Automatic, Careful Online Packing of Groceries Using a Soft Robotic Manipulator and Multimodal Sensing

by

Jeana Choi

S.B. Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . .
Department of Electrical Engineering and Computer Science
January 14, 2022

Certified by . . . .
Daniela Rus
CSAIL Director, Andrew (1956) and Erna Viterbi Professor of Electrical Engineering and Computer Science, Deputy Dean of Research, Schwarzman College of Computing
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Automatic, Careful Online Packing of Groceries Using a Soft Robotic Manipulator and Multimodal Sensing

by

Jeana Choi

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the use of soft robotic manipulators with multimodal sensing for estimating the physical properties of unknown objects to enable sorting and packing. Although bin packing has been a key benchmark task for robotic manipulation, the community has mainly focused on the placement of rigid rectilinear objects within the container. We address this by presenting a soft robotic hand that uses a combination of vision, motor-based proprioception and soft tactile sensors to identify and pack a stream of unknown objects. We translate the ill-defined human conception of a "well-packed container" into metrics that match combinations of our different sensor modalities and demonstrate how this works in a grocery packing scenario, where objects of arbitrary shape, size and stiffness come down a conveyor belt. The proposed multimodal approach is supported by physical experiments demonstrating how the integration of multiple sensing modalities can address complex manipulation applications.

Thesis Supervisor: Daniela Rus
Title: CSAIL Director, Andrew (1956) and Erna Viterbi Professor of Electrical Engineering and Computer Science, Deputy Dean of Research, Schwarzman College of Computing

Acknowledgments

I would like to give a heartful thanks to everyone in Distributed Robotics Laboratory for their support and kindness.

First, I would like to thank my MEng advisor, Professor Daniela Rus, for helping me grow as a researcher and also providing valuable feedback about the grocery packing project. There is so much to learn from her passion and drive for robotics and research, and I am very grateful for the opportunity to learn the DRL ways of thinking in the past year.

Next, Lilly Chin, my graduate student mentor, welcomed me with (socially distanced) open arms in Fall 2020, and graciously walked me through the onboarding process and introduced me to the UR-5 robot. She has my sincerest gratitude for her patience and sharing feedback especially in the 2020-2021 academic year, when we were one of the few people in lab due to Covid restrictions on lab space occupancy.

Valerie Chen introduced me to the project, and also essentially became my soulmate in lab - spending over 40 hours a week together in Fall 2021. I would like to thank her especially for the amount of motivation and new ideas she comes up with in research, which inspires me to do the same.

In addition, I would like to specifically thank: Mieke Moran for always checking up on me and giving me delicious snacks, Jim Bern and Cenk Baykal for lending a listening ear and providing feedback, John Romanishin for his speedy help during hardware crises, Xiao Li and Yutong Ban for the too-frequent Cava dinner runs, Brandon Araki for all the support with the early-stage robot demos in Spring and Summer of 2021, and Annan Zhang and Peter Werner for their endless supply of late-night coffee.

Lastly, I would like to express my biggest thanks to my parents and my sister, who have always supported and encouraged me to try my best regardless of the situation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Soft robotic grasping offers a potentially robust solution to the bin packing problem. Picking items from clutter and placing them into ordered bins has been an important benchmark for the broader robotic manipulation community, as exemplified by the Amazon Picking and Robotics Challenge [34]. However, current solutions have only focused on vision-based segmentation for rigid grippers, meaning that these systems only can grasp rigid rectilinear objects with significant pre-computation [11, 16, 25, 29, 37]. Rigid systems often shies away from extended environmental interaction to avoid visual occlusion or damaging grasped objects, which has meant limited focus on the "placing" part of the "pick and place" task. Indeed, current solvers may reach intractable run times when instructed to pack as few as six objects [38]. Soft grippers can avoid many of these pre-computation issues, as their compliance makes them robust to changes in objects' stiffness, shapes and placement. A gripper can use its softness to grasp objects of arbitrary material properties without the models or precise location information that its rigid counterparts would require [8, 24, 32].

In this thesis, our vision is to utilize the qualities of a soft gripper to create an automated end-to-end system that can safely grasp unknown items and pack them carefully in the bin. In the long term, this system could extend to deploy in the real world for specific applications in bin packing.

However, sensorizing these flexible grippers remains challenging, especially when multiple sensor modalities are needed. A soft gripper's deformability makes it difficult to accurately place tactile sensors and localize forces spatially along the gripper [31, 39]. In online applications where the material properties of objects and the order they arrive are unknown – such as in bagging groceries, loading a dishwasher, or packing for a move – it becomes critical to combine the global scale of vision with the localized scale of tactile sensors. These different sensing modalities complement one another to ensure an accurate understanding of an object's material properties in a timely manner [13, 40]. While there has been significant focus in gaining accurate proprioception of soft grippers, either through using vision *for* tactile sensing [1, 20] or incorporating rigid sensing elements within soft systems [14, 28], there is still currently relatively little overlap between soft robotics with contemporary sensor fusion techniques for complex online applications.

We address this gap by creating an end-to-end online bin packing system that can automatically and carefully pack an unknown stream of groceries with a soft gripper by using multiple sensing modalities to understand grocery item size, shape, and stiffness. Specifically, we present a system that utilizes an online algorithm for bin packing with constraints as objects arrive on a conveyor belt. We combine RGB-Depth cameras with pressure-transduction based sensors to provide the sensory feedback needed to make appropriate packing decisions for our robot arm-mounted soft gripper. We incorporate proprioceptive feedback from the rigid servo motors that drive our soft gripper with the soft tactile sensors and external vision systems.

We demonstrate the power of this soft robotic system by comparing its performance against sensorless and vision-only systems in a grocery packing scenario. Grocery packing presents a strong case study as groceries range widely in size, shape, weight and fragility. Bagging groceries well requires combining a mechanical system that can safely manipulate objects safely with algorithms that ensure groceries on the bottom of the bin are not crushed by groceries above it. Our system combines the robust safe handling of soft grippers with the richness of a multimodal sensor suite to outperform traditional vision-only based approaches in this complex task. Although the objects range in size and fragility — from heavy boxes to delicate produce — the robot is able to detect the shape and stiffness on an object in real time and determine a placement sequence that does not cause any object to be crushed by the weight of the objects placed on top of it. In addition, we test long runs of the grocery packing system to evaluate its robustness against time.

