# Contest II Report

GROUP 9

Gary Leung, 1002177155

Litos (Hanze) Li, 1002526493

Justin (Zhaocong) Yuan, 1002352777

Jojo (Yizhi) Zhou, 1003002396

March 25, 2020

# Contents

# 1 Problem Definition

## 1.1 Objective

Navigation and localization is essential for mobile robots to assist or replace human labour in the industrial setting. A basic localization problem was solved in the first contested with laser range finders. To take it one step further, we can use modern camera sensors such as Microsoft Kinect to retrieve more image data, which allows the robot to understand the environment more accurately and plan its actions accordingly. Robots with the ability to recognize objects have a wide range of applications from warehouse assistance (i.e., identify package labels) to self-driving (i.e., reading traffic signals).

The objective for this contest is to efficiently and accurately navigate the Turtlebot and identify five objects placed at different locations in a known environment. The object identification must be achieved using the RGB camera on the Kinect sensor to perform SURF feature detection. Three of these them contains unique images for recognition, the forth one contains a duplicated image, and the last one is left blank. While the mapping part is taken care of by a provided 2D set map and coordinates of all five objects, it is our task to localize and navigate the robot as well as identifying the objects.

## 2 Strategies

To tackle this contest, the turtlebot will first localize itself inside the map through manually manipulate the base of the robot until the robot is able to localize itself based on laser readings. In simulation, the robot localize itself in the Gazebo environment by calling *spinOnce*() at the beginning to localize itself and the initial robot pose to be recorded from AMCL package.

Then for every object inside the map with known coordinates, we first come up with an optimal traversal order using classical travelling salesmen algorithm for shortest travel path and then use *moveToGoal*() function (provided in the given Navigation template) to move to the location of interest one by one.

At each object, we look at the image from k different viewpoints and match the scene image with object image and store the result in a list. In practice, we choose k=3 to balance efficiency and accuracy. After traversing all objects, the robot will return to the starting position and determine the labels of each scene image using our Probabilistic State Estimation described in section 3.2.2.1. Finally, the robot will return a file containing the position and matched objective image for each object.

# 3 Robot Design and Implementation

## 3.1 Sensory Design

Multiple sensor units are available for use in the TurtleBot, including 3 cliff sensors, 2 wheel drop sensors, 3 front bumpers, a gyroscope, an odometer, an RGB camera, a depth sensor, and a microphone array. In this contest, three groups of sensors are utilized in the navigation algorithm of the TurtleBot, including the depth sensors, the odometer, and the set of bumpers.

In this contest, the robot makes use of the following three types of sensors:

- RGB camera
- Odometry

gather necessary information for localization, path planning and object detection, gyro and wheel encoders for odometry, a depth sensor (laser) and a RGB camera.

### 3.1.1 Odometry

Built-in gyro scope and wheel encoders are leveraged in determining the orientation and position information of the robot relative to its initial pose. The robot first defines a right-handed coordinate frame centered at itself at the beginning of each run and the pose information is accessed throught the AMCL server, which implements the adaptive (or KLD-sampling) Monte Carlo localization approach using a particle filter to track the pose of a robot against a known map. We did not directly use Wheel odometry since reading from wheel encoders could be error-prone due to wheel slippage and measuring errors etc. and this error would accumulate through navigation.

### 3.1.2 RGB Camera

The robot uses the RGB Camera on Microsoft Kinect 360 to capture and identify images on object boxes. The RGB camera used consists of red, green and blue channels. The brightness on each channel are measured separately with a range from 0 to 255. Each pixel is thus represented by a 3D vector where each dimension is one channel's brightness. The measurement of RGB camera is later assessed in *imageCallback* and stored in a matrix variable called *img* for comparing with other template images.

## 3.2 Algorithm Design

### 3.2.1 Navigation

Travelling Salesmen problem is a classical optimization problem for traversing several known destinations in shortest path/time. Our Navigation algorithm makes use of the known map to decide on a traversal order for each object and take images of the object before comparing them with template images. The general flow can be seen in Figure 1 and more detailed procedure is summarized in the following:

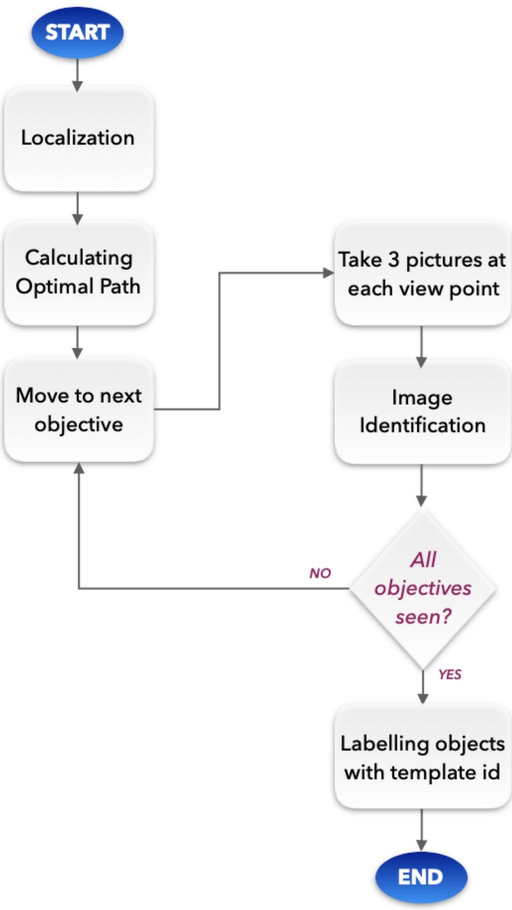


Figure 1: Navigation Flow Chart

1. We first begin by localizing our starting position (as shown in Fig 2. This is done through a *spinOnce*() call while the robot stays static, the initial robot pose will be recorded from the AMCL package (which ensures consistency for localization and navigation later on).

2. Once we localize our starting position, we use a brute-force solution to the travelling salesman problem to determine the optimal traversal order for all the boxes. Given the relatively small exploration space with only 5 objects, we enumerate all permutations and find the one with the least total distance traversed. This method allows quick and accurate high-level planning with negligible overhead.

3. We use the $moveToGoal()$ function to navigate to our points of interest. In this case, after determining the traversal order, we move to each box location one by one.

