1 Object Identification using OpenCV

In mp2, you are required to take pictures using the Raspberry Pi camera and identify known objects in those pictures. You can find pictures of the known objects in http://goo.gl/2rwbzP. In this section, we describe and provide you with code that implements the object identification algorithm. The code along with the test images are available here http://courses.engr.illinois.edu/cs424/mp/object-identification.tar.gz.

Note the following:

- The camera should be mounted on the left side of the cargo bay of the robot. As the robot follows the wall, a camera mounted on the left side will be able to capture objects placed inside the maze. Figure 1 shows such a picture taken from the mounted camera.

- You may need to adjust your speed when taking a picture, otherwise the photo might have motion blur, and won’t be good for object detection. Also make sure the camera doesn’t wiggle when the robot is moving.

- Object identification is relatively slow. It may take 2 seconds to process one frame. You are supposed to run it as a thread inside your main process for the robot. The detection should run in the Raspberry Pi, while the robot is traversing the maze.

- RaspiCam Cv::retrieve function gives you an OpenCV object of class Mat. The object identification thread should use the Mat object obtained from the camera. You are not supposed to store it as a file and read it back.

1.1 Steps of Object Identification

An object identification algorithm requires the following steps. Here we go through the example code robovision.cc that you can adapt for your program.
Keypoint Detection and Feature Description  You need to detect interesting keypoints from the scene as the features of that picture. You should also compute the feature vectors corresponding to those keypoints. This step is required so that you would be able to compare and match the feature vectors between two pictures. The following code creates a detector that uses SURF algorithm to compute the keypoints and the vectors for a query image and a test image.

```cpp
// Detect the keypoints and extract descriptors using SURF
int minHessian = 100;
int nOctaves = 4;
int nOctaveLayers = 3;
Ptr<SURF> detector = SURF::create(
    minHessian, nOctaves, nOctaveLayers, true);

// Keypoint detector and descriptor.
vector<KeyPoint> keypoints_query, keypoints_scene;
Mat descriptors_query, descriptors_scene;
detector->detectAndCompute(
    img_query, Mat(), keypoints_query, descriptors_query);
detector->detectAndCompute(
    img_scene, Mat(), keypoints_scene, descriptors_scene);
```

In this snippet we are running the feature extraction for both the query image (an object we are looking for), and the scene image (photo received from camera). *Note that you do not need to perform the feature extraction for the query image every time there is a scene image. Doing so will slow you down. You can precompute the features of the query images beforehand when you start.*
There are many mechanisms for feature detection and description, including SIFT, SURF, ORB, BRISK, etc. Many of these algorithms are implemented in OpenCV. A practical challenge for these algorithms is to be invariant to lighting, scaling, rotation, distortion, blur, noise, etc so that the variations in the picture of the same object would not affect detection. SIFT generally performs best against many variations, however it is also the slowest. SURF comes next, and is much faster than SIFT. We are using SURF in our example code for detection. You are, however, welcome to try with the faster but less robust algorithms that are also available in OpenCV. You can start here [http://docs.opencv.org/3.1.0/d9/d97/tutorial_table_of_content_features2d.html](http://docs.opencv.org/3.1.0/d9/d97/tutorial_table_of_content_features2d.html)

**Keypoint Matching**  In this stage we match the keypoint feature vectors extracted from the query image with those of the scene image. In the following snippet, we take the feature vectors from the query image, and use a Brute Force Matcher to find 2-Nearest Neighbors in the scene image. The matches with good similarity are queued for further verification and object localization.

```cpp
// Matching descriptor vectors using Brute Force matcher
BFMatcher matcher(NORM_L2);
vector<vector<DMatch>> matches;
matcher.knnMatch(descriptors_query, descriptors_scene, matches, 2);

vector<DMatch> good_matches;
for(int i = 0; i < descriptors_query.rows; i++) {
    if (matches[i][0].distance < 0.75 * matches[i][1].distance)
        good_matches.push_back(matches[i][0]);
}
```