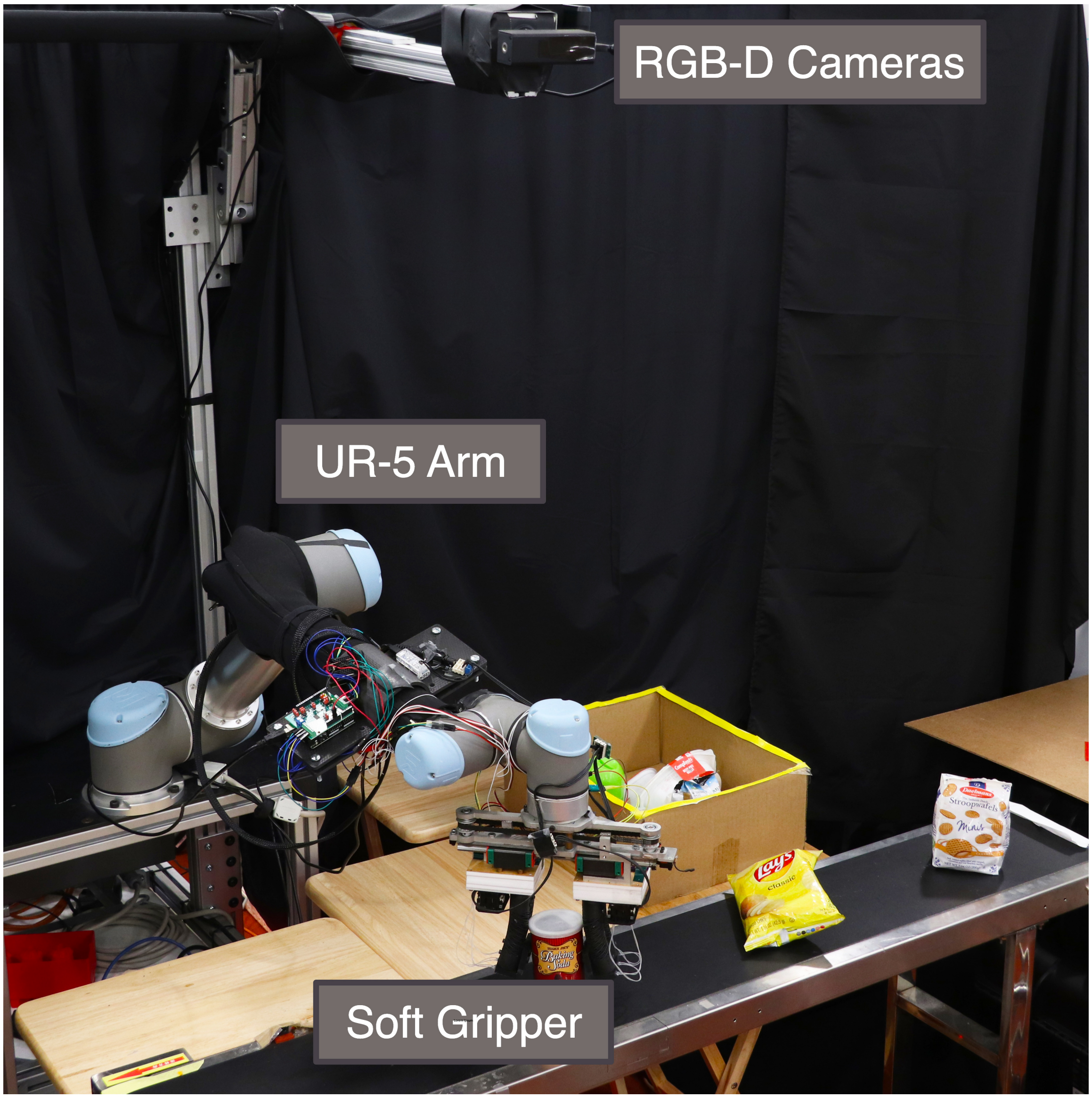


Figure 1-1: The setup for an end-to-end online bin packing system. Multimodal sensing is utilized to achieve grasping previously unknown items from the conveyor belt and making real-time decisions about the packing location based on the item's physical properties of size, shape, and stiffness.

In summary, we make the following contributions:

1. An automated end-to-end system that can continuously pick and pack grocery items from a conveyor belt safely into the bin.

2. An integrated physical soft grasping platform that merges vision, motor-based proprioception and pressure-based tactile sensing in a soft grasping system.

3. An online packing algorithm that takes in multiple sensor inputs to create a "well-packed" container that matches human expectations.

4. Physical experiments with our multimodal approach and comparisons against traditional blind and vision packing methods in a realistic grocery packing scenario with irregular objects.

1.1 Thesis Organization

This thesis is based on the following academic papers co-written by the thesis author.

1. (in submission) J. Choi*, V. K. Chen*, L. Chin, and D. Rus "Soft Robotic Manipulation with Multimodal Sensing For Online Packing of Groceries", in *IEEE International Conference on Soft Robotics (RoboSoft)*, 2022 - basis for Ch. 3-7

Chapter 2 is an overview of soft robots in manipulation, and multimodal sensing in this field. Chapter 3 overviews and details the hardware architecture, including the robot, gripper, and sensors. Chapter 4 highlights and explains the algorithms used to evaluate the vision, tactile, and proprioception data stream. Chapter 5 explains the overall grocery packing algorithm along with its automation and limitations. Chapters 6 and 7 delve into the evaluation method and results of the performed experiments. Lastly, Chapter 8 summarizes the research done during the Master of Engineering program, and lists potential directions this modular multimodal system may take in future work.

Chapter 2

Related Work

Robot research on packing has focused on minimizing unoccupied volume or runtime for a given number of rigid objects [16, 33]. These works often rely on knowing the packed objects' and bin's geometry beforehand, with many requiring significant off board pre-processing [2, 37, 41]. For online applications, where objects are not known beforehand and may be deformable or fragile, these methods are insufficient.

A large amount of research has been conducted on manipulation-based tasks with single modal perception, such as only using vision to generate dense three-dimensional maps of the environment [30] or using haptic feedback through an underactuated soft hand to classify objects [15]. While utilizing one modality for manipulation may be less complicated, it is difficult to achieve an end-to-end system of grasping unknown objects, classifying their delicacy, and packing them online and safely in bins.

Sensor fusion may provide an effective way to achieve online packing. In particular, visual and tactile sensors provide complementary ranges of data, focusing on global and localized scales respectively. When combined, these different modalities can enable detection of corrupted data from a sensor and deeper understanding of an object's tactile properties [13, 40]. Widely studied for mobile robot applications (eg. navigation, state estimation and localization), sensor fusion has been shown to improve grasp reliability [4, 6, 21, 40], task accuracy [12, 42], and scene/object understanding for robot manipulation tasks [5] as well as facilitate human-robot handovers [19]. These research discuss grasping strategies using the sensor feedback data and the importance of using both to extract different characteristics from the surrounding environment. F. Sun discusses the benefits of using visual sensing to determine color and shape and coupling that with haptic feedback to receive information about the object's softness, stiffness, and surface texture.

Different modalities may also be combined to perform end-to-end precision tasks such as wire-insertion [12], peg-insertion [23], and vegetable-cutting [40].

With few exceptions [22], the majority of previous work on sensor fusion for robot manipulation relies upon machine learning for at least one portion of the pipeline [4, 6, 23]. In M. A. Lee et al., the robot requires self-supervised learning utilizing tactile sensors and a camera to determine the optimal method for inserting its peg-shaped end effector to the matching hole. While powerful for learning various grasping policies, these methods require detailed prior knowledge of the items, such as pre-designed CAD models or built-in learned representations, which again makes these methods ill-suited for online applications.

The context includes the weight and delicacy of the current object relative to what is already in the bin and what is yet to come on the conveyor belt. For example, the robot should not place potatoes on top of lettuce. Prior work on bin packing focused on the size and shape of known rigid objects to minimize wasted space in the bin [33,37,38]. However, geometric properties alone do not capture the full constraints required for packing without crushing.

Although the sensorization of soft grippers is an active area of research, there has been relatively little overlap with contemporary sensor fusion techniques. One major challenge for soft sensorization is the soft gripper's deformability, making it difficult to accurately place tactile sensors and localize forces spatially along the gripper. Significant focus has thus been placed on getting accurate proprioception as an intermediate step before more integrated sensor fusion [39]. The most popular combination of sensor modalities is in the use of vision *for* tactile sensing, where the high resolution of a camera or time-of-flight sensor is used to track the deformation of a soft surface to get tactile information [1,20]. Others incorporate rigid elements to provide proprioception within their soft structure, occasionally supplementing this with further tactile sensors [14, 28]. We build on this approach and our previous work by choosing a strategy where we create an end-to-end grocery packing system, using multimodal sensing to achieve manipulation of unknown objects with soft fingers and packing them safely in the grocery bin. We incorporate proprioceptive feedback from the rigid servo motors that drive our soft gripper with the soft tactile sensors and external vision systems to provide our multimodal approach [9].

