Cupinator – Project summary

Members:
Lior Zur-Lotan, 300197654, llutan@cs.bgu.ac.il
Maor Ashkenazi, 312498017, maorash@cs.bgu.ac.il
Micahel Bar-Sinai, 033017229, basinam@cs.bgu.ac.il

Cupinator Description

Our objective was to enable a basic ability of object recognition in an unknown environment, but for known objects (targets). We wanted to let a robot find an object in order to enable further handling (e.g. retrieving the object or avoiding it). To be able to demonstrate this and also to produce a concrete ability, we implemented the project around the task of recognizing a red cup (the object recognition) and approaching it (object handling).

The main issues tackled in the project:

1. Object recognition:
    a. Noisy environment: Since our goal was to allow the robot to identify its targets in unknown environments, we had to handle a lot of potential noise in the environment, both in the form of false positive recognitions (i.e. Identifying the wrong object as the target) and false negatives (e.g. missing the object due to poor lighting)
    b. Limited sensors: Recognizing an object first required separating the object from its environment. Doing this requires some minimal accuracy of the sensors. Specifically, minimal picture capture resolution, accurate color identification and accurate depth perception. With respect to these sensors, a robot can be a very limited platform, so these limitations must be compensated for in software
    c. Limited resources: Operating in an unknown environment means there is very little that can be assumed or preprocessed. This leads to a lot of computations that must be done in real time and without being addressed, these computations can prevent the recognition process from even completing in a reasonable time.
2. Motion in the environment (an instance of object handling):
    a. Obstacles: Since there is no preliminary information about the environment, a general procedure for obstacle avoidance must be implemented without interfering the object recognition procedure
    b. Inaccurate motion control: Since the robot operates in the real world, it's unrealistic to expect it to move as accurately as required. This requires a form of correction for any possible inaccuracies or mistakes
    c. Limited sensors: In order for the robot to move in any environment, it must be aware of its location and bearing. In our project we implemented such awareness using the robot's IMU module, basing our orientation mainly on odometer readings. This means that our orientation accuracy is strictly limited by the accuracy of the odometer.

Like these issues, our project was divided into two components, Object Recognition and Motion Control. A third component, the Controller, was implemented to synchronize these two components and to handle user input.

Cupinatorsim Package:

The cupinatorsim package was created to allow testing of the controller module before the walker and scanner modules were completed, and even without the robot itself. It contains simulation modules for both the walker and the scanner nodes.

The simulation nodes listen on, and publish to, the same topics as the real ones. Real world delays are simulated using rospy.sleep(). The package comes with a launcher script (cupinatorsim/launch/all.launch) to allow easy launching.

RoomScannerSim (RoomScannerSim.py)

Simulates the room scanner. The node is initialized with a distance to the simulated cup, and reduces that distance in half each scan (this is in line with the walking algorithm implemented by the controller, where, above a certain distance, the robot travels half the distance to the cup, and then readjusts its bearings).

WalkSim (WalkerSim.py)

Simulates the walker module. Waits for WalkCommands, pauses for 2 seconds, and then published a successful travel result.

Cupinator Components details

Controller Description:

The controller is responsible for handling user input and managing the other two components (Object Recognition and Motion Control).

Handling input

Handling user input is implemented by a separate node (“CupinatorCli”) that parses user input and publishes commands to a ROS topic, accordingly.
The commands available to the user are:
scan – start the room scanning (publishes a message to topic cupinator/ctrl/command)
stop – stop the mission (publishes a message to topic cupinator/ctrl/command)
status – print current robot status (no command is published; CupinatorCli listens to /cupinator/ctrl/status and thus can retrieve the state locally)
help – print a helpful message to the console
quit – terminate the application

Controlling the Robot

The controller module is built around a state machine that has the following states:
IDLE – Robot is awaiting a command
ROOM_SCAN – Robot is scanning the room for a cup
NO_CUP – Robot has nowhere to go, as cup not found
APPROACHING_CUP – Robot is making a long ride towards the cup
READJUSTING_BEARING – Robot is re-adjusting its bearings towards the cup between long rides

ARRIVING_AT_CUP – Robot is making the final ride towards the cup
AT_CUP – Robot have arrived at the cup
MISSION_FAIL – Robot failed to arrive at the cup (e.g. due to an obstacle).