Note that you can replace BFMatcher with FLANN matcher (Fast Library for Approximate Nearest Neighbors) for slightly higher speed with possible lower quality. Details are available here [http://docs.opencv.org/3.1.0/d7/dff/tutorial_feature_homography.html](http://docs.opencv.org/3.1.0/d7/dff/tutorial_feature_homography.html)

**Outlier Elimination and Object Localization**  After the matching stage, a lot of outliers are removed. However, there are still many outliers that can result in false matching. To remove outliers we use the fact that if there is a match, there will be a subset of keypoints in the test image that will match with keypoints from the query image, and it will be possible to find a perspective projection that localizes the query object in the scene. To find that we use `findHomography` function with RANSAC algorithm.

```cpp
Mat H = findHomography(query, scene, RANSAC);
perspectiveTransform(query_corners, scene_corners, H);
```

The code is inside the `alignPerspective` function of `robovision.cc`. False positives can still occur, and we use some heuristics to verify the geometric construction of the projection, and eliminate many false positives. For example,
the detected projection should form a convex hull, it should not have too small angles, it should not have too small or too large area, it should not have a large portion outside the scene, etc.

### 1.2 Running Object Identification

Download and untar the archive of the object identification folder by running

```sh
gwget http://courses.engr.illinois.edu/cs424/mp/object-identification.tar.gz
tar -zxvf object-identification.tar.gz
```

Inside you will get the code in `robovision.cc` and a `Makefile`. One test image is stored in the folder `scene-image`. The query images are stored in the folder `query-image`. Note that there are two versions of the query images. There are the full sized versions named like `ancient-lamp.jpg`, `mammoth.jpg` which you should use when printing those in a paper to test your robot. On the other hand, use the lower resolution version from `query-image/low-resolution`, named like `ancient-lamp-600.jpg` or `mammoth-600.jpg` with the identification program. The reason is that higher resolution images slows down matching.

*Note that the object identification program crops a portion of the test image from the bottom.* This is because the images taken by mounting the camera on the robot will have some portion of the side of the robot at the bottom. Note in Figure 1, the lower 15% of the image is all white which is basically side of the robot. To reduce ambiguity during detection and to speed up feature extraction, we crop it. We find the useful portion of the picture is the top 85% that contains useful information when the camera is mounted on the left side near the front corner. The actual amount of cropping you should do may be different depending on exactly where you mount the camera.

The `robovision.cc` code expects the filename for query image, test image, output image, and the crop fraction as command line arguments. It performs the feature extraction, matching, homography, and false positive elimination steps, and prints the time required for the important steps on the console. It produces a message stating the outcome of the matching. Regardless of that outcome, it prepares an output image that shows the query image and the test image side-by-side, and plots the matching keypoints. In case of a match it localizes the query image by drawing the perspective projection on the test image. You can use the output image to debug false positives or false negatives that may arise.

Here we present some examples of running the object identification program. Run `make` to compile. The following command in blue detects if object 'Roman glass' is in the test image of Figure 1. Output image is Figure 2a.

```sh
./robovision query-image/low-resolution/roman-glass-600.jpg \ scene-image/irobot_scene_1.jpg matched_image_roman-glass.jpg 0.85
```

Feature extraction query image 0.431939 sec
Feature extraction scene image 1.354 sec
Matching and alignment 0.831468 sec
Object found
Detect if object ‘Ancient lamp’ is in Figure 1. Output image is Figure 2b.

```
./robovision query-image/low-resolution/ancient-lamp-600.jpg \ scene-image/irobot_scene_1.jpg matched_image_ancient-lamp.jpg 0.85
Feature extraction query image 0.640285 sec
Feature extraction scene image 1.39866 sec
Matching and alignment 0.951198 sec
Object found
```

Detect if object ‘Mammoth’ is in Figure 1. Output image is Figure 2c.

```
./robovision query-image/low-resolution/mammoth-600.jpg \ scene-image/irobot_scene_1.jpg matched_image_mammoth.jpg 0.85
Feature extraction query image 0.477079 sec
Feature extraction scene image 1.42309 sec
Failed rule1: Not a convex hull
Matching and alignment 0.591072 sec
Object not found
```