Chapter 3

Hardware Architecture

Our soft end-to-end robotic packing system has four major components: (1) a soft multiplexed manipulator from [7] which provides proprioceptive feedback through its servomotors, (2) pressure-based tactile sensors attached to the fingers of this gripper, (3) two external RGB-D cameras to provide visual information about objects to be grasped and the packing area, (4) the algorithm that integrates these systems together to perform online packing.

These components are integrated together on a UR5 robot arm to pick unknown objects off a conveyor belt and pack them into a bin either immediately or after being set aside to pack other items first. The RGB-D camera detects the object's location and provides an estimate of the object's size. The tactile sensors provide additional information about the object's estimated stiffness. All of these properties are determined in real time without significant pre-computation, enabling true online packing.

3.1 Robot

The UR-5 robot arm was used for this system, where it has 6 joints and a maximum reach of 33 inches. The gripper module is attached to the robot's end effector joint, and its cables run down the arm of the robot. Each of the robot's six joints have a maximum velocity of 0.1572 radians per second, and maximum joint acceleration limit of 0.0349 $\frac{\text{rad}}{\text{s}^2}$.

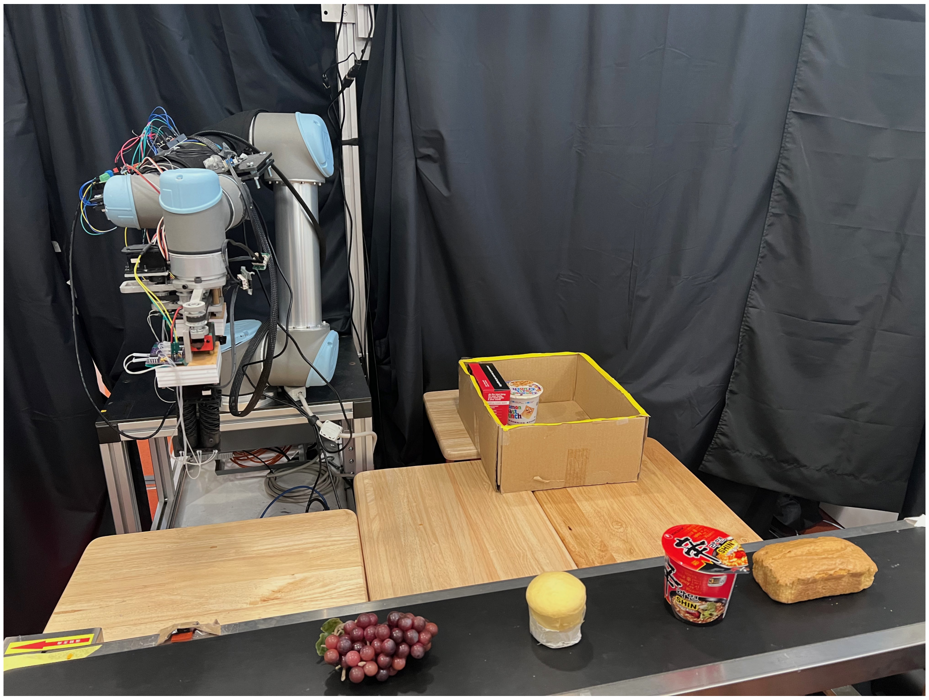


Figure 3-1: UR5 robot with the grocery packing setup. The grocery items, buffer zone, and packing bin can also be seen from this view.

3.1.1 Safety Considerations

There were several safety measures taken to reduce potential harm to both the operator and the robot with gripper. First, the joints were configured to limit the robot from colliding with the gripper. Our setup was fixed such that the gripper would remain perpendicular to the ground by fixing the robot's end effector joint to only rotate about the x and y axis. Next, there were several restrictions added to the planning space to optimize for a valid path and to provide safety to the robot operator. As a result, there were collision surfaces added in the software to block off irrelevant search space, and a horizontal barrier indicating where the conveyor belt was located. This was so that the robot arm would not try to reach through the conveyor belt and potentially damage both the gripper and the belt setup.

3.2 Proprioceptive Gripper

We use a soft gripper previously introduced in [7]. Briefly, the actuators of our gripper are constructed from handed shearing auxetics (HSAs), which are electrically-driven by servo

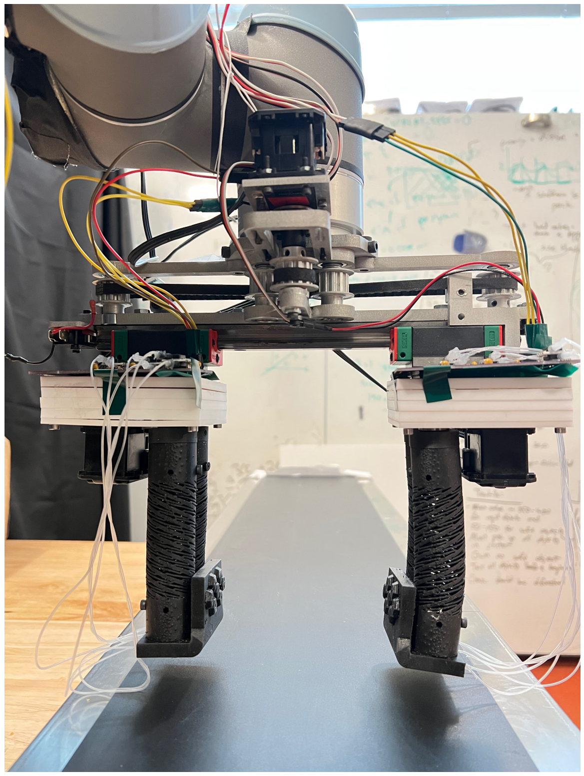


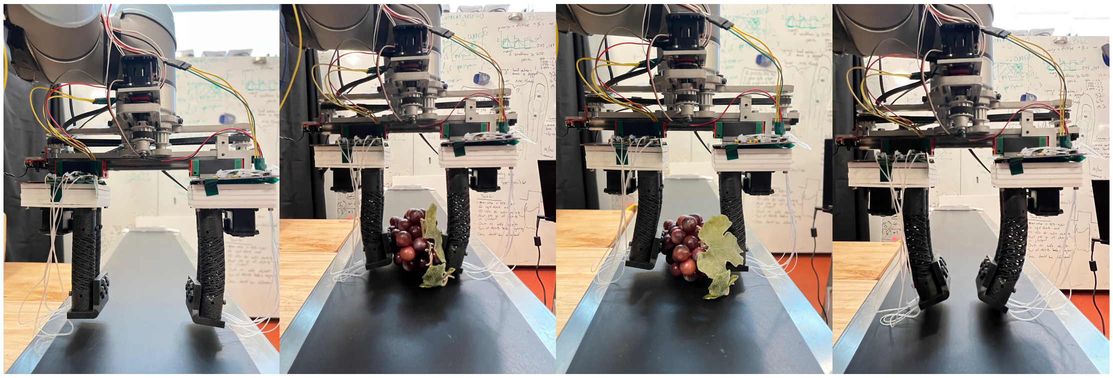
Figure 3-2: Soft Gripper consisting of HSA fingers and driven by Dynamixel Servos.

motors. We 3D print the HSAs via digital project lithography, as we reported previously, from a proprietary photopolymer resin (FPU 50, Carbon, Inc.) [35].

This gripper offers multiplexed manipulation, enabling us to grasp objects using parallel plate grasps, soft finger grasps or a combination of the two. The gripper module is 23 cm in length and 16.5 cm in height, with a maximum item cross-section clearance of 11 cm.