When the controller receives a command from the user to initiate a scan, it moves from IDLE state to ROOM_SCAN state, calling the Object Recognition into action. A scan can be initiated only from IDLE or NO_CUP states.
When the scan completes, the Controller either moves to the NO_CUP state (scan failed) or the APPROACHING_CUP state (target was found), calling the Motion Controller in order to perform the action.
Depending on the distance to the target cup, due to the unpredictable nature of the robot's motion and the environment, the Controller might break the approach into several segments to allow adjustments to be made during approach, in this case, between each segment the Controller will move to the READJUSTING_BEARING state and invoke a special feature of Object Recognition designed to for this purpose.
Once the Motion Controller completes the approach through the last segments, the Controller moves to the AT_CUP state.
At any state the user can request the Controller to return to the idle state, in which case the Controller will notify the other components to gracefully quit their current operation and stand by for new commands.

Controller Implementation (CtrlNode.py and CupinatorCli.py):

The controller is implemented using two nodes. CupinatorCli which handles the user input is implemented in CupinatorCli.py and CupinatorCtrl which Controllers the other two components is implemented in CtrlNode.py.

CupinatorCli (CupinatorCli.py)

This node is implemented as a dispatcher, receiving user input (CupinatorCli.py:88), mapping it to a Controller command (CupinatorCli.py:90) and sending over to the CupinatorCtrl node (CupinatorCli.py::send_command). The node also listens to a status topic in order to post status updates for the user.

CupinatorCtrl (CtrlNode.py)

Initially the CupinatorCtrl node initialize a set of ROS publishers and subscribers in order to receive input and send/receive messages to/from the other components (CtrlNode.py:182-189), and sets the state machine into IDLE state.
The 3 topics CupinatorCtrl listens to are:

1) "/cupinator/ctrl/command" of type CtrlCommand, handled by command_callback (CtrlNode.py:136) – This topic represents commands sent from CupinatorCli. command_callback simply checks which command was sent and calls the respective handler
2) "/cupinator/room_scanner/result" of type RoomScannerResult, handled by scan_callback (CtrlNode.py:105) – This topic represents the result of the Object Recognition (both the regular scan and the special scan used to adjust the robot's bearing during approach)

3) "/cupinator/walk/result" of type WalkResult, ahdnled by walk_callback (CtrlNode.py:136) – These messages represent the result of the Motion Controller operation and are simply a wrapper for a Boolean indicating success or failure.

The 3 topics CupinatorCtrl publishes are:

1) "/cupinator/ctrl/status" of type CtrlStatus – Messages are published to this topic to publish the current state and status of the Cupinator (CtrlNode.py:167,174). * CupinatorCli prints these messages to stdout
2) "/cupinator/room_scanner/command" of type RoomScannerCommand – Commands to the Object Recognition are published under this topic (CtrlNode.py:47, 76, 85). The possible commands are "start" and "stop", along with a Boolean flag indicating if the command relates to a full scan or a bearing adjustment
3) "/cupinator/walk/command" of type WalkCommand – Commands to the Motion Controller are published under this topic (CtrlNode.py:67). The commands contain angular and linear distances to travel.

The flow of the Controller is implemented as set of asynchronous callbacks going back and forth between the Controller, Object Recognition and the Motion Controller.

Motion Controller Description:

The motion controller is responsible for handling all of the robot's motion including orientation and collision avoidance.

Motion Control and Orientation

Motion control operates based on requests made to the Motion Controller for angular rotation and linear motion (defining a motion vector originating in the robot's current position).
When a motion request is received, the odometer is read to indicate where the motion vector originates, then, firstly, a closed-loop angular rotation starts. The Motion Controller constantly commands the robot to rotate towards the motion vector's orientation, while constantly sampling the odometer to indicate whether the target orientation has been reached. When the target orientation is reached, the Motion Controller starts another closed-loop motion, linear this time, towards the target position. The odometer is used in the same fashion in the linear case as it is used in the angular case.
Once the target is reached, the Motion Controller publishes a message to indicate success.