4. At each box location, we look at the image from several different viewpoints for better

accuracy (as shown in Fig 3. These viewpoints face towards the image and surround the center of the box with tunable offsets. Before we start navigation, we calculate all these viewpoints from the given box poses and cache them. In addition, view points for a particular box can be traversed in either forward or backward order, we pick the one with smaller distance from the starting point to the last visited box.

5. Once we finish traversing all boxes in order and have seen all the images, we then return to our initial starting pose, log out image identification results and terminate the program.

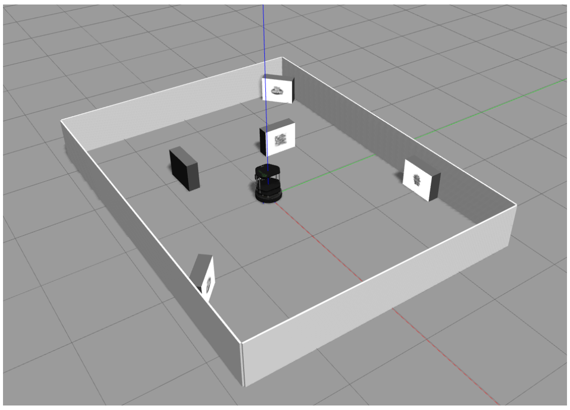

Figure 2: Robot localizes itself upon starting

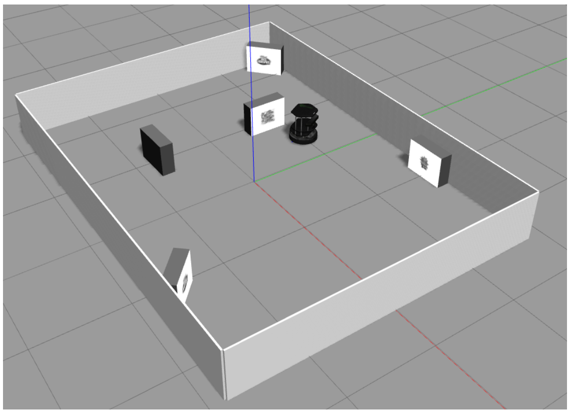

Figure 3: Robot at a viewpoint looking at image on box

#### 3.2.2 Image Identification

Image Identification is performed after the turtlebot took pictures at each view point around the target object. The scene images taken by the webcam are then feed in to the *imagePipeline.cpp* file for processing and comparing the scene image to the object image for updating an identification vector *templateIDs* <> which stores the identified scene image id at each given coordinate.

The general flow of image pipeline can be seen in Figure 4.

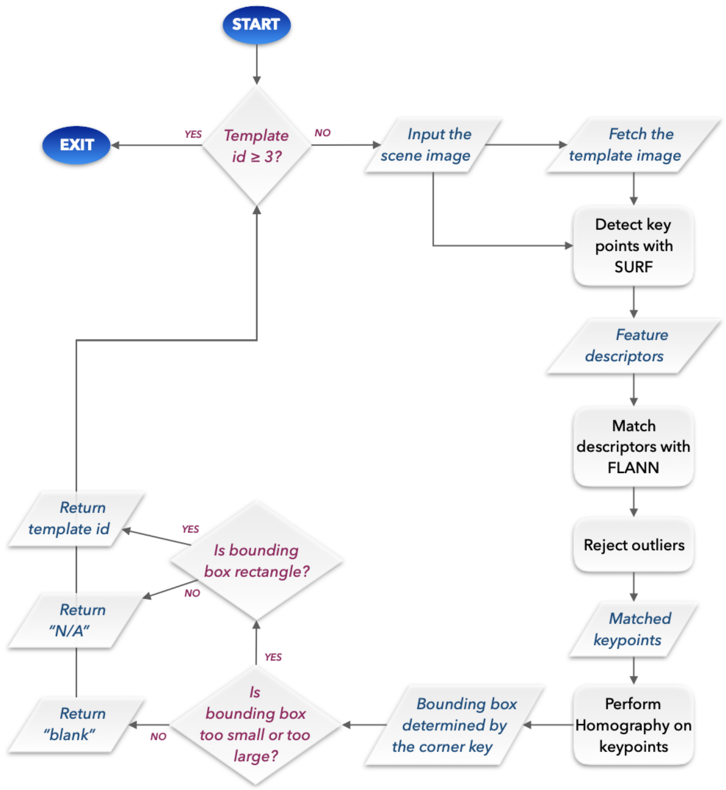


Figure 4: Image Identification Pipeline

In terms of comparing two images and identifying if they are same, we use Speeded Up Robust Features (SURF) to obtain keypoint descriptions from object (template) images and scene (seen) images. SURF computes the determinant of the Hessian matrix for image gradients as a measure of local change around points of interest. The extracted features are invariant against different image transformation including scale, rotation and translation, therefore the robustness is ensured and the algorithm is able to identify the images quite well given some degree of variations.

More detailed steps for deciding which objective image (or blank image) does the scene image belongs to is as following:

1. Compare seen picture with one of the given template, detect the keypoints and calculate descriptors using SURF Detector
2. Match such descriptor vectors using FLANN matcher
3. Calculate max and min distances between keypoints
4. Reject outliers (Draw only "good" matches with distance smaller than $3 \cdot min\_distance$)
5. Get corresponding matched keypoints and transform keypoints in the object to the keypoints in the scene using the $findHomography$ function.

6. Get the bounding box area by calculating length and width using corner points in the transformed keypoints set.

7. Check if the bounding box is a rectangle and the area is valid, if yes, then we find a good match between seen picture and this template.

8. Iterate through step 1-7 with all 3 templates so that:

   - if we find a good match between one of the templates and the picture captured by the turtlebot, we return the template id, or
   - if we find that the picture is highly unlikely to be match any template, we return "blank", or
   - if we are neither confident in a good match or a blank image, we return "n/a".

   The matching result is stored in the *imagePipeline* and the final matching decision is made probabilistically, as explained in Section 3.2.2.1.

As can be seen in Fig 5 and 6, the SURF algorithm will fit a valid bounding box wrapping the scene objective very well while not be able to find a valid bounding rectangle when comparing the scene object with mismatched template.

