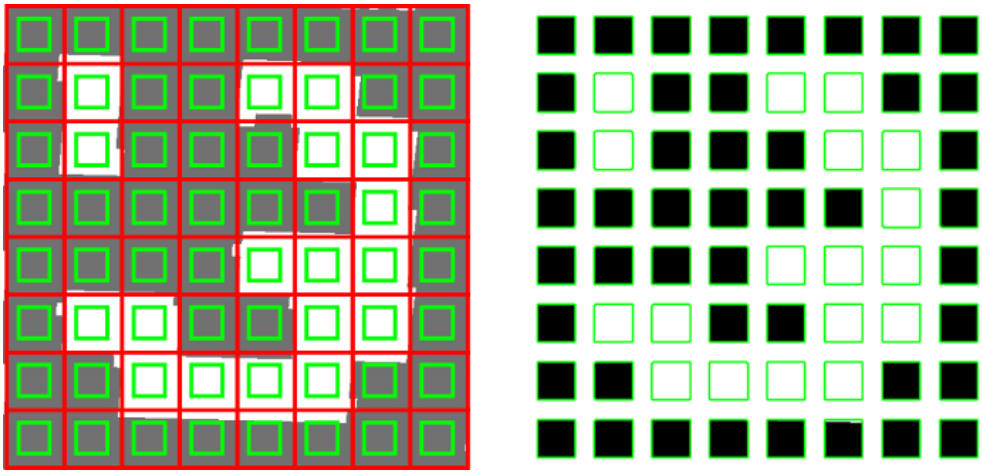
- `int perspectiveRemovePixelPerCell = 4`

perspectiveRemoveIgnoredMarginPerCell

When extracting the bits of each cell, the numbers of black and white pixels are counted. In general, it is not recommended to consider all the cell pixels. Instead it is better to ignore some pixels in the margins of the cells.

The reason for this is that, after removing the perspective distortion, the cells' colors are, in general, not perfectly separated and white cells can invade some pixels of black cells (and vice versa). Thus, it is better to ignore some pixels just to avoid counting erroneous pixels.

For instance, in the following image:



Marker cell margins

only the pixels inside the green squares are considered. It can be seen in the right image that the resulting pixels contain a lower amount of noise from neighbor cells. The `perspectiveRemoveIgnoredMarginPerCell` parameter indicates the difference between the red and the green squares.

This parameter is relative to the total size of the cell. For instance if the cell size is 40 pixels and the value of this parameter is 0.1, a margin of 40*0.1=4 pixels is ignored in the cells. This means that the total number of pixels that would be analyzed in each cell would actually be 32x32, instead of 40x40.

Default value:

- `double perspectiveRemoveIgnoredMarginPerCell = 0.13`

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.

maxErroneousBitsInBorderRate

The bits of the marker border should be black. This parameter specifies the allowed number of erroneous bits in the border, i.e. the maximum number of white bits in the border. It is represented relative to the total number of bits in the marker.

Default value:

- `double maxErroneousBitsInBorderRate = 0.35`

errorCorrectionRate

Each marker dictionary has a theoretical maximum number of bits that can be corrected (`Dictionary.maxCorrectionBits`). However, this value can be modified by the `errorCorrectionRate` parameter.

For instance, if the allowed number of bits that can be corrected (for the used dictionary) is 6 and the value of `errorCorrectionRate` is 0.5, the real maximum number of bits that can be corrected is 6*0.5=3 bits.

This value is useful to reduce the error correction capabilities in order to avoid false positives.

Default value:

- `double errorCorrectionRate = 0.6`

Corner Refinement

After markers have been detected and identified, the last step is performing subpixel refinement of the corner positions (see OpenCV `cornerSubPix()` and `cv::aruco::CornerRefineMethod`).

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.

cornerRefinementMethod

This parameter determines whether the corner subpixel process is performed or not and which method to use if it is being performed. It can be disabled if accurate corners are not necessary. Possible values are `CORNER_REFINE_NONE` , `CORNER_REFINE_SUBPIX` , `CORNER_REFINE_CONTOUR` , and `CORNER_REFINE_APRILTAG` .

Default value:

- `int cornerRefinementMethod = CORNER_REFINE_NONE`

cornerRefinementWinSize

This parameter determines the window size of the subpixel refinement process.

High values can cause close corners of the image to be included in the window area, so that the corner of the marker moves to a different and incorrect location during the process. Also, it may affect performance.

Default value:

- `int cornerRefinementWinSize = 5`

cornerRefinementMaxIterations and cornerRefinementMinAccuracy

These two parameters determine the stop criteria of the subpixel refinement process. The `cornerRefinementMaxIterations` indicates the maximum number of iterations and `cornerRefinementMinAccuracy` the minimum error value before stopping the process.

If the number of iterations is too high, it may affect the performance. On the other hand, if it is too low, it can result in poor subpixel refinement.

Default values:

- `int cornerRefinementMaxIterations = 30`
- `double cornerRefinementMinAccuracy = 0.1`