Detect if object ‘Mayan calendar’ is in Figure 1. Output image is Figure 2d.

```
./robovision query-image/low-resolution/mayan-calendar-600.jpg \ scene-image/irobot_scene_1.jpg matched_image_mayan-calendar.jpg 0.85
Feature extraction query image 1.72273 sec
Feature extraction scene image 1.39788 sec
Failed rule4: Contains very small angle
Matching and alignment 3.28273 sec
Object not found
```
2 Finding Contour

As part of mp2, you are required to plot the contour of the maze the robot is in. There are two steps of this. At first you need to record the important waypoints while the robot is travelling. And then, you need to connect the successive waypoints, plot using OpenCV, and produce an output image.

2.1 Calculating Waypoints

As we do not have any GPS or navigation device on the robot, we cannot use global coordinates to determine the position. Therefore, we opt for a local coordinate system. After starting, when the robot reaches the first wall, and aligns itself parallel to the wall, consider that point as \((0,0)\). We need to track subsequent moves to calculate the coordinates. Note that it is not necessary to track your position every interval. Tracking this way will increase the complexity and make the contours look noisy. Instead, you can track ‘waypoints’, or important turning points during the travel. For example, once you detect the wall at the right side has ended, you have to record a new waypoint. To do that, you should calculate the distance you have travelled from the last waypoint, and create a new waypoint from the last waypoint. You can use the formula, \(\text{distance} = \text{speed} \times \text{time}\) to find your distance from the last waypoint. To use this formula, you should keep track of your speed also. Now, once the wall ended and you need to turn clockwise, you should adjust your direction vector in the program appropriately. Similarly, if you have hit or come very near to a wall perpendicular to the current wall, you should calculate a new waypoint, and adjust your distance vector in the program.

Here we describe a simple mechanism for tracking the position of the robot. There are two important variables you should maintain, and one important formula your program requires. The variables are (1) Last recorded waypoint, and (2) Current direction vector. The formula is the transformation matrix for vector rotation. Figure 3a and 3b illustrates how these variables are used.

- Last recorded waypoint should be maintained as \((x, y)\) coordinate. You can initialize it as \((0, 0)\). Once you need to generate a new waypoint \(B\), and your last waypoint was \(A\), you can use the formula in Equation 1.

\[
B = A + \text{dist} \times U
\]

Here \(U\) is a unit vector representing direction, and \(\text{dist}\) is the distance travelled since last waypoint \(A\). In OpenCV, you can represent both the points and the direction vector as \texttt{Point2f}. OpenCV has C++ operators overloaded, and you can directly write \texttt{pos += dist * dir}, where \texttt{pos} represents the waypoint, and \texttt{dir} represents the current direction vector. Note that \texttt{point2f} contains the \(x\) and \(y\) coordinates in float, not in double.

- Current direction vector maintained as \((i, j)\) unit vector. You can initialize it to be \((0, 1)\), which means the robot is facing positive \(y\)-axis. Once the
Figure 3: Calculating waypoints using vector algebra

robot rotates, you need to update this direction vector. For example, you should update it when the wall at right side ends and the robot needs to turn right, i.e. clockwise. Note that counter clockwise rotations are expressed as positive quantities, and clockwise rotations are negative. You should also update the direction vector when the robot needs to turn left because it has hit a wall when moving parallel to another wall. Note that the direction vector must start normalized, and should always stay normalized. The update formula for the direction vector is $V = R \times U$, where $R$ is a $2 \times 2$ rotation matrix for rotation of angle $\theta$, given by Equation 2.

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$  \hspace{1cm} (2)

If you do not want to use this matrix, you can use the equivalent linear equations instead, Given $U$ and $V$ are OpenCV Point2f objects, you can access $U_x$ by $U.x$, and $U_y$ by $U.y$. Note that $\theta$ is always in radian.

$$V_x = U_x \cdot \cos \theta - U_y \cdot \sin \theta$$ \hspace{1cm} (3)

$$V_y = U_x \cdot \sin \theta + U_y \cdot \cos \theta$$ \hspace{1cm} (4)

2.2 Drawing Contour using OpenCV

2.2.1 List of waypoints

Because you need to draw the contour of the maze, you should record the waypoints you generate, preferably in a std::vector. The following statements create a vector named waypoints, insert two Point2f points in it, and prints those points.
std::vector<Point2f> waypoints; // Create a list of points.