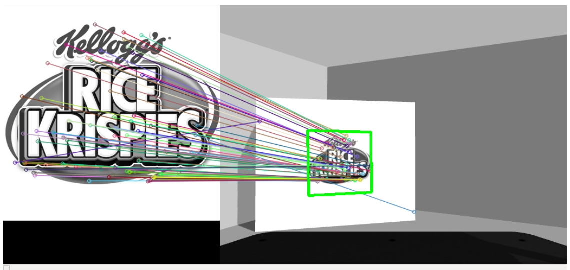


Figure 5: Matching Result

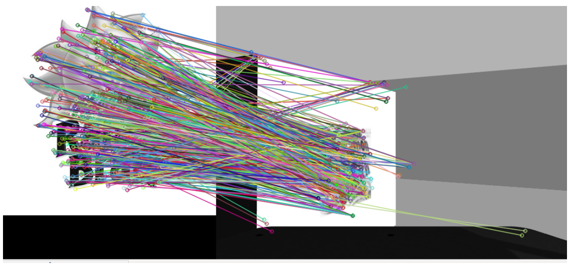

Figure 6: Mismatching Result

#### 3.2.2.1 Image Identification as Probabilistic State Estimation

Motivated by the image detection results from multiple view points, we look for a way to combine these incremental observations by using state estimation/belief update. Besides, given the constraints that only 4 out of 5 box has a valid image with only 1 repetition, each observation on a box is intrinsically correlated to estimation of all boxes, which we can use to accelerate image identification given only partial observations.

We formulate the image identification problem as online probabilistic state estimation and conduct approximate likelihood maximization to retrieve the desired image categorization. When the robot navigates to a view point and processes an image, it updates a state/belief $s$ which is a $5 \times 4$ matrix. The i-th row $s[i]$ is box $i$'s belief on what image it has. $s$ can be modeled as the joint discrete variables under the constraints that except one repetition, each variable should take a distinctive value.

Our objective is to infer the posterior of the belief $s$ given a series of observations $\{o^t\}_{1:T}$

$$s^* = \underset{s}{\operatorname{argmax}}\, P(s|o^1, o^2, ..., o^T)$$

To use the online observations, we decompose and approximate the posterior

$$\begin{aligned} P(s|o^1, o^2, ..., o^T) &= P(o^1, o^2, ..., o^T|s)P(s)/P(o^1, o^2, ..., o^T) \\ &= \prod_{t=1}^{T} P(o^t|s)P(s)/\prod_{t=1}^{T} P(o^t) \\ &= \prod_{t=1}^{T} \frac{P(o^t|s_i, s_{-i})}{P(o^t)} P(s) \\ &= \prod_{t=1}^{T} \frac{P(s_i|o^t)P(s_{-i}|o^t)}{P(s)} P(s) \end{aligned}$$

where we have assumed observations are independent, the state can be decomposed to joint state of box $s_i$ (observed at time $t$) and state of other boxes $s_{-i}$, we further assume $s_i$ and $s_{-i}$ is independent for computational purposes, but technically they are subjected the the constraint mentioned above should the joint cannot be factored individually. Taking the log likelihood and assuming a uniform prior $P(s)$, we have

$$s^* = \underset{s}{\operatorname{argmax}} \sum_{t=1}^{T} logP(s_i|o^t) + logP(s_{-i}|o^t)$$

To carry out the optimization, we simply accumulate the logits $logP(s_i|o^t)$ and $logP(s_{-i}|o^t)$ over each incoming observation, and take the label-wise argmax at the end. The logits can be approximated by the matched area btween each captured image $I_t$ and the corresponding template images $I_i$.

$$logP(s_i|o^t) \sim Area(I_t) \cup Area(I_i)$$

For $logP(s_{-i}|o^t)$, we can simply use the negative of $logP(s_i|o^t)$ since it's an valid upper bound.

$$
\begin{aligned}
P(s_i, s_{-i}|o^t) &< 1 \\
logP(s_i|o^t) + logP(s_{-i}|o^t) &< 0 \\
logP(s_{-i}|o^t) &< -logP(s_i|o^t)
\end{aligned}
$$

Lastly, to account for accumulating errors in state estimation, we can use a exponential average to discount new observations, controlled by weight $w$, and arrive at the following joint state update rule for each new observation on box $i$

$$
\begin{aligned}
s[i] &\leftarrow (1-w) * s[i] + w * logP(s_i|o^t) * \alpha \\
s[k] &\leftarrow (1-w) * s[k] - w * logP(s_i|o^t) * \beta, \quad k \neq i
\end{aligned}
$$

parameters $\alpha$ and $\beta$ further controls the update rate, note that $\beta < \alpha$ since our factorization violates the constraint and hence the state updates to unobserved boxes should be smaller.

# 4 Future Recommendations

With the current setting, the turtlebot knows exactly the map as well as where the object image located in the map, which is a strong assumption and such prior information may not be accessible in real-world scenario. In the future, the algorithm can be extended to automatically localize and mapping before navigating to and identifying the desired object. This could be done using SLAM simultaneously with the current algorithm and image detection can also be utilized along the way the turtlebot navigate.

It is also noticed that at the beginning of the navigation, we have to manually move the robot in order for it to localize itself, this process can be automated in the future so that the whole process can run without any human intervention.

Furthermore, during the navigation phase, the efficiency and accuracy could be further improved since the current algorithm took k determined number of pictures at each target location and process the image using the designed algorithm which consumes extra times and some of the pictures taken might not be great. Instead, we propose that in the future we can quickly pre-process and identify the pictures taken at the same object and only take enough number of pictures untill we are confident about correctly identifying the object.

## Appendix A - Team Contributions

Table 1 is the contribution of each teammates in Contest1.