3.2.1 Grasping

For this work, we upgraded the servo motors from HiTec HS-5585MH to Dynamixel MX-28T servos. This turns the motor control system from open-loop to closed feedback, as these new servos provide feedback on their position, speed, and detected load. This enables new features such as dynamic grasps, object grasp detection and potential size estimation. In this thesis, we use proprioception to dynamically grasp items based on measured load, enabling safe handing of potentially delicate objects. Additionally, we leverage estimates of object size from vision and position feedback to adjust the amount the gripper opens to



(a) Full Open (b) Full Close (c) Partially Open (d) Auxetic Close

Figure 3-3: A visual representation of the gripper's different grasping modes. (a) The gripper is opened to its physical hardware limit, (b) The fingers are closed just enough for sufficient contact with the item, (c) the gripper opened partially but not fully, generally used to drop items carefully into the bin, (d) A demonstration of the auxetic fingers closing to ensure better contact with the item.

pack objects, allowing the system to pack items into narrower gaps.

More specifically, there were three Dynamixel Servos used for the system - two to control each finger, and one for the open/close functionality of the gripper, which we will refer to as the track servo. Position control was used to control each finger, which determined the amount of twisting of the HSAs to provide a better grasp on the item. The track servo used modified velocity-based control, where the gripper would close with a constant velocity until a sharp spike in the measured load value was reached, indicating contact with the current item the gripper is attempting to grasp. At this point, the goal velocity would be set to zero, stopping the gripper from closing too far and potentially damaging both itself and the item. Since the setup only allowed for one synchronous read or write at a time, a state machine needed to be defined that could interleave between reading the position, velocity, and load values, and writing commands to set the velocities and positions of the Dynamixel Servos. There are several actions a gripper can take for opening and closing, combining a mix of the three servos:

(a) *Open Fully:* The track servo is commanded with a negative velocity of -28.6 revolutions per minute, until the open position limit is reached. This gripper action is used to either place an item or reset the gripper to its resting state.

(b) *Close Fully:* The track servo is commanded with a positive velocity of 45.8 revolutions per minute, until either the load threshold or the maximum close position is reached. This gripper action is generally used to grasp an item.

(c) *Open Partially:* The gripper is commanded a negative velocity of -28.6 revolutions per minute until the track servo reads zero in load value, indicating that there is no more contact with the item. Due to the thickness of the fingers, this functionality remains vital such that the gripper does not open too wide and bump the grocery bin.

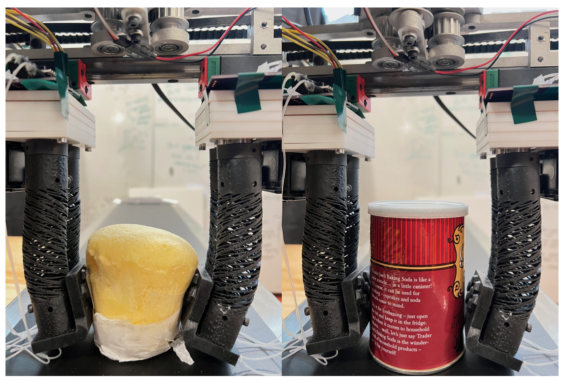
(d) *Finger Close:* The two servos that actuate the fingers are activated with position control, twisting the HSAs such that the gripper curves inward for better contact with the item it is grasping.

3.3 Tactile Sensors

In previous work, we have integrated tactile sensing capabilities in soft robotic grippers through soft capacitive [9] and resistive sensors [36]. Inspired by promising opportunities with fluidic tactile sensing [14, 18], we use the same 3D printer used to fabricate the HSAs to rapidly manufacture arrays of fluidic sensors. The fluidic sensor arrays consist of hollow, thin-walled hexagonal prisms in a semi close packed configuration. These features rest on a thick elastomeric panel, through which empty fluidic channels run from the inner cavity of each sensor to an edge of the panel. The entire sensor assembly is printed from a proprietary elastomeric polyurethane resin (EPU 40, Carbon, Inc.). Excess resin is removed from the printed part by aspirating with vacuum to create open channels. After the resin removal hole is sealed with Gorilla Super Glue Gel, silicone tubing is used to connect the closed volumes to differential pressure transducers (HSCDRRN160MDAA5, Honeywell).

As is common with tactile sensing approaches, lack of sufficient contact area with the target object resulted in uncharacteristically low sensor readings. However, due to the compliant nature of our soft gripper, this case of insufficient contact area resulted only for rigid target items. Softer target items allowed for reliable contact area due to the dual compliance of both gripper and target, while rigid target items forced the compliant gripper

Figure 3-4: Hardware setup with digitally-fabricated HSA fingersand digitally-fabricated hexagonal tactile sensors.



(a) Grasping soft item (b) Grasping rigid item

Figure 3-5: Grasping a soft item allows for better contact area, whereas rigid items have the phenomenon of lower contact area due to the soft nature of the gripper.

to bend away from rigid edges. We leverage this phenomenon due to the geometry of our flexible gripper to better separate rigid and soft objects, setting a lower tactile threshold for rigid items (Figure 5-2).

3.4 External Cameras

The external vision system uses two RGB-D cameras: an ASUS Xtion 2 to detect the locations and sizes of objects on the conveyor belt and an ASUS Xtion Pro Live to determine the best packing location in the bin. The Xtion 2 is placed 1.2 meters above the conveyor belt setup, and the Xtion Pro Live is secured 1.2 meters above the packing box. The cables run down the 80-20 beam in the center of the setup, and connect by USB to the laptop. As these are stereo cameras and automatically adjust the exposure and brightness, we manually set the depth registration on and turn off the auto-exposure for color segmentation, later explained in Chapter 4.

Chapter 4

Software Architecture

4.1 Robot Operating System

For this system, Python 2.7 and Robotic Operating System (ROS) Kinetic were used [26]. There is a ROS node for each external sensor, including the two RGB-depth cameras, Dynamixel Servos controlling the gripper, and for the custom tactile sensor. In addition, there are a series of nodes for the motion planning, node for both object detection and online packing, node for coordinate transforms, and a node for the grocery packing logic. 4-1 shows the complete data pipeline necessary for all moving parts to run smoothly during an active session of the grocery packing system.

4.1.1 Motion Planning

MoveIt! [10] was used as the motion planner of the UR-5 arm. The exact search algorithm was RRT-Connect, which provides a path from bidirectionally running RRT from the start and goal state, and a segment length of 0.005m was used. A maximum of 10 seconds is given for planning time, and the maximum velocity and acceleration scaling factors are .3 and .15, respectively.