// Insert some points.
Point2f apoint(10,10); // Create point (10, 10).
waypoints.push_back(apoint);
waypoints.push_back(Point2f(100, 200)); // Insert point (100, 200) directly.

// Print those points.
for (auto point : waypoints)
    std::cout << point << std::endl;

2.2.2 OpenCV drawing and geometry functions

You can use Point2f for points, which stores \(x\) and \(y\) coordinates as 32-bit floats. To plot the points you need a drawing context. The fundamental drawing context in OpenCV is Mat. To plot your waypoints and draw the contours, you need to create a Mat object. You can plot lines, circles, using the functions line or circle. Color objects are generated using Scalar. Scalar(255, 0, 0) specifies blue color, and Scalar(0, 0, 255) is red. Note that OpenCV uses BGR format internally instead of RGB.

The following code creates a Mat object of size 1600 \(\times\) 1200, and uses white background color. It then reads the waypoint vector that was previously created, and plots the points with a small circle, and connects those using lines. It then calculates the bounding rectangle of the plotted points and draws that rectangle. Finally, the work is stored as a png file named irobot_plot.png.

// Create a drawing context, and use white background.
Mat img_output(1200, 1600, CV_8UC3, Scalar(255, 255, 255));

// Plot the waypoints using blue color.
Scalar lineColor(255, 0, 0);
int lineWidth = 1;
int radius = 3;
for (int i = 0; i < waypoints.size() - 1; i++) {
    line(img_output, waypoints[i], waypoints[i + 1],
         lineColor, lineWidth, CV_AA);
    circle(img_output, waypoints[i], radius,
           lineColor, CV_FILLED, CV_AA);
}

// Draw the bounding rectangle using orange color
Rect bound = boundingRect(waypoints);
rectangle(img_output, bound, Scalar(0, 165, 255));

// Finally store it as a png file
imwrite("irobot_plot.png", img_output);
Note that if you use `arrowedLine` instead of `line`, it will add an arrow at the end, which may be helpful during debugging. Note that the presented code is just a toy example. In your code, you should use the bounding rectangle to determine the bounds of the waypoints beforehand, and translate and scale them properly so that your final picture doesn’t get arbitrarily large. Following is a list of helpful links.

- You can know more about `Mat` from [http://docs.opencv.org/3.1.0/d6/d6d/tutorial_mat_the_basic_image_container.html](http://docs.opencv.org/3.1.0/d6/d6d/tutorial_mat_the_basic_image_container.html)
- Detailed documentation of basic data structures like `Point2f` or `Matx` [http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html](http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html)
- Examples of basic drawing using OpenCV [http://docs.opencv.org/3.1.0/d3/d96/tutorial_basic_geometric_drawing.html](http://docs.opencv.org/3.1.0/d3/d96/tutorial_basic_geometric_drawing.html)
- Detailed documentation of drawing functions [http://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html](http://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html)
- Documentation for various computational geometry algorithms like `convexHull` or `boundingRect` [http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html](http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html)

## 3 Helpful Features of C++11

This section is a potpourri of various helpful features of C++11.

- To sleep current thread for `ival` milliseconds, you need to `#include<chrono>` and use `this_thread::sleep_for(chrono::milliseconds(ival))`
- You can use the following to specify deadline using current time and an `interval`, and sleep until that deadline. In this way you can generate `start_time` and the deadline at the beginning of some execution. After the execution call `sleep_until` to sleep for only rest of the interval.
  ```
  auto start_time = std::chrono::system_clock::now();
  auto deadline = start_time + std::chrono::milliseconds(interval);
  // Execute more code here ......
  this_thread::sleep_until(deadline);
  ```