Table 1: Contest 2 Team Contribution.

| | Gary | Litos | Justin | Jojo |
|---|---|---|---|---|
| Overall Design | × | × | × | × |
| Navigation Algorithm | × | | × | |
| Image Processing Algorithm | | × | | × |
| Pipeline | × | × | × | × |
| Debugging | × | | × | × |
| Simulation and Testing | × | | × | |
| Report | × | × | × | × |

## Appendix B - Header Files

## boxes.h

```
#pragma once

#include <opencv2/highgui/highgui.hpp>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <vector>

class Boxes {
  public:
    std::vector<cv::Mat> templates;
    std::vector<std::vector<float>> coords;
  public:
    bool load_coords();
    bool load_templates();
};
```

**imagePipeline.h**

```
#pragma once

#include <image_transport/image_transport.h>
#include <std_msgs/String.h>
#include <opencv2/core.hpp>
#include <cv.h>
#include <cv_bridge/cv_bridge.h>
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include <boxes.h>
#include <vector>

// Matching status
#define RAISIN 0
#define CINNAMON 1
#define RICE 2
#define AMBIGUITY -1
#define BLANK -2
#define IMAGE_TOPIC "camera/rgb/image_raw" //
    kinect:"camera/rgb/image_raw" webcam:"camera/image"


using namespace cv;
using namespace std;
using namespace cv::xfeatures2d;

// Criterias
#define MaxArea 40000.
#define MinArea 1000.
#define MaxGoodArea 60000.
#define MinGoodArea 2000.

// Sizes
#define NumTargets 3
#define NumBoxes 5
#define NumStatus 5
// #define NumViewPoints 5

```

```
// Logit update
#define alpha 2.
#define beta 1.5

class ImagePipeline
{
private:
    cv::Mat img;
    bool isValid;
    image_transport::Subscriber sub;
    vector<int> templateIDs;                          // stores final IDs
    vector<vector<float>> logits; // 1st index: boxID, 2nd index-2: target

public:
    void imageCallback(const sensor_msgs::ImageConstPtr &msg);
    int getTemplateID(Boxes &boxes);
    float getArea(std::vector<Point2f> scene_corners, cv::Mat img_object);
    float performSURF(cv::Mat img_scene, cv::Mat img_object);

    // To be called at each img capture
    void updateLogits(Boxes &boxes, int boxID);

    // To be called at the end
    void finalizeTemplateID(int boxID); // updates this.templateIDs from
   this.logits
    void finalizeTemplateIDs();         // wrapper for the
   finalizeTemplateID

    // Constructor
    ImagePipeline(ros::NodeHandle &n) : templateIDs(NumBoxes, -1)
    // logits(NumBoxes, vector<int>(NumStatus, 0))
    {
        image_transport::ImageTransport it(n);
        sub = it.subscribe(IMAGE_TOPIC, 1, &ImagePipeline::imageCallback,
   this);
        // reset all image template ids
        for (int box = 0; box < templateIDs.size(); box++)
        {
            templateIDs[box] = -1;
        }
        isValid = false;
```

```

        for (int i = 0; i < NumBoxes; i++) {
            vector<float> v(NumStatus, 0.0);
            logits.push_back(v);
        }
    }


    // Utilities
    inline void setTemplateID(int templateID, int boxID)
    {
        templateIDs[boxID] = templateID;
    }

    inline int box_to_ID(int boxID)
    {
        return templateIDs[boxID];
    }

    inline std::string ID_to_name(int templateID)
    {
        std::string name = "N/A";
        switch (templateID)
        {
        case RAISIN:
            name = "Raisin Bran";
            break;
        case CINNAMON:
            name = "Cinnamon Toast Crunch";
            break;
        case RICE:
            name = "Rice Krispies";
            break;
        case BLANK:
            name = "Empty Surface";
        default:
            name = "Empty Surface";
        }

        return name;
    }
```

```

    inline std::string box_to_name(int boxID)
    {
        return ID_to_name(box_to_ID(boxID));
    }
};
```

### navigation.h

```cpp
#pragma once

#include <nav_msgs/MapMetaData.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Twist.h>
#include "ros/ros.h"
#include <vector>
#include <map>
#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>
#include <nav_msgs/OccupancyGrid.h>
#include <std_msgs/String.h>
#include <string>


#include <robot_pose.h>
#include <imagePipeline.h>
#include <boxes.h>
#include <math.h>
#include <iostream>
#include <fstream>


class Navigation {
  public:
    Boxes boxes;

    Navigation(ros::NodeHandle &n, Boxes &_boxes, int n_view_points) :
    robotPose(0, 0, 0), imagePipeline(n){
      // map stuff
      mapSub = n.subscribe("/map", 1, &Navigation::mapCallback, this);
      num_view_points = n_view_points;

      // get boxes handle
      boxes = _boxes;

      // localization and image stuff
      amclSub = n.subscribe("/amcl_pose", 1, &RobotPose::poseCallback,
```

```
    &robotPose);

      // manuall move robot
      vel_pub =
    n.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);
    }

    void traverseAllBoxes();
    int getCurrentBoxId();
    void logImageIDs();

  private:
    int width;
    int height;
    float resolution;
    double angular_max = M_PI / 6;
    std::vector<float> origin;
    // int[] map;

    RobotPose robotPose;
    ros::Subscriber mapSub, amclSub;
    ros::Publisher vel_pub;
    ImagePipeline imagePipeline;

    int num_view_points;
    std::map<int,std::vector<std::vector<float>>> box_view_points;



    static bool moveToGoal(float xGoal, float yGoal, float phiGoal);
    void mapCallback(const nav_msgs::OccupancyGrid::ConstPtr& msg);
    void getViewPoints(std::vector<std::vector<float>> coords);
    std::vector<int> getTraversalOrder(std::vector<std::vector<float>>
    coords, int starting_pos);
    void localizeStartingPose();
    int getDist(std::vector<float> coor1, std::vector<float> coor2);
    void traverseBox(int box_idx);
    void publishVelocity(float angular, float linear, bool spinOnce =
    false);

};
```

### robot$_{pose.h}$

```
#pragma once

#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <vector>

class RobotPose {
  public:
    float x;
    float y;
    float phi;
  public:
    RobotPose(float x, float y, float phi);
    void poseCallback(const geometry_msgs::PoseWithCovarianceStamped& msg);
    std::vector<float> toCoords() {
      std::vector<float> coord{x,y,phi};
      return coord;
    }

};
```

