Jupiter

A Mini-project in Computer Vision

Department of Computer Science, Ben-Gurion University of the Negev

By

Asaf Chelouche, asafchel@post.bgu.ac.il

Oded Lacher, lachero@post.bgu.ac.il

Table of Contents

Project Description

This project has two major aspects: computational vision ("CV") and working with ROS – Robot Operating System. The CV portion consists of recognizing a red ball in a room, approaching it, grabbing it and placing it in a basket. This process repeats until no more balls are present in the room.

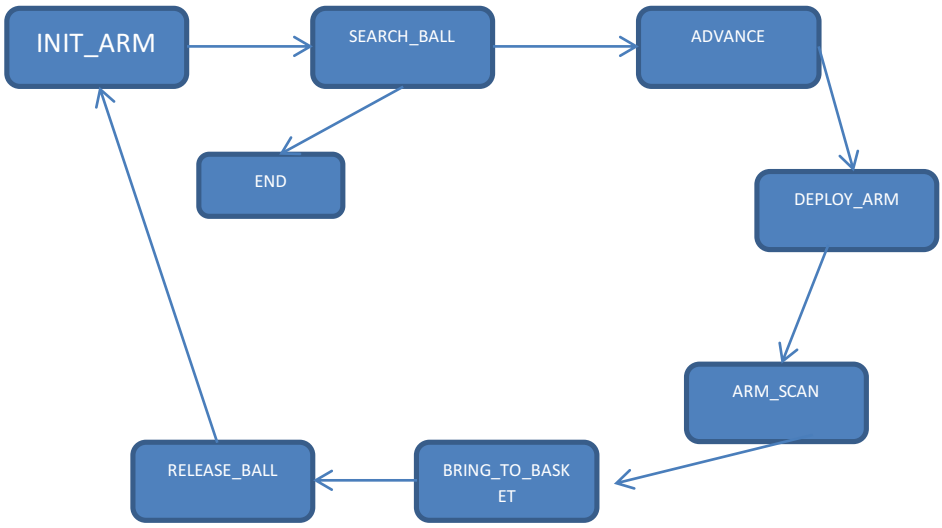
First we had to choose an algorithm for ball recognition. We decided to recognize a ball based on its shape and color, but we did not want to limit ourselves to the spherical algorithms alone. The reason for this decision was the intention of providing a more generic recognition algorithm, which would work on other objects with a non-circular nature. After trying several techniques, we chose the Blob detection algorithm, which is implemented in OpenCV. The main reason for this decision was the opportunity to recognize more than a ball. Other than the generic use, this was also proved useful as in real world situations, where the picture is far less than perfect as in a simulation, since we need to handle different objects that aren't the ball itself. This is beacuse there are lighting differences which originate from self-shading and various color temperatures of various lighting fixtures. Another advantage we saw in this algorithm was that it's less complicated than others, so the processing time of a single frame is quite fast. We will further elaborate in details the algorithm and its steps.

We chose to work on the Komodo robot. We used ROS to control the robot's behavior and the implementation of the CV algorithm described above.

This project is based on the "Pluto" project by Dina Svetlitsky and Alex Kovalchuk.

Program Architecture

State machine



Every block in the diagram represents a state. A transition from one state to the next occurs only when the previous state doesn't hold several invariants.

1. **INIT_ARM** – The first stage of the program, is to position the arm in a starting position, on the side of the robot, so it won't interfere with the upper Asus camera. Only after the hand is in the starting position we can start searching for the ball.
2. **SEARCH_BALL** – In this stage the robot turns in place, looking for a ball with the Asus camera. If the robot completed a 360-degree turn, and found no balls, it transitions to the END state. Otherwise, when the ball's horizontal position is at a certain section of the frame, it transitions to the ADVANCE state.
3. **ADVANCE** – the robot moves forward until the ball's center reach the lower 10% of the frame, then it transitions to the DEPLOY_ARM state. Furthermore, it stops any image processing conducted.
4. **DEPLOY_ARM** – deploys the arm in a downward-looking position, and then transitions to the ARM_SCAN state.
5. **ARM_SCAN** – resumes image processing. Moves the robot forward until the ball's reaches a height of 10 pixels from the frame's bottom. If needed, combines left or right angular movement to keep the ball fairly centered. When these conditions are met, the arm is lowered towards the ball. When it reaches a certain position, it is fair to assume that the ball is in a suitable position, a final adjustment "droop" is made, and the fingers

are closed, considering pressure applied on the ball. When the fingers are closed, it transitions to the BRING_TO_BASKET state.