Collision Avoidance

In order to avoid collisions, the Motion Controller uses a separate thread to constantly poll the robot's laser scanner. If the minimal safe distance threshold is crossed, the polling thread raises an alert flag indicating this to any active motion operations.
A motion operation is not forced to abort in case of an alert being raised. This is dependent on the context of the operation. If a motion operation aborts at the raise of the alert flag, the Motion Controller publishes a message to indicate failure to complete the request.
The alert flag is reset whenever a motion operation is initiated (angular or linear).

Motion Controller Implementation (Walk.py):

The Motion Controller is implemented as a node ("Walk") running the class Walker (Walk.py:12, 214).
The Walker controls all the operations of the Motion Controller.
The use of the Walker class enables other nodes to enjoy the abilities of the Motion Controller in a non-ROS-based fashion, enabling synchronous and reliable local motion control when the need arises.
The Motion Controller listens to 3 topics:

1) "/cupinator/walk/command" of type WalkCommand, handled by Walker::walkUntilCollision (Walk.py:192) – This topic represents requests made to the Motion Controller. They contain an angle and a distance representing a vector along which the Motion Controller should drive the robot
2) "/komodo_<id>/odom_pub" of type Odometry, handled by Walker::checkDistanceTraveled (Walk.py:64) – This topic is used to implement the sensor half of the closed-loop motion control. The odometer reading are translated by the callback to alerts to the other part of the motion control, indicating whether the motion is complete
3) "/komodo_<id>/scan" of type LaserScan, handled by Walker::checkCollision (Walk.py:41) – This topic handles the core of the Collision Avoidance of the Motion Controller. The handler processes the LaserScanner messages and decides whether there's an obstacle in too close proximity to the robot (raising an alert flag in case there is).

The Motion Controller also publishes 2 topics:

1) "/komodo_<id>/cmd_vel" of type Twist – Messages are published to this topic (Walk.py:153, 182) for common motion control
2) "/cupinator/walk/result" of type WalkResult – WalkResult messages wrap a Boolean flag indicating whether the Motion Controller succeeded in reaching its target (Walk.py:206).

Motion Control

Motion Control operation starts when a message is received via the "/cupinator/walk/command" topic, and the collisionAlert flag is reset (Walk.py:192).

The Motion Controller first initiates an angular rotation operation, calling the Walker::moveAngular method (Walk.py:194).
In this method, the new rotation target is set (Walk.py:135-136), the odometry reader is initiated (Walk.py:138-139) and the "/komodo_<id>/cmd_vel" messages are published in a loop (Walk.py:146-154) until the odometer indicates the target has been reached (Walk.py:148).
If the collisionAlert flag is raised, and the method is not instructed to ignore it (according to the "ignoreCollisionAlert" parameter), the rotation is aborted (Walk.py:149-151).
At the end of the method, the robot is instructed to halt (Walk.py:157-158) , and the odometry reader is deactivated (Walk.py:161) - Computing the orientation from the odometry reading requires too much CPU to leave running when not needed.

Next, if the WalkCommand message included a linear component in the request, the Motion Controller will initiate a linear motion operation (walk.py:198-199 by calling Walker::moveLinear.

The Walker:moveLinear method works similarly to the Walker:moveAngular method – setting a target distance (Walk.py:170-171), initiating the odometry reader (Walk.py:173-174) and publishing "/komodo_<id>/cmd_vel" messages in a loop (Walk.py:176-182). Finally, it will halt the robot's motion (Walk.py:185-186) and disable the odometery reader (Walk.py:189).