# Appendix C - Source Files

### navigation.cpp

```
#include <navigation.h>

bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal){
  // Set up and wait for actionClient.
    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base", true);
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server to come up");
    }
  // Set goal.
    geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(phiGoal);
    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose.header.frame_id = "map";
```

```
    goal.target_pose.header.stamp = ros::Time::now();
    goal.target_pose.pose.position.x =  xGoal;
    goal.target_pose.pose.position.y =  yGoal;
    goal.target_pose.pose.position.z =  0.0;
    goal.target_pose.pose.orientation.x = 0.0;
    goal.target_pose.pose.orientation.y = 0.0;
    goal.target_pose.pose.orientation.z = phi.z;
    goal.target_pose.pose.orientation.w = phi.w;
    ROS_INFO("Sending goal location ...");
  // Send goal and wait for response.
    ac.sendGoal(goal);
    ac.waitForResult();
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
        ROS_INFO("You have reached the destination");
        return true;
    } else {
        ROS_INFO("The robot failed to reach the destination");
        return false;
    }
}

void Navigation::mapCallback(const nav_msgs::OccupancyGrid::ConstPtr&
   msg) {
    width = msg->info.width;
    height = msg->info.height;
    resolution = msg->info.resolution;
    ROS_INFO("Map: (%d, %d) retrieved", width, height);
    // only get map once
    mapSub.shutdown();
}

void Navigation::getViewPoints(std::vector<std::vector<float>> coords) {
    float margin = 0.5;
    // float box_size = 0.0;
    int i = 0;

    for(auto b: coords) {
        std::vector<std::vector<float>> view_points;
        float x = b[0];
        float y = b[1];
        float angle = b[2];
```

```

        // offset center of view points
        // float r = sqrt(x*x + y*y) + box_size;
        // x = r * cos(angle);
        // y = r * sin(angle);

        // generate view points
        float ang_delta = M_PI / (num_view_points + 1);
        for (int v = 0; v < num_view_points; v++) {
            float view_ang = angle - M_PI/2.0 + (v+1)*ang_delta;
            float view_x = x + margin * cos(view_ang);
            float view_y = y + margin * sin(view_ang);

            // robot angle
            float robot_view_angle = view_ang - M_PI;


            std::vector<float> view_pose{view_x, view_y,
    robot_view_angle};
            view_points.push_back(view_pose);
        }

        // add to current box
        box_view_points.insert(
            std::pair<int,std::vector<std::vector<float>>>(
                i, view_points
        ));
        i++;
    }
    ROS_INFO("Finished obtaining %d view points", num_view_points);
}


std::vector<int>
    Navigation::getTraversalOrder(std::vector<std::vector<float>> coords,
    int starting_pos){
    // Modified travelling salesman problem solution from:
    https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

    // store all vertex apart from source vertex
    std::vector<int> vertex;
```

```
    for (int i = 0; i < coords.size(); i++)
        if (i != starting_pos)
            vertex.push_back(i);

    // store minimum weight Hamiltonian Cycle.
    int min_path = INT_MAX;
    std::vector<int> min_vertex = vertex;
    do {
        // store current Path weight(cost)
        int current_pathweight = getDist(coords[starting_pos],
    coords[vertex[0]]);

        // compute current path weight
        for (int i = 0; i < vertex.size() - 1; i++) {
            current_pathweight += getDist(coords[vertex[i]],
    coords[vertex[i+1]]);
        }
        current_pathweight += getDist(coords[vertex[vertex.size()-1]],
    coords[starting_pos]);

        // update minimum
        if (current_pathweight < min_path){
            min_path = current_pathweight;
            min_vertex = vertex;
        }

    } while (next_permutation(vertex.begin(), vertex.end()));
    ROS_INFO("Finished obtaining traversal order");

    // Append starting coord to end of vector.
    vertex.push_back(starting_pos);
    return vertex;
}

int Navigation::getDist(std::vector<float> coor1, std::vector<float>
    coor2){
    return pow(pow((coor1[0] - coor2[0]),2) + pow((coor1[1] -
    coor2[1]),2),1/2);
}

void Navigation::publishVelocity(float angular, float linear, bool
```

```
    spinOnce /*= false*/)
{
    ROS_INFO("Publishing - Linear: %f, Angular: %f", linear, angular);
    geometry_msgs::Twist vel;
    vel.angular.z = angular;
    vel.linear.x = linear;
    vel_pub.publish(vel);

    if (spinOnce)
    {
        ros::spinOnce();
    }
}

void Navigation::localizeStartingPose() {
    // rotate for 5 secs for better localization
    publishVelocity(angular_max, 0.0, true /* SpinOnce */);
    ros::Duration(5).sleep();
    publishVelocity(0.0, 0.0, true /* SpinOnce */);

    // localize the starting position
    ros::spinOnce();
    std::vector<float> starting_pos{robotPose.x, robotPose.y,
    robotPose.phi};
    origin = starting_pos;
    ROS_INFO("Finished obtaining starting pose: %f, %f, %f", robotPose.x,
    robotPose.y, robotPose.phi);
}

int Navigation::getCurrentBoxId() {
    // get nearest box as current
    std::vector<float> curr_pos = robotPose.toCoords();
    int box_id = -1;
    float curr_dist = 10000;

    for (int i = 0; i < boxes.coords.size(); i++) {
        float dist = getDist(curr_pos, boxes.coords[i]);
        if (dist < curr_dist) {
            box_id = i;
            curr_dist = dist;
        }
```

```
    }
    return box_id;
}

void Navigation::traverseAllBoxes() {
    // set up starting pose
    localizeStartingPose();

    // set up all view points first
    getViewPoints(boxes.coords);

    // determine box traversal order
    std::vector<std::vector<float>> traversal_nodes = boxes.coords;
    traversal_nodes.push_back(origin);
    std::vector<int> indices = getTraversalOrder(traversal_nodes,
    traversal_nodes.size()-1);

    // traverse every box and then return to starting point
    for (int i = 0; i < indices.size() - 1; i++) {
        traverseBox(indices[i]);
        // periodic log
        logImageIDs();
    }
    moveToGoal(origin[0], origin[1], origin[2]);
    logImageIDs();
}

void Navigation::traverseBox(int box_idx) {
    // traverse the given box from current position
    std::map<int,std::vector<std::vector<float>>>::iterator it =
    box_view_points.find(box_idx);
    if(it == box_view_points.end()) {
        ROS_INFO("Cannot find box with given index...");
    }
    else {
        // get current position and current image
        ros::spinOnce();
        std::vector<float> curr_pos = robotPose.toCoords();

        // determine forward or backward traversal order
        std::vector<std::vector<float>> view_points = it->second;
```

```
        int start_idx = 0;
        int end_idx = view_points.size();
        int step = 1;

        float dist_first = getDist(curr_pos, view_points[start_idx]);
        float dist_last = getDist(curr_pos, view_points[end_idx-1]);
        if (dist_first > dist_last) {
            start_idx = view_points.size()-1;
            end_idx = -1;
            step = -1;
        }

        ROS_INFO("Traversed box %d, %d", start_idx, end_idx);
        // start traversing
        int curr_idx = start_idx;
        while (curr_idx != end_idx) {
            std::vector<float> curr_goal = view_points[curr_idx];

            ROS_INFO("Moving to box %d point %d from %d to %d: %f, %f, %f", box_idx, curr_idx, start_idx, end_idx, curr_goal[0], curr_goal[1], curr_goal[2]);
            // move to veiw point
            moveToGoal(curr_goal[0], curr_goal[1], curr_goal[2]);
            ROS_INFO("Current pose: %f, %f, %f", robotPose.x, robotPose.y, robotPose.phi);
            // do image stuff
            ros::spinOnce();
            imagePipeline.updateLogits(boxes, box_idx);
            imagePipeline.finalizeTemplateID(box_idx);
            // next view point
            curr_idx += step;
        }
        ROS_INFO("Traversed box %d", box_idx);
    }
}


void Navigation::logImageIDs() {
    ofstream myfile;
    myfile.open("c2_output.txt");