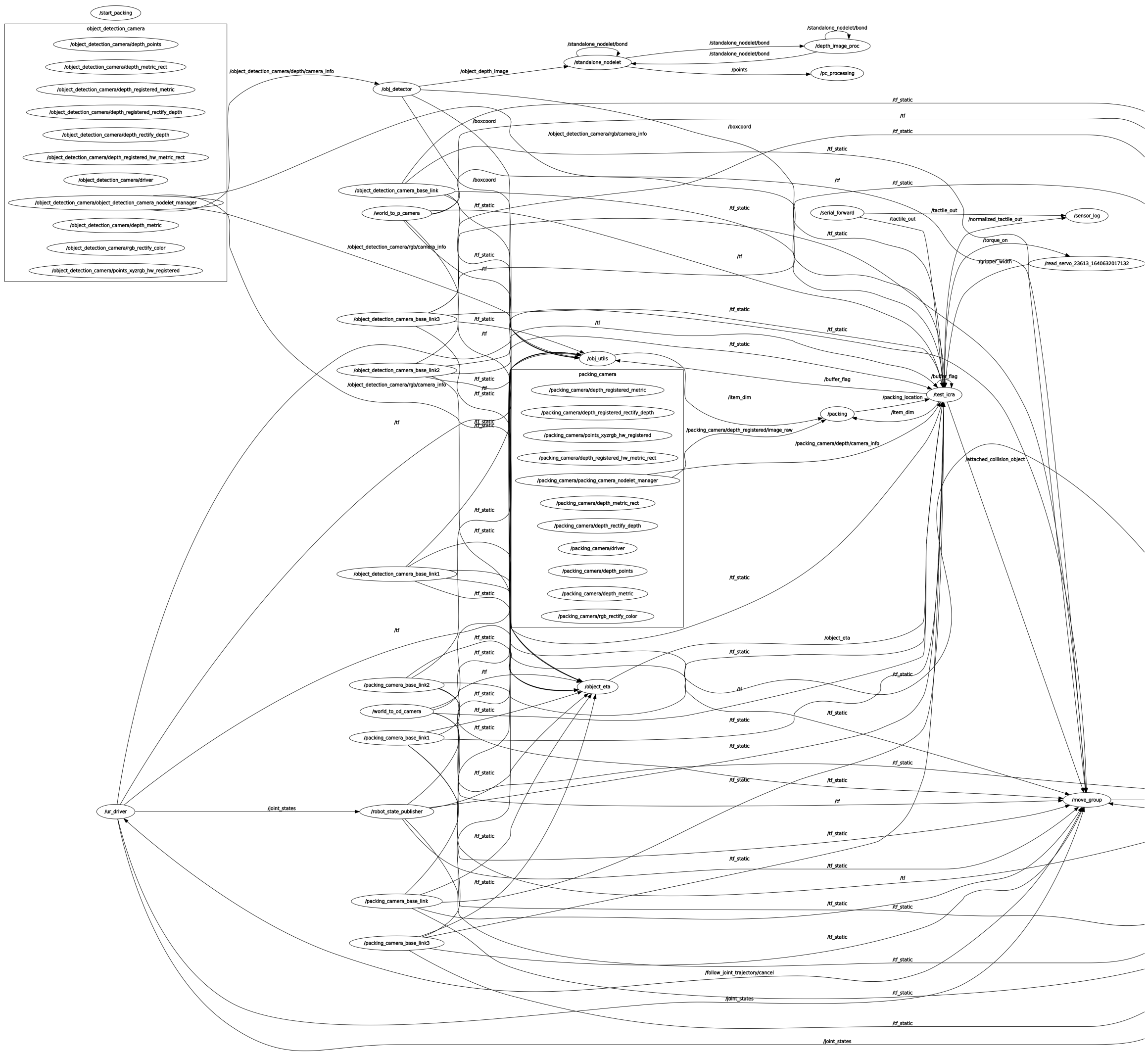


Figure 4-1: The RQT Tree graph for the grocery packing system. All nodes are shown, from the UR5 robot services to the external sensors and grocery packing software module.

4.2 External Vision System

The external vision system uses two RGB-D cameras, as described in Chapter 3. Algorithms for object detection and online packing were defined and implemented in separate ROS nodes, and run synchronously while the grocery packing system is active. In order to integrate the two cameras to the software architecture, coordinate transforms were required to translate the incoming camera stream into real world points understandable by the UR5 robot's motion planning system.

4.2.1 Coordinate Transform

Since we are using RGB-Depth cameras, it is possible to estimate the 3D pose of an item given its 2D coordinate pixels and depth from the camera.

$$w \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \begin{bmatrix} R \\ t \end{bmatrix} K$$

In the equation above, w is a scalar factor, x and y are the 2D pixel coordinates, X, Y, and Z make up the desired 3D point in the world coordinate frame, and R, t represent the extrinsic camera matrix, K being the intrinsic camera matrix.

For object detection and packing, the robot needs to know what 3d coordinate to move to in its coordinate frame. Therefore, there needs to be a coordinate transform from each of the cameras to the robot frame. This can be measured once because the cameras stay at a fixed distance from the base of the robot.

4.2.2 Object Detection

Objects to be packed are brought to the UR-5 via conveyor belt. Since the conveyor belt has a uniform black color and consistent location, color segmentation can be used via OpenCV to threshold out the belt and locate any number of grocery items.

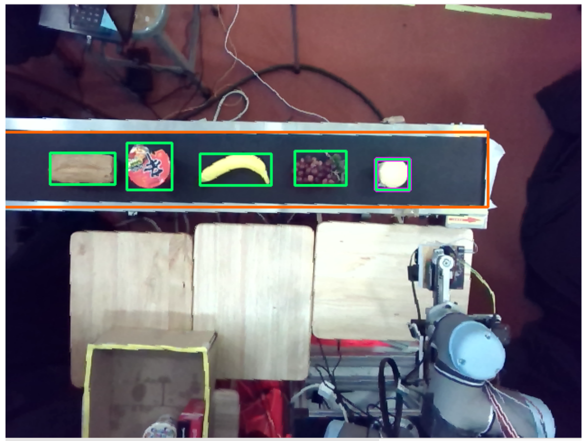


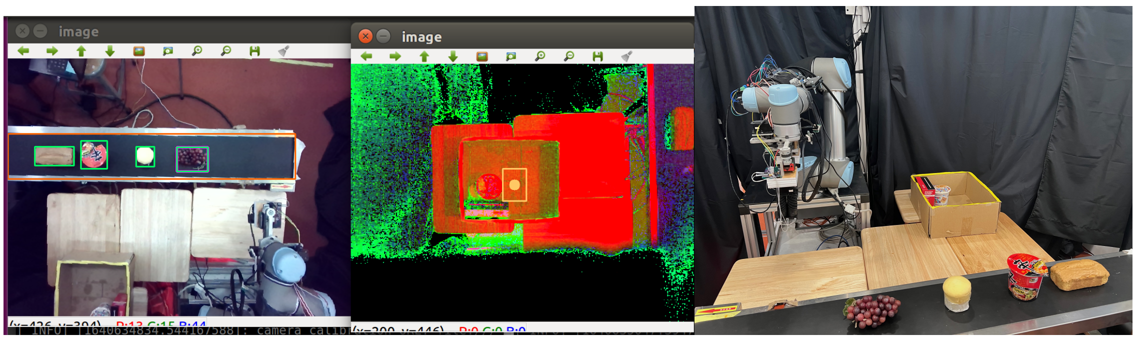
Figure 4-2: Snapshot from the Xtion2, which detects the objects on the conveyor belt. The green rectangles indicate bounding boxes around the detected grocery items, and the green rectangle with a purple outline indicates the item closest to the end of the conveyor belt.

Once the items have been segmented, we fit a bounding box to each item's contours to estimate its size. From all the items detected on the belt, we focus on the item closest to the edge of the conveyor belt. This can be calculated with a simple x coordinate comparison, since we know that the conveyor belt moves with one degree of freedom in the x axis with respect to the pixels of the Xtion2 camera. Therefore, the item with the largest x value is the item the system focuses on for grasping.

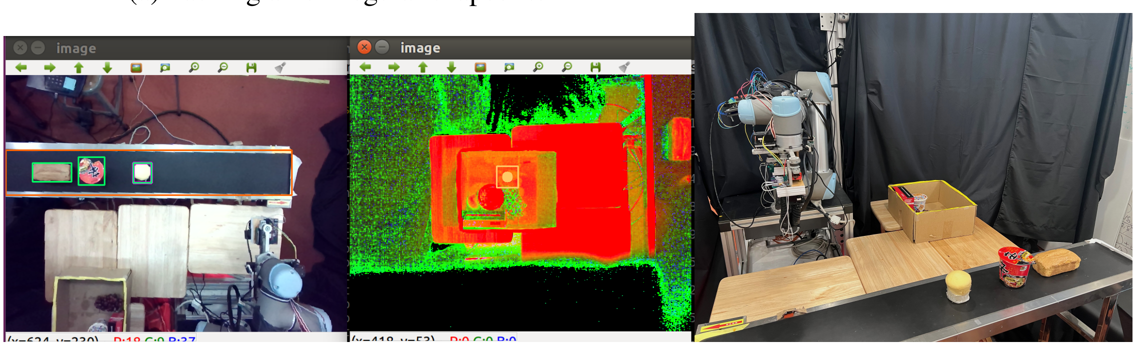
Given the average time to plan and execute robot trajectories from the neutral position to the conveyor belt and the speed of the conveyor belt, we determine a meeting waypoint and the time the object will reach that point. We then use MoveIt! to plan the appropriate path for the UR-5 arm to move and grasp the object at that point.