Orientation (Odometry)

The Motion Controller uses Odometry reading for its orientation. The core of this work is done in Walker::checkDistanceTraveled, which is a callback for the "/komodo_<id>/odom" topic.
When the odometer is initialized, its reference point is reset to "None", this indicates that the reference point needs to be recalculated from the next odometer reading (Walk.py:75-82).
The method will first compute the linear distance traveled away from the reference point and raise a flag if that distance matches the requested linear travel distance (Walk.py:86-90).
Next it will Convert the angular odometry reading from Quaternion to Euler while translating the range of the Euler measurements from $[-\pi, \pi]$ to $[0, 2\pi]$ (walk.py:92-107). The odometry reader will then compare its angular reading to the target distance, raising a flag if the target has been reached (Walk.py:117-123).

Collision Avoidance

Collision Avoidance is based on the constant polling of the robot's laser scanner in the callback Walker::checkCollision (Walk.py:41).
The Walker gets an arc of 512 points (distances), of which only the middle 461 points are used (Walk.py:50,52). The 25 points at each edge are discarded since the laser is blocked, and so measures cannot be made at these angles. These 461 points create an arc of $160°$, covering the front of the robot as well as the front wheels, providing full frontal collision detection.
All of the 461 points are tests against a minimal distance threshold (Walk.py:53). If and only if any point crosses the threshold, the collisionAlert flag is raised (Walk.py:54-56, 60-61).
The minimal safe distance is set to 30cm based on the measurements of the robot and empirical tests.

Object Recognition Description:

This component is handles all the tasks related to finding the target object in the environment, including capturing the images during the room scan, identifying the target object and finding its location relative to the robot.

Object recognition is divided into two parts, the Room Scanner and the Recognizer.

Room Scanner

The Room Scanner is responsible for handling communication with the other components of the Cupinator, capturing camera and depth sensor inputs, moving about the room during the scan and computing the object's location relative to the robot given a positive target identification.
The Room Scanner is implemented by the RoomScanner class and using the Walker class.

The Room Scanner starts its operation when it receives a command from the Controller. The command will either indicate that a full scan is required or just a bearing adjustment ("stationary scan").

For a full scan command the Room Scanner will spawn 2 threads. The first responsible for the Room Scanner's responsibilities and the second runs the Recognizer portion of the Object recognition.
For a stationary scan, no all operations are performed synchronously and no new threads are spawned.
In either case, if the command issued is a stop command the Room Scanner will signal all threads to stop as soon as they can (at the end of the current processing job), and return to a sort of "Standby" mode.

The main Room Scanner thread works by capturing both BGR (converted from RGB) image and depth sensor reading (represented as a grayscale image). The main thread will then send this captured data to through the secondary thread using a synchronized producer-consumer queue (in a stationary scan, this is done synchronously by the main thread). Next, the Room Scanner will rotate in order to capture the next frame of the scan (60°, as is the horizontal field of view of the Asus Xtion Pro Live used by our robot) – This part is naturally skipped during stationary scan.
Once all the frames have been captured and sent to the Recognizer, the Room Scanner waits for the Recognizer to finish processing all the images. When the Recognizer completes, the Rooms scanner uses the detected object to compute the location of the target and publishes a message with the result.

<u>Recognizer</u>
The Recognizer is a specialized implementation of the ShapeContext[1] algorithm optimized for our robot platform. Specifically, it's a variation of the algorithm with a few added preprocessing steps to account for differences between the scenarios for which the algorithm is deigned - clear shapes in a known environment - and our actual scenario -noisy images in an unknown environment.
The stages of our adaptation of ShapeContext:

1) *Filter input image by color – allow only the color of the target object to pass. Produce a binary image
2) * Inflate white regions in binary image
3) Use Canny edge detector to extract contours from binary image. Produces an array of contours
4) Uniformly Sample points from every contour.
5) *Filter our small contours below some threshold (threshold varies according to given array of contours)
6) *Compare each contour to the target's image stored in the DB using ShapeContext
    a. For each point on the contour compute the vector from that point to all other points on the contour
    b. For each point, sort all vectors into buckets stored as a matrix of angles and distances
    c. For each point, compute the cost of matching it's matrix to the matrices all the points sampled from the DB image (using χ^2 test)
    d. Create a matrix of costs and find the best matching by solving the assignment problem (using the Hungarian algorithm).
7) Store the contour with the minimal matching cost in an internal heap

Parts marked with an asterisk ('*') are not part of the original ShapeContext, but were added in order to handle some of the issues encountered during Object Recognition development.