```

```
    // load current image recognition progress
    for (int i = 0; i < boxes.coords.size(); i++) {
        int img_id = imagePipeline.box_to_ID(i);
        std::string img_name = imagePipeline.ID_to_name(img_id);
        float x = boxes.coords[i][0];
        float y = boxes.coords[i][1];
        ROS_INFO("Box %d is image %d (%s) at (%f, %f)", i, img_id, img_name.c_str(), x, y);

        // log current result to file
        myfile << img_name << "," << x << "," << y << "\n";
    }

    myfile.close();
}
```

**record**$_{c}am_{f}rame.cpp$

```cpp
#include "opencv2/opencv.hpp"
#include "iostream"

int main(int, char**) {
    // open the first webcam plugged in the computer
    cv::VideoCapture camera(0);
    if (!camera.isOpened()) {
        std::cerr << "ERROR: Could not open camera" << std::endl;
        return 1;
    }

    // create a window to display the images from the webcam
    cv::namedWindow("Webcam", CV_WINDOW_AUTOSIZE);

    // this will contain the image from the webcam
    cv::Mat frame;

    // capture the next frame from the webcam
    camera >> frame;

    // display the frame until you press a key
    while (1) {
        // show the image on the window
        cv::imshow("Webcam", frame);
        // wait (10ms) for a key to be pressed
        if (cv::waitKey(10) >= 0)
            break;
    }
    return 0;
}
```