6. **BRING_TO_BASKET** – positions the arm on top of the basket. Physically, this is the same position that is set in INIT_ARM. When met, it transitions to the RELEASE_BALL state.
7. **RELEASE_BALL** – releases the ball, so it falls into the basket. When the fingers are reported to be in the open position, transitions to the INIT_ARM state.
8. **END** – ending state, program terminates.

ROS nodes

1. state_machine (`state_machine.py`) – implements the state machine diagram. Controls program flow and transition between states. Subscribes to ROS topics to be alerted of changing conditions, and publishes to ROS topics to facilitate new states.
2. detector (`detector.py`) – implements the CV element of the project. Uses Blob Detection to find the ball and report its position. Transitions between the Asus camera and arm camera based on an incoming message in a custom ROS topic. Assumptions – the ball is in a certain shade of red, and it is placed on a pedestal of a certain height. We filter out the red shades with two filters, in order to find both dark-red and light-red shades, since using a single filter which is more general would also exhibit false detections.
3. arm_movement (`arm_movement.py`) – manages the movement of the robot's arm between its various states. When the robot is in the ARM_SCAN state, it becomes a sort of local state machine that directly dictates the robot's movement. When moving the arm, the joints are divided into two groups, each of them moves all respective joints simultaneously. This is to prevent the arm from colliding with the elevator.
4. robot_movement (`robot_movement.py`) – controls the robot's wheels, i.e. movement and stopping in place.
5. finger_movement (`finger_movement.py`) – an implementation of a ROS service, to enable opening and closing the fingers. When closing the fingers, the pressure which is applied on the servos is taken into consideration when deciding if the ball is firmly grabbed. The code was adapted from a snippet by Boahn Lou (`loubohan@gmail.com`).