When the Room Scanner queries the Recognizer it simply pops from the heap.

[1] - https://www.cs.berkeley.edu/~malik/papers/BMP-shape.pdf

Object Recognizer Implementation (RoomScanner.py, CupRec.py and cupRec sub-directory):

The Object Recognizer is implemented as a node (“RoomScanner”) running the class RoomScanner (RoomScanner.py:23, 417). The RoomScanner class handles all ROS communication as well as all Room Scanner responsibilities. It also controls the Recognizer.
The Recognizer is implemented in the class CupRec which uses the subdirectory cupRec. CupRec implements all the enhancements made for the original ShapeContext algorithm, while the cupRec subdirectory contains an original implementation of the algorithm as well as the DB directory (which contains images of any target for recognition). Almost all image processing (including in the cupRec directory) was done using OpenCv

The 3 topics which the Room Scanner subscribes to are:

1) “/cupinator/room_scanner/command” of type RoomScannerCommand, handled by RoomScanner::scanner_cb (RoomScanner.py: 91) – Messages received under this topic represent commands sent to the Room Scanner. These messages contain a command string – either “start” or “stop” – and a flag indicating whether a “start” command calls for a full scan or just a bearing adjustment scan (“in place scan”).
2) "/komodo_<id>/komodo_<id>_Asus_Camera/rgb/image_raw" of type Image, handled by RoomScanner::bgr_listener_cb (RoomScanner.py:58) – Messages of this topic contain RGB images taken from the robot’s camera. Due to bandwidth and other resource limitations, this topic only publishes messages when there’s a subscriber listed, and when such messages are published, other topics might be deprived of any bandwidth (starvation), and so the subscriber for this topic is only active for a minimal period of time*. Images are converted from ROS-RGB image to OpenCV-BGR image for further processing with OpenCV.
3) "/komodo_<id>/komodo_<id>_Asus_Camera/depth/image_raw" of type Image, handled by RoomScanner::depth_listener_cb (RoomScanner.py:74) – Messages of this topic contain 16bit depth data per pixel measured in millimeters. As in the previous topic, due to bandwidth and other resource limitations, this topic only publishes messages when there’s a subscriber listed, and when such messages are published, other topics might be deprived of any bandwidth, and so the subscriber for this topic is only active for a minimal period of time*.

* The minimal period of time depends on the time between the moments the first subscriber registers and the camera and its drivers to calibrate it. We arbitrarily used 1 second startup period which proved to be sufficient but not too long.

The Object Recognizer publishes to 1 topic:

- “/cupinator/room_scanner/result” of type RoomScannerResult – The messages published under this topic contain the result of a scan (full or stationary). They include the angle and distance to the cup, as well as the cost of matching the identified object to the DB image.

Room Scanner

The Room Scanner starts its operation when a message is received under the “/cupinator/room_scanner/command” topic. The message is dispatched from RoomScanner::scanner_cb to either RoomScanner::stationary_scan or to RoomScanner::room_scan using a worker thread to allow more messages (such as “stop”) to be processed.

RoomScanner::stationary_scan can be seen as a special case of the RoomScanner:room_scan, so will explain the later in detail.

The worker thread starts by looping over all frames needed for the scan (RoomScanner.py:281).
First it captures both a BGR image and a depth measurement (RoomScanner.py:291). These captured images are stored by key* in a set, and the BGR image is sent (along with the key) to the Recognizer thread (RoomScanner.py:295).
Next, the worker will reorient the robot towards the next frame to capture using the Walker (RoomScanner.py:304).
* The key is used for retrieval of the captured images later when Recognizer outputs its result.