**imagePipeline.cpp**

```
#include <imagePipeline.h>

#define IMAGE_TYPE sensor_msgs::image_encodings::BGR8
#define IMAGE_TOPIC "camera/rgb/image_raw" // kinect:"camera/rgb/image_raw" webcam:"camera/image"

using namespace cv;
using namespace std;
using namespace cv::xfeatures2d;

void ImagePipeline::updateLogits(Boxes &boxes, int boxID)
{
    int iMatch = getTemplateID(boxes) + 2; // note that getTemplateID returns -2 to 2
    for (int i = 0; i < NumStatus; i++)
    {
        int change = (i == iMatch) ? (alpha) : (beta);
        logits[boxID][i] += change;
    }
    std::cout << "Updated logits" << std::endl;
}

void ImagePipeline::finalizeTemplateID(int boxID)
{
    vector<float> currLogits = logits[boxID];

    // bestIndex = best of softmax or simply the max
    float maxLogit = -1000;
    int bestIndex = 0;
    for (int i = 0; i < currLogits.size(); i++) {
        if (currLogits[i] > maxLogit) {
            maxLogit = currLogits[i];
            bestIndex = i;
        }
    }
    std::cout << "Finalized Template ID " << boxID << " " << bestIndex << std::endl;
    setTemplateID(bestIndex - 2, boxID); // note that getTemplateID returns -2 to 2, index is however 0 to 4
```

```
}

void ImagePipeline::finalizeTemplateIDs()
{
    // finalize the templateIDs vector once and for all
    for (int box = 0; box < NumBoxes; box++)
    {
        finalizeTemplateID(box);
    }
}

int ImagePipeline::getTemplateID(Boxes &boxes)
{

    int templateID = AMBIGUITY;
    cv::Mat target_1 =
   imread("/home/yt1234gary/catkin_ws_mie/src/Capstone/mie443_contest2/boxes_database/temp
   IMREAD_GRAYSCALE);
    cv::Mat target_2 =
   imread("/home/yt1234gary/catkin_ws_mie/src/Capstone/mie443_contest2/boxes_database/temp
   IMREAD_GRAYSCALE);
    cv::Mat target_3 =
   imread("/home/yt1234gary/catkin_ws_mie/src/Capstone/mie443_contest2/boxes_database/temp
   IMREAD_GRAYSCALE);

    if (!isValid)
    {
        std::cout << "ERROR: INVALID IMAGE!" << std::endl;
    }
    else if (img.empty() || img.rows <= 0 || img.cols <= 0)
    {
        std::cout << "ERROR: VALID IMAGE, BUT STILL A PROBLEM EXISTS!" <<
   std::endl;
        std::cout << "img.empty():" << img.empty() << std::endl;
        std::cout << "img.rows:" << img.rows << std::endl;
        std::cout << "img.cols:" << img.cols << std::endl;
    }
    else
    {
        // Store rectangle areas of each img-target matching in an array
        std::vector<float> matchedAreas(NumTargets, 0.0);
```

```
    for (int i = 0; i < NumTargets; i++)
    {
        cv::Mat target = boxes.templates[i];
        matchedAreas[i] = performSURF(img, target);
        cout << "Target " << i << " | area = " << matchedAreas[i] << endl;
    }

    // Examine each matching area and decide which one (or none) to be chosen
    int candidateID = -1, candidateCount = 0, antiCandidateCount = 0;
    for (int i = 0; i < NumTargets && candidateCount < 2; i++)
    {
        float area = abs(matchedAreas[i]);
        bool isRectangle = (matchedAreas[i] > 0);
        bool isGoodSized = (area < MaxGoodArea && area > MinGoodArea);
        bool isOutofBound = (area > MinArea || area <= MinArea);

        if (isRectangle && isGoodSized)
        {
            candidateID = i;
            candidateCount++;
            cout << "\n--- Matching target " << i << isRectangle << isGoodSized << "---\n";
        }
        else
        {
            antiCandidateCount++;
            cout << "\n--- Not Matching target " << i << isRectangle << isGoodSized << "---\n";
        }
    }

    switch (candidateCount)
    {
    case 1: // one good match found
        templateID = candidateID;
        break;
    case 0: // other cases
    case 2:
    case 3:
```

```
            templateID = BLANK;
            break;
        default: // certain that all three are non-matches
            if (antiCandidateCount == 3)
                templateID = BLANK;
        }

        // // Use: boxes.templates
        // cv::imshow("view", img);
        // cv::waitKey(10);
    }
    cout << "\n--- Finished template id matching " << "---\n";
    return templateID;
}

void ImagePipeline::imageCallback(const sensor_msgs::ImageConstPtr &msg)
{
    try
    {
        if (isValid)
        {
            img.release();
        }
        img = (cv_bridge::toCvShare(msg, IMAGE_TYPE)->image).clone();
        isValid = true;
    }
    catch (cv_bridge::Exception &e)
    {
        std::cout << "ERROR: Could not convert from " <<
    msg->encoding.c_str()
                  << " to " << IMAGE_TYPE.c_str() << "!" << std::endl;
        isValid = false;
    }
}

float ImagePipeline::getArea(std::vector<cv::Point2f> scene_corners,
    cv::Mat img_object)
{
    if (scene_corners.size() < 4)
    {
        cout << " scene_corners size not correct, should be 4 " << endl;
```

```
        return 0.0;
    }

    vector<Point2f> points(4);
    for (int i = 0; i < scene_corners.size(); i++){
        points[i] = scene_corners[i] + Point2f( img_object.cols, 0);
    }

    // Get corner points for rectangle
    auto x_min = fmin(fmin(points[0].x, points[1].x), fmin(points[2].x,
    points[3].x));
    auto y_min = fmin(fmin(points[0].y, points[1].y), fmin(points[2].y,
    points[3].y));
    auto x_max = fmax(fmax(points[1].x, points[1].x), fmax(points[2].x,
    points[3].x));
    auto y_max = fmax(fmax(points[1].y, points[1].y), fmax(points[2].y,
    points[3].y));

    float length = abs(x_max - x_min);
    float width = abs(y_max - y_min);
    float area = length * width;

    // check if area is a valid rectangle, return engative value of the
    area if not valid
    if ((abs(points[0].x - points[3].x) < 50) && (abs(points[1].x -
    points[2].x) < 50) && (abs(points[0].y - points[1].y) < 50) &&
    (abs(points[2].y - points[3].y) < 50))
        return area;
    else
        return -1 * area;
}

float ImagePipeline::performSURF(cv::Mat img_scene, cv::Mat img_object)
{
    //-- Step 1 & 2: Detect the keypoints and calculate descriptors using
    SURF Detector
    int minHessian = 400;
    Ptr<SURF> detector = SURF::create(minHessian);
    std::vector<KeyPoint> keypoints_object, keypoints_scene;
    Mat descriptors_object, descriptors_scene;
    detector->detectAndCompute(img_object, Mat(), keypoints_object,
```

```
                                  descriptors_object);
    detector->detectAndCompute(img_scene, Mat(), keypoints_scene,
                                  descriptors_scene);

    //-- Step 3: Matching descriptor vectors using FLANN matcher
    FlannBasedMatcher matcher;
    std::vector<DMatch> matches;
    matcher.match(descriptors_object, descriptors_scene, matches);

    double max_dist = 0;
    double min_dist = 100;

    //-- Quick calculation of max and min distances between keypoints
    for (int i = 0; i < descriptors_object.rows; i++)
    {
        double dist = matches[i].distance;
        if (dist < min_dist)
            min_dist = dist;
        if (dist > max_dist)
            max_dist = dist;
    }

    //-- Draw only "good" matches (i.e. whose distance is less than 3*min_dist )
    std::vector<DMatch> good_matches;

    for (int i = 0; i < descriptors_object.rows; i++)
    {
        if (matches[i].distance < 3 * min_dist)
        {
            good_matches.push_back(matches[i]);
        }
    }

    Mat img_matches;
    drawMatches(img_object, keypoints_object, img_scene, keypoints_scene,
                good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
                vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

    //-- Localize the object
```

```
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;
    for (int i = 0; i < good_matches.size(); i++)
    {
        //-- Get the keypoints from the good matches
        obj.push_back(keypoints_object[good_matches[i].queryIdx].pt);
        scene.push_back(keypoints_scene[good_matches[i].trainIdx].pt);
    }

    if (obj.size() < 4 || scene.size() < 4)
    {
        return 0;
    }
    Mat H = findHomography(obj, scene, RANSAC);

    //-- Get the corners from the image_1 ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0, 0);
    obj_corners[1] = cvPoint(img_object.cols, 0);
    obj_corners[2] = cvPoint(img_object.cols, img_object.rows);
    obj_corners[3] = cvPoint(0, img_object.rows);
    std::vector<Point2f> scene_corners(4);

    cout << "OBJ: " << obj.size() << endl;
    cout << "SCENE: " << scene.size() << endl;

    if (H.empty())
        return 0;

    perspectiveTransform(obj_corners, scene_corners, H);

    //-- Draw lines between the corners (the mapped object in the scene -
   image_2 )
    line(img_matches, scene_corners[0] + Point2f(img_object.cols, 0),
   scene_corners[1] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
    line(img_matches, scene_corners[1] + Point2f(img_object.cols, 0),
   scene_corners[2] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
    line(img_matches, scene_corners[2] + Point2f(img_object.cols, 0),
   scene_corners[3] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
    line(img_matches, scene_corners[3] + Point2f(img_object.cols, 0),
   scene_corners[0] + Point2f(img_object.cols, 0), Scalar(0, 255, 0), 4);
```

```

    float area = getArea(scene_corners, img_object);
    cout << "AREA: " << area << endl;

    //-- Show detected matches
    // imshow("Good Matches & Object detection", img_matches);

    // waitKey(5000);
    return area;
}
```