4.2.3 Online Packing

Once an object is grasped, the vision system then identifies a favorable packing location. First, the vision system locates the edges of the packing box in the RGB image via color segmentation. The mask of the bin is then applied to the registered depth image, leaving only the region of packing interest. Since our gripper's fingers are x cm in diameter, and the gripper widens about the x or y axis of the robot, the mask of the bin is also eroded. This assures that the gripper will not bump into the bin, which could be a safety hazard in the real world.



(a) Packing a non-regular shaped item



(b) Packing a regular shaped item

Figure 4-3: (a) The closest item to the end of the conveyor belt is the grapes, whose bounding box from the Xtion2 camera is converted to the Xtion Pro Live. The middle figures indicates the calculated packing point based on the depth mask, and the right figures show the system in the same moment from an external view. (b) Demonstrating packing a regularly-shaped item, the muffin.

Object dimensions recorded previously during the detection and grasping tasks are translated from the object detection camera's pixels into the packing camera's pixel coordinates. This was done by converting the 2D dimensions into 3D coordinates, then converting them back into 2D dimensions corresponding to the appropriate camera.

A kernel of ones with dimensions of these translated length and width values is convolved with the depth image of the bin to create a heatmap of packing locations. We perform this operation twice: once with the kernel reflecting the current object orientation, and once with the kernel rotated 90 degrees. The packing location is found to be the location with the highest score of the two heatmaps. Although this approach is not the most optimal, it bypasses the computational intractability and training requirements found in contemporary algorithms [17, 38], allowing us to perform online packing without significant pre-computation.

4.3 Tactile Data Pipeline

The tactile sensors have pressure readings that are read through an Arduino, which then spins a ROS node and continuously publishes the raw values as Float32 type numbers. This topic is then subscribed to by the Grocery Packing module, which reads and converts the data stream of raw values as mentioned in the previous chapter, and normalizes the six output values. The clean tactile output is utilized for decision making as explained in Chapter 5.1.2. While the sensor readings are converted continuously, the software system removes any outlier values, such as a negative value or a value that is much higher than the expected range of output based on our characterization.

4.4 Proprioceptive Gripper Control

The HSA fingers are controlled by Dynamixel Servos, which can be read and written to through a microcontroller using the Dynamixel SDK. In this thesis, we interleave between reading the servos' position, velocity, & load value and writing velocity & position commands to the servo. As discussed before, the Dynamixel Servo in our setup cannot read and write simultaneously, so we utilize a state machine to only read when no write commands are given by the Grocery Packing module. In the last iteration, we utilized limit switches to indicate a hardware stop when the open and close limits were reached. After switching to the Dynamixel Servos, we use its proprioceptive properties to know exactly when the gripper's position is out of bounds.

Chapter 5

Grocery Packing Algorithm

The main algorithm for the online packing of groceries consists of decision-making based on the multimodal sensing capabilities of the system. The RGB-D sensors capture the size estimate of the grocery items and safe location to pack, the 3D printed soft fingers grasp the items using proprioception, and the embedded fluidic sensors on the fingers measure the pressure from the grasp. Combining these modalities allows for a fluid and autonomous grocery packing robot, without the system requiring any priors about the item before being spotted on the conveyor belt. Our use of a buffer table exemplifies the safety consciousness of the system, where items classified as delicate will be packed later, on top of the non-delicate items.

5.1 Grocery Packing Algorithm

First, the cameras continuously keep track of the items on the conveyor belt and the grocery bin with the packed items. While there are still grocery items that have not been packed, our multimodal system first checks whether an item has been found on the conveyor belt. If an item exists, the system prioritizes picking up this item as the it will move in and out of the robot arm workspace. For the closest item in reach of the arm, the system records the dimensions of the object as measured by the RGB-D sensors, meets the object on the belt, and grasps it using proprioceptive feedback. A reading from the tactile sensors is taken, and the object's packing priority score is calculated as described in Section 5.1.2. If this score is greater than our initial classification threshold, then the item is packed in the optimal location specified by the vision sensors. If there was no item found on the conveyor belt, the system packs the item in the buffer zone with the highest priority score. This method is also represented in Algorithm 1.

Algorithm 1: Grocery Packing Algorithm

```
input: n grocery items to pack
p ← 0; while n > 0 do
    Items move along conveyor belt;
    if exists item on conveyor belt then
        if closest item is within range of robot then
            RGB-D sensor calculates item dimensions;
            UR5 robot arm grasps object;
            Tactile sensors read pressure for 1 second;
            p = delicacy score based on classifier with test set Pick least delicate
              item from buffer;
            if item's property p ≥ THRESHOLD then
                RGB-D sensor calculates optimal packing location;
                Place item in box;
                n = n − 1;
            end
        end
    else
        Place item in buffer;
    end
end
```

5.1.1 Robot Actions

(a) *Return Home*

The robot in the grocery packing algorithm returns to a fixed neutral home position after completing a pick and place task. At this point, the gripper position is reset to fully open

(b) *Pick Item From Belt*

When the robot is in the home position, the Xtion 2 camera oversees the conveyor to detect the item closest to the end of the belt, and records its dimensions. Using the

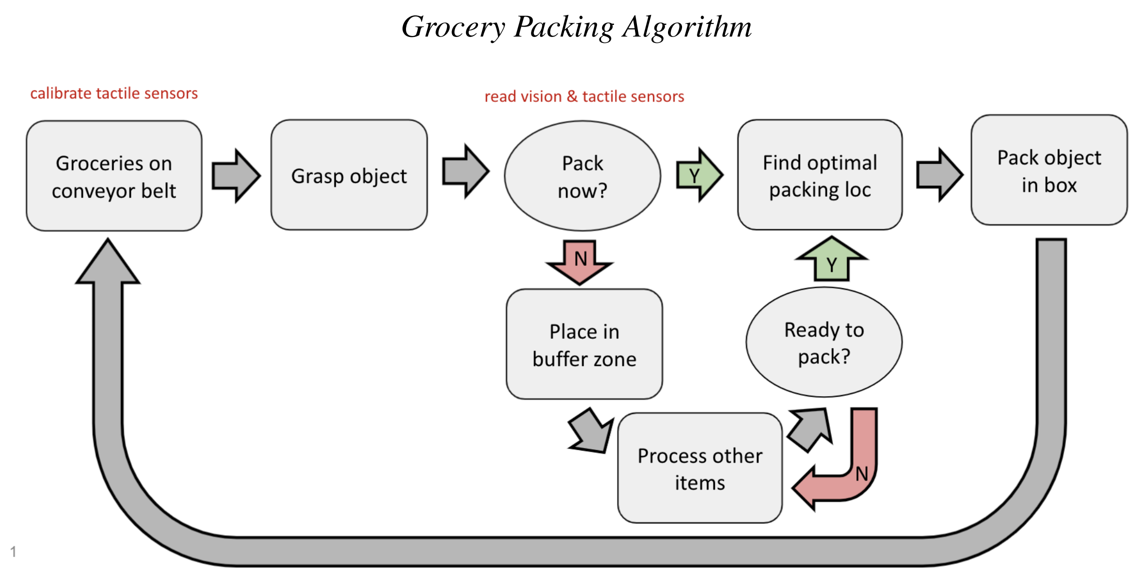


Figure 5-1: A flowchart of our system grocery packing algorithm with multimodal sensing. The first instance of the system is activated when groceries are detected on the conveyor belt. Depending on whether there are items in the buffer or the belt, the robot makes the decision to move to the appropriate location and pack the groceries in a safe way.

estimated time of arrival logic, the robot moves towards the predicted location of the moving item, and grasps it. The robot then lifts the item from the conveyor belt. If utilizing the tactile sensors, the item's stiffness is measured at this point.