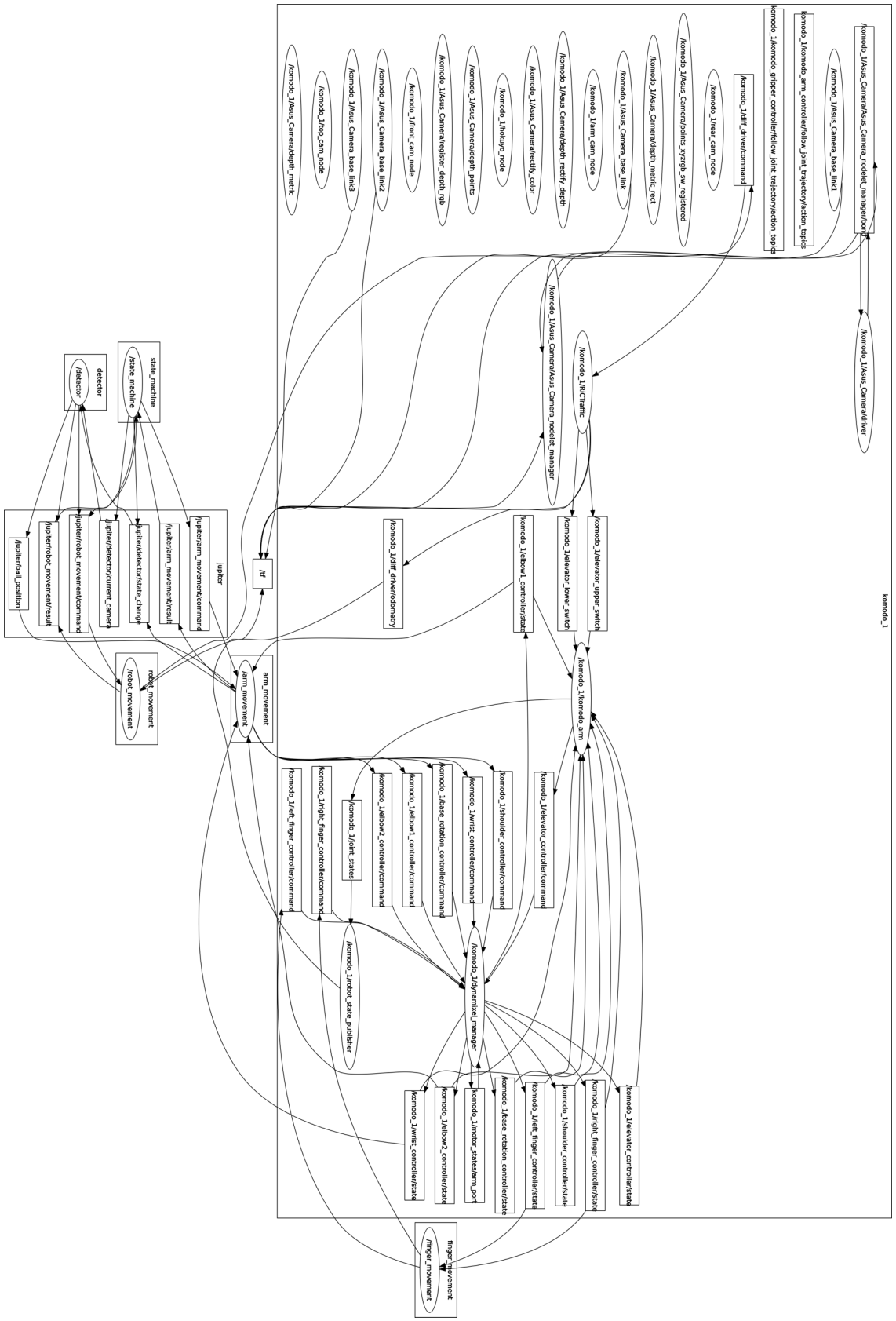


Figure 1: the system's ROS graph

Custom ROS topics, services and messages

Topics

1. /jupiter/arm_movement/command – send arm movement commands to arm_movement node.
2. /jupiter/arm_movement/result – await reports from arm_movement node that the arm is in the requested position.
3. /jupiter/detector/current_camera – instruct the detector node to use either the Asus camera or arm camera.
4. /jupiter/detector/state_change – instruct the detector node whether to process images or not.
5. /jupiter/robot_movement/command – send robot movement commands to robot_movement node.
6. /jupiter/robot_movement/result - await reports from robot_movement node that the robot has stopped in place.
7. /jupiter/processed_image – with respect to the currently active camera, publishes its view with an overlay of green rings, marking all possible balls. The selected object, which is determined to be the ball, is marked with a further blue ring.
8. /jupiter/processed_image_bw – same as topic no. 7, but the image itself is the binary image that results from the color filtration.
9. /jupiter/ball_position – used to report the ball's position in the arm camera's view at the ARM_SCAN state. These reports flow directly to the arm_movement node.

Services

1. fingers – used to instruct the fingers to open or close. Parameters and returned value:
    a. Action – (string) CLOSE_FINGERS or OPEN_FINGERS.
    b. Value – (float64) pressure value to reach.
    c. Reading – returned value (float64). Ignored, required in services.

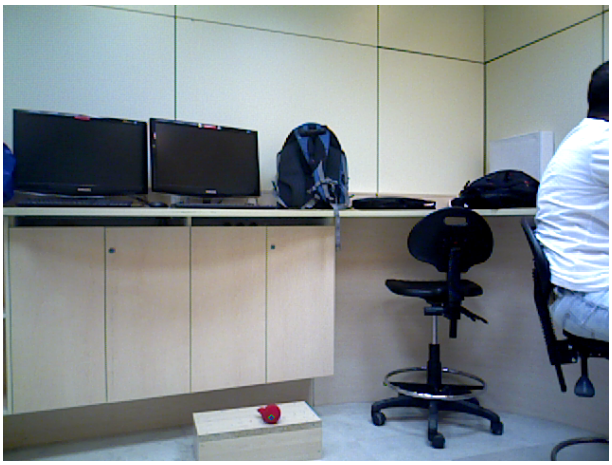
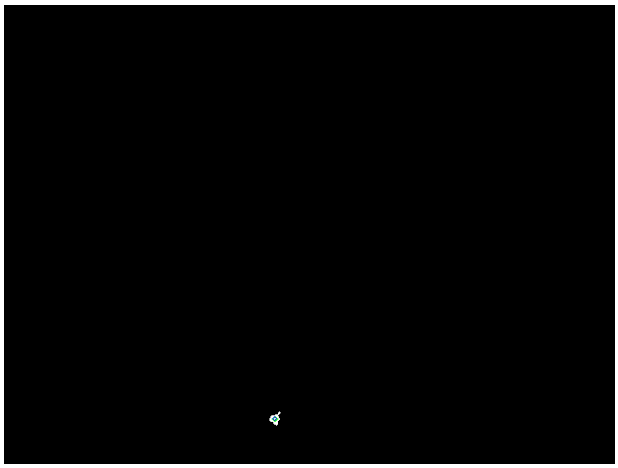
Messages

1. BallPosition.msg – a ball position message, as described in topic no. 9. Included fields:
    a. detected – (bool) a Boolean indication of successful detection.
    b. x – (uint32) the ball's center point x value.
    c. y – (uint32) the ball's center point y value.
    d. radius – (uint32) the ball's radius.
    e. img_width – (uint32) the arm camera's image width.
    f. img_height – (uint32) the arm camera's image height.