### contest2.cpp

```
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>

// Matching status
#define RAISIN 0
#define CINNAMON 1
#define RICE 2
#define AMBIGUITY -1
#define BLANK -2

int main(int argc, char **argv)
{
    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;

    // Initialize box coordinates and templates
    Boxes boxes;
    if (!boxes.load_coords() || !boxes.load_templates())
    {
        std::cout << "ERROR: could not load coords or templates" <<
    std::endl;
        return -1;
    }
    for (int i = 0; i < boxes.coords.size(); ++i)
    {
        std::cout << "Box coordinates: " << std::endl;
        std::cout << i << " x: " << boxes.coords[i][0] << " y: " <<
    boxes.coords[i][1] << " z: "
                  << boxes.coords[i][2] << std::endl;
    }

    // nav contains subscriber to map_server, amcl and imagePipeline
    std::cout << "Loading navigation: " << std::endl;
    Navigation nav(n, boxes, 3);

    // Execute strategy.
    // while (ros::ok())
```

```
    // {
        std::cout << "Beginning navigation: " << std::endl;
        nav.traverseAllBoxes();
    // }
    return 0;
}
```

### boxes.cpp

```
#include <ros/package.h>
#include <boxes.h>

bool Boxes::load_coords() {
    std::string filePath = ros::package::getPath("mie443_contest2") +

    std::string("/boxes_database/gazebo_coords.xml");
    cv::FileStorage fs(filePath, cv::FileStorage::READ);
    if(fs.isOpened()) {
        cv::FileNode node;
        cv::FileNodeIterator it, end;
        std::vector<float> coordVec;
        std::string coords_xml[5] = {"coordinate1", "coordinate2",
    "coordinate3", "coordinate4",
                                  "coordinate5"};
        for(int i = 0; i < 5; ++i) {
            node = fs[coords_xml[i]];
            if(node.type() != cv::FileNode::SEQ) {
                std::cout << "XML ERROR: Data in " << coords_xml[i]
                          << " is improperly formatted - check input.xml"
    << std::endl;
            } else {
                it = node.begin();
                end = node.end();
                coordVec = std::vector<float>();
                for(int j = 0; it != end; ++it, ++j) {
                    coordVec.push_back((float)*it);
                }
                if(coordVec.size() == 3) {
                    coords.push_back(coordVec);
                } else {
                    std::cout << "XML ERROR: Data in " << coords_xml[i]
                              << " is improperly formatted - check
    input.xml" << std::endl;
                }
            }
        }
        if(coords.size() == 0) {
            std::cout << "XML ERROR: Coordinate data is improperly
```

```
formatted - check input.xml"
                    << std::endl;
            return false;
        }
    } else {
        std::cout << "Could not open XML - check FilePath in " << filePath << std::endl;
        return false;
    }
    return true;
}

bool Boxes::load_templates() {
    std::string filePath = ros::package::getPath("mie443_contest2") +
                           std::string("/boxes_database/templates.xml");
    cv::FileStorage fs(filePath, cv::FileStorage::READ);
    if(fs.isOpened()) {
        cv::FileNode node = fs["templates"];;
        cv::FileNodeIterator it, end;
        if(!(node.type() == cv::FileNode::SEQ || node.type() == cv::FileNode::STRING)) {
            std::cout << "XML ERROR: Image data is improperly formatted in " << filePath
                      << std::endl;
            return false;
        }
        it = node.begin();
        end = node.end();
        std::string imagepath;
        for(; it != end; ++it){
            imagepath = ros::package::getPath("mie443_contest2") +
                        std::string("/boxes_database/") +
                        std::string(*it);
            templates.push_back(cv::imread(imagepath, CV_LOAD_IMAGE_GRAYSCALE));
        }
    } else {
        std::cout << "XML ERROR: Could not open " << filePath << std::endl;
        return false;
    }
```

```
    return true;
}
```

**robot**$_{pose.cpp}$

```
#include <robot_pose.h>
#include <tf/transform_datatypes.h>

RobotPose::RobotPose(float x, float y, float phi) {
  this->x = x;
  this->y = y;
  this->phi = phi;
}

void RobotPose::poseCallback(const
    geometry_msgs::PoseWithCovarianceStamped& msg) {
  phi = tf::getYaw(msg.pose.pose.orientation);
  x = msg.pose.pose.position.x;
  y = msg.pose.pose.position.y;
}
```

**webcam**$_p$*ublisher.cpp*

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sstream> // for converting the command line parameter to integer

int main(int argc, char** argv)
{
  // Check if video source has been passed as a parameter
  if(argv[1] == NULL){
    std::cout << "****No Camera Selected****" << std::endl << "Input camera number to use, ie. 0 for default laptop camera." << std::endl;
    return 1;
  }

  ros::init(argc, argv, "image_publisher");
  ros::NodeHandle nh;
  image_transport::ImageTransport it(nh);
  image_transport::Publisher pub = it.advertise("camera/image", 1);

  // Convert the passed as command line parameter index for the video device to an integer
  std::istringstream video_sourceCmd(argv[1]);
  int video_source;
  // Check if it is indeed a number
  if(!(video_sourceCmd >> video_source)) return 1;

  cv::VideoCapture cap(video_source);
  // Check if video device can be opened with the given index
  if(!cap.isOpened()) return 1;
  cv::Mat frame;
  sensor_msgs::ImagePtr msg;

  ros::Rate loop_rate(30);
  while (nh.ok()) {
    cap >> frame;
    // Check if grabbed frame is actually full with some content
    if(!frame.empty()) {
      msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8",
```

```
    frame).toImageMsg();
      pub.publish(msg);
      cv::waitKey(1);
    }

    ros::spinOnce();
    loop_rate.sleep();
  }
}
```