(c) *Place Item In Buffer*

At this time, the robot is in possession of a delicate grocery item. It moves towards the first available buffer location, and places the item down, then moves up in the z direction to complete the placing of the item.

(d) *Pick Item From Buffer*

The system holds a record of the items in the buffer zone and its recorded properties of size and stiffness. Based on our metric of delicacy, the robot moves to the least delicate item in the buffer zone, and grasps to pick it up.

(e) *Pack Item In Bin*

The robot is grasping an item, and using the Xtion Pro Live camera, determines the deepest location in the packing bin that can fit the current item. The robot then places the grocery item at that location, either in the robot's end effector's current orientation or rotated 90 degrees. The robot then lifts its arm in the z direction to complete the placement.

5.1.2 Classifying Delicate Items

As grocery packing is complex and optimal packing challenging to define, in this thesis, we utilize a classification based on fragility of objects and object geometry to pack grocery items. We propose that two main properties of each object, detected by our multimodal sensing system, can be leveraged in determining toward the ideal of safe packing: size and stiffness. Size is determined using the vision sensors, and stiffness is determined by the tactile sensors.

Using a calibration set of 25 grocery items, we perform three trials of grasps and calculate the average tactile output per item along with its calculated area. We see a general trend that, due to their compliance, softer objects apply lesser amounts of force to the tactile sensors when the object are grasped. By labeling the calibration set with either rigid or soft and large or small, we can perform a simple binary classification to decide whether an item is rigid or not. We also apply a lower threshold to separate rigid items that do not have sufficient contact with the tactile sensor when grasped due to our soft gripper geometry, as explained in Section II. B.

5.1.3 System Limitations

While the decision making, manipulation, and packing of groceries is done autonomously, there are some system limitations regarding the grocery items and continuous running of the system. First, since the conveyor belt for the system has a belt of 15 cm in width and the gripper has a maximum cross-sectional clearance of 11 cm, the orientation of the items placed onto the conveyor belt were limited to ones that could satisfy the width constraints. If we needed to pack larger items or scale up in size, then we could possibly use a bigger conveyor belt and gripper. Another method is to change the orientation of the robot's end effector itself when grasping the item from the conveyor belt.

As our goal is to run an end-to-end grocery packing system without heavy computation,

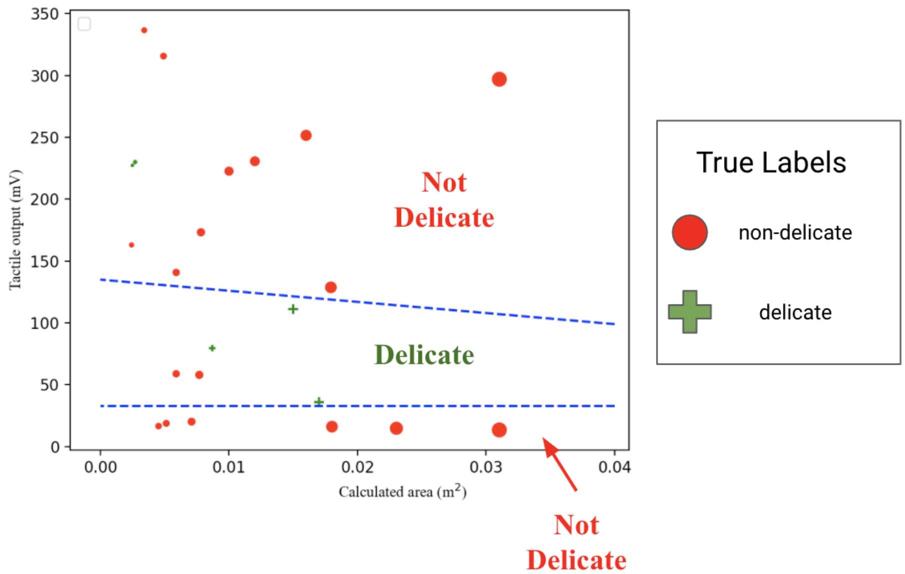


Figure 5-2: Using the calibration set of grocery items, we determine two thresholds for classifying whether the item should be packed immediately or not. Tactile readings can be converted to force outputs using Equation 6.1.

the packing locations chosen are calculated by two convolutions by the depth kernel. This may not lead to the same solution found by a machine learning model, but the simple elegance of the current packing method and cheap and accurate-enough solution works in our setup. With the current gripper design, it is not trivial to achieve 'tight packing' due to each HSA finger being 3 cm in diameter, and the gripper being 23 cm in length.

As explained in Section 5.1.2, we apply a lower threshold to separate rigid items without enough contact with the tactile sensors attached to the fingers. This is partly due to the placement of the sensors at the bottom of the finger; since the fingers are soft, the contact between the tactile sensors and rigid item may not be sufficient 3-5. For soft objects, the finger molds its shape around the item, but for hard objects, the contact area at the bottom of the finger may be completely missed.

5.2 Automation/User Control

In this thesis, we discuss the method for an automatic grocery packing system. This is achieved by using the Grocery Packing ROS node to execute the next desired action in sequence. While the node is spinning, the grocery packing algorithm described above is executed, and based on the current state of the environment, determines the next action for the robot to take. The user will press enter in the terminal to start the program, then can place the next grocery item on the conveyor belt. Since we are only working with one robot packing groceries at a time, the user will place the next item when the robot is finished packing the previous item. The exact timing is not critical however, since the vision system tracks the item down the conveyor belt, and the flexibility of the soft gripper allows for a higher likelihood of successful item grasping.

5.2.1 Continuously Running System

In order to achieve an autonomously running system that can stay active for long periods of time, we require a setup that can detect when a box is full, and stay active by packing multiple boxes. We achieve this condition by taking the mask of the packing box and checking both the height of the highest item and the average height of the items in the box. If there exists an item that exceeds the edge of the box or on average the items are almost near the top of the box, the grocery box is considered full. In our current system, the user is alerted by the system when the box is full, where they can swap the fully packed box with a new empty box.

In future work, perhaps there could be a second mobile robot whose job is to swap the boxes, or place the items on the belt itself. The shape of our current conveyor belt is also limited in this scenario - in most real world scenarios, the conveyor belt is round, and the items would keep rotating, like suitcases in the baggage claim. Another method to avoid manual interception regarding the grocery bag swapping is to place the boxes themselves on this rounded conveyor belt - this way, the system can turn the conveyor belt on to (1) return the fully packed bag and (2) continue to pack automatically with the next empty bag.

Chapter 6

Sensor Characterization

Figure 6-1: 25 hand-labeled calibration items were evaluated to determine the priority packing heuristic. Soft: clementine, grapes, juice box, mozzarella cheese. Medium: book, celery salt, empty soda can, instant ramen, jello, milk carton, mustard bottle, toy peach, toy pear, wheat thins. Rigid: apple, coffee can, fake cake, plastic lemon, Pringles tube, Rubik's dodecahegon, Spam, tea box, tuna can, vegemite, vitamin bottle.