Once all frames have been captured and the robot is back in its original orientation, the worker thread will wait for all processing to be done by the Recognizer (RoomScanner.py:311), and then it will query CupRec for the best match among all the frames.
The results will then be processed for publishing (RoomScanner.py:324). First the position of the recognized object will be determined given the matched contour produced by the Recognizer – its distance will be measured from the center of the contour (RoomScanner.py:346, 350 and RoomScanner:contour_center_in_frame) and the angle will be calculated based on the contour's center in its frame (RoomScanner.py:346-348 and RoomScanner::point_angle_in_frame).
Note, the center of the contour is taken as the arithmetic mean of the contour's points for simplicity. The poor accuracy of this method is irrelevant for this purpose, and so simplicity and performance were favored.

Lastly, the worker thread will create and publish a message with the results (RoomScanner.py:354-364).

<u>Recognizer</u>
The recognizer is implemented by the CupRec class (CupRec.py:15). During startup, CupRec loads all images in the DB folder (cupRec/DB) and computes their shape context (CupRec.py:56-60), storing the computed data in a list for later matching (CupRec.py:70-71). The specific process for computing the shape context of the image is the same for the DB images as it is for the captured images and will be detailed below.

The recognizer can either aggregate and process message asynchronously using the pair of methods CupRec::aggregate_image and CupRec::cupRec_aggregate_query, or it can process a single message synchronously (used by the stationary scan of the RoomScanner) using CupRec:cupRec_single_query.
The operations of both methods are very similar with several additions made in the asynchronous method, so we'll detail on that method.

Image processing start with a call to CupRec::aggregate_image (CupRec.py::262). CupRec computes the cost of matching each contour in the input image to the DB (CupRec.py:281) and then stores the resulting contours and the processed data in a heap for later retrieval (CupRec.py:283-290).

Computing the matching cost is peformed by CupRec:compute_contours_match_cost (CupRec.py:200). First the shape context of every contour in the input image is computed (CupRec.py:218) and then a comparison is made for each computed shape context with the shape context of the images stored in the DB. The comparison is done using the SC::diff method (CupRec.py:238-240) that compares shape contexts and which is part of the ShapeContext implementation under the cupRec directory.

To compute the shape contexts of the input image, CupRec::compute_shapecontext (CupRec.py:128) is used. First, this method creates a color filter for certain ranges of red colors (our target during the project was a red cup) and passes the input image through that filter (CupRec.py:136-138) creating a binary image. Contours are then sampled from this binary image with an adaptive threshold sample count (CupRec.py:144-154) that serves as a size-filter for objects in the binary image. If there are too many contours sampled, the minimal threshold is raised in order to limit the number of contour calculations needed. Each sampled contour is sent to SC::compute (CupRec.py:157-160) in order to compute the shape context of that contour (according to the original ShapeContext algorithm). The list of shape contexts computed is then returned.

The sampling of contours from a given binary image is done performed by CupRec::sample_contours_from_binary_img (CupRec.py:77). This method first dilates the white regions of the binary image to fix any holes added to the binary image due to noise in the original image (CupRec.py:86-89) and then the dilated image is sent to Canny edge detector to create an edge map of the image and this edge map is then broken down into separate arrays of contours by the OpenCV findcontours method (CupRec.py:90-96).
The arrays of contours are then sampled uniformly, discarding any contour which is too small to pass the sample count threshold (CupRec.py:101-120).
The resulting list of contour samples is returned for the rest of the processing operation detailed above.

Summary

We implemented object recognition in an unknown environment given limited resources and sensors heavily relying on prior information about the target object and the robot’s properties. Using a very simple database composed of generic images allows for easy extension and modification of the target object.
Using a central controller to synchronize the operations of the project enables future disassembling of the project’s components and so it will make simple to upgrade any of them in the future.

Current tests show that sensor limitations still pose a problem and improvements in that area present much promise for enhancing the abilities of this work.

Future extension can include feature extraction from database images to allow more precise filters and more robust object recognition. Also, using more sensor inputs for motion control could improve orientation and accuracy greatly.