Computer Vision Element

As described above, we used the blob detection algorithm provided by OpenCV, with several pre-processing steps to make the algorithm more precise.

1. Gaussian Blur - blurs the image and provides a more even distribution of the red shades, for more accurate red shade detection later on. Uses a 9×9 kernel.
2. Red shades detection - we use two different masks, for lighter and darker shades of red, and combined the results together bitwise. We used different masks for the Asus camera and the Arm camera, as they have different image sensors, so they render colors slightly differently.
3. Conversion to grayscale.
4. Binary image creation – for better blob detection.
5. Blob detection – using an OpenCV method. We fine-tuned various parameters to have more accurate detection, such as circularity and area (in pixels).
6. Keypoint filtration – the Blob Detection outputs a list of keypoints. For the Asus camera, only those that are in the bottom half of the frame are considered as candidates. The keypoint with the largest radius is determined to be the ball.

Figures 2a, 2b: an overlay of the keypoints and selected ball over the raw RGB feed from the Asus camera (left) and binary image after red shades filtration (right)

Figures 3a, 3b: an overlay of the keypoints and selected ball over the raw RGB feed from the arm camera (left) and binary image after red shades filtration (right). Notice that the image is upside-down.

Challenges faced

1. Between the running the code in a simulation ("gazebo") and on the actual robot, different launch files are used, which result in different topic names. This is adjusted with launch parameters and an adjustment function.
2. Also, the real and simulated image sensors have different resolutions.
3. The real arm camera output is inverted upside-down.
4. Low refresh rate of the cameras – around 10Hz.
5. When running the code on the actual robot, there are four major real-world implications:
    a. Friction between the smooth floor and rubber wheels – prevents very accurate control of the robot's motion with the wheels. Dictates use of very low speeds to make the motion as accurate as possible.
    b. Lighting conditions and ball self-shading – requires very fine-tuning of the color masks for specific settings.
    c. Lag time between a movement command that is sent to the wheels and its physical manifestation.
    d. Arm movement accuracy.
6. Accommodation for balls of various sizes – the solution is to use a pre-written service that takes into consideration the pressure that is applied on the finger servos.
7. Finding a general and robust CV algorithm that doesn't heavily rely on the objects' circularity.

Areas for further improvement

1. When starting to search for the ball in the SEARCH_BALL state, if the ball is already in the frame, move to the appropriate direction rather than arbitrarily moving clockwise.
2. Also, use the rear-facing camera at the same time to cut the search time in half (scans the scene as two 180-degree sectors instead of a single 360-degree sector).
3. When advancing towards the ball using the arm camera, rely more on the physical features of the target object rather than on its color properties. This opens a possibility to disqualify small objects that appeared circular from afar, but aren't in reality.
4. Multiple color masks for target objects of various colors, i.e. red balls and blue cubes in the same scene.
5. Use depth readings from depth sensors in the Asus and arm cameras.
6. Use front facing laser collision-detector to determine the appropriate stopping position from the pedestal, on which the ball is placed.
7. Move the robot faster on rougher surfaces, e.g. asphalt or sidewalks.
8. Use the MoveIt package to move the arm in a more natural way than controlling individual joints.
9. Pre-scan the scene to detect all balls, and then determine the most time-efficient manner in which to collect them.

Demonstration Videos

1. A short introduction: https://www.youtube.com/watch?v=UtKnoQu_9Ho
2. Execution from start to finish: https://www.youtube.com/watch?v=GswU1YVCJvo

Technical notes

To run the project in gazebo (simulation setting):

```
$ roslaunch jupiter jupiter_sim.launch
```

To run the project on the actual robot, first copy all of the source code to the robot itself to the directory `~/catkin_ws/src/jupiter`. Then, run `catkin_make` in `~/catkin_ws` to generate the code for the services and messages. Afterwards, launch with:

```
$ roslaunch jupiter jupiter.launch
```

All source code is available at `github.com/asafch/komodo`.