6.1 Vision Characterization

We characterize the vision by running a test set of 25 objects of various sizes 6-1. We use the camera to calculate the width and length of the object and calculate area as the product of the two. The figure 6-2 represents the actual size vs the estimated size of each test object, which closely correlates to the line $y = x$, showing a relatively accurate estimate of size by vision. Using this, we determine a threshold for 'small' and 'large' items used in the initial determination of whether to pack a test object or place it in the buffer zone to be packed later.

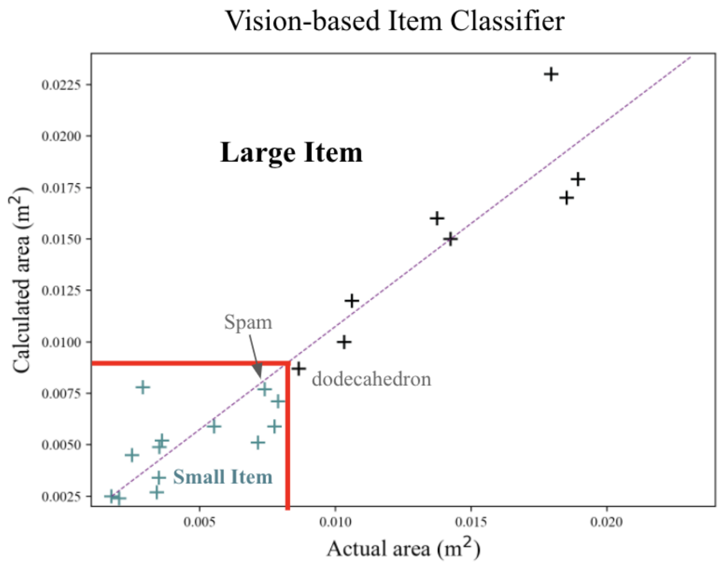


Figure 6-2: We evaluate the vision sensing method of using the item's area by comparing the actual area of the item from a bird's eye view of the conveyor belt with the system's calculation of item area based on the vision data.

6.2 Tactile Characterization

To characterize the soft tactile sensors, we conducted a series of experiments where weights were placed upon a sheet of cardboard (.8g, used to distribute weight over the different hexagonal sensors) on each sensor cap, which has seven hexagonal sensors (Figure 6-3). Three trials were collected for each weight class, and weight totals ranging from 10g to 200g in 10g increments were used, since measured forces at the gripper fingers range from 0.75 N to 2 N, depending on where on the fingers the object is grasped [7]. Data were collected and plotted for three sensors on each sensor cap for readability. The fitted line allows conversion from millivolts to Newtons of force as per equation 1.

$$force = (sensor\_output - 5.46)/.25 \tag{6.1}$$

The conversion metric we determine here allows for interpretation of the tactile sensors

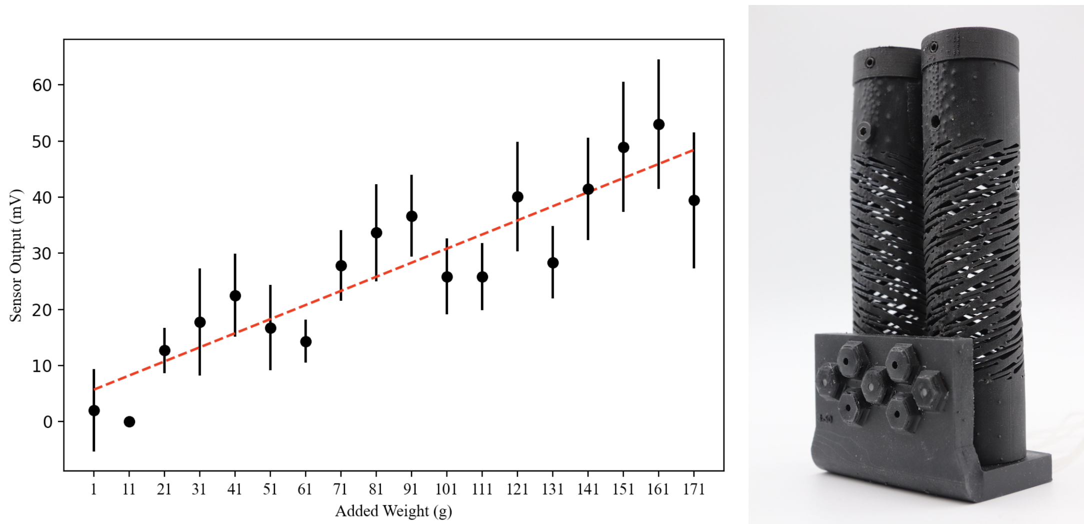


Figure 6-3: Left: Evaluation of 3 tactile sensors on one tactile sensor array against increasing applied load. The three outputs from one array are averaged to give a single measured output in millivolts. Standard deviation is also shown. Right: Hardware setup with digitally-fabricated HSA fingers and digitally-fabricated hexagonal tactile sensors.

in our system, where we convert the raw sensor data into an understandable force in millivolts, and force, before the gripper grasps the item on the conveyor belt and takes the tactile sensor reading.

6.3 Proprioceptive Characterization

We utilize three Dynamixel Servos in our gripper system - the main belt servo to control the parallel close and two for the fingers. To account for grasping unknown items, we use the velocity control mode for the main belt servo, setting a constant velocity to close the gripper. When the gripper reaches a firm grasp on the item, the belt servo's load values spike and we set the servo velocity to zero.

Similarly, when we finish packing the item in the bin, the gripper will move the box or disturb other objects if it opens fully. As a result, we set a constant velocity value to open the gripper until the load values reach zero, indicating that the item is fully released.

Chapter 7

Experimental Results

Grocery items may be fragile, deformable, irregularly-shaped, perishable, and/or uncooked, all of which contribute to an opaque idea of "optimal" packing for groceries [27]. We choose a human-legible method of evaluation for our grocery packing task based on scoring with a rubric, allowing for more intuitive understanding as well as potential expansion to reinforcement learning or other methods.

Robotics for grocery retrieval, picking and packing has seen recent developments. Aquilina et al. [3] performed concept validation of a robotic self-checkout system; however, the system relies on hard-coded information about the grocery items, the packing procedure considers geometry only, and the suction-based end effector limits the graspable items.

Figure 7-1: System evaluation was conducted with 15 test items. Soft: bread, chips, kale, muffin, seaweed. Medium: cheese, crackers, gum, Pringles, stroopwafels. Rigid: baking soda, ice cream, soup can, sprinkles, pot roast. Delicate: bread, chips, crackers, kale, muffin, seaweed, stroopwafels. Heavy: baking soda, gum, ice cream, pot roast, soup, sprinkles, stroopwafels.

7.1 Experimental Setup

Our experimental setup consists of a UR5 robot arm outfitted with a compliant robotic gripper referenced in [7], modified to use 3D printed handed shearing auxetics [35]. In front of the robot is a conveyor belt running at constant speed (.10 m/s) where items are loaded manually from the end further from the robot. Three small tables are supplied adjacent to the robot, which provide buffer space to place the items aside before they are packed. To the side of the robot is a cardboard box, which serves as the packing bin, and has colored markers at the edges for detection by the vision system. An ASUS XtionPro is mounted above the packing bin, and an ASUS Xtion2 is mounted above the conveyor belt.

For the purposes of the grocery packing experiments, the following assumptions are made to the system. First, the robot picks up one item at a time from the conveyor belt, where the gripper grasps the item at its center point. The next items will be placed on the conveyor belt by the user once the robot is done packing the previous item or the robot is in an idle state. Next, the conveyor belt moves at a constant speed throughout the course of all experiments, so the Xtion2 camera can predict where the tracked items will be in a future point of time. Lastly, we classify items into two categories: delicate or not delicate, and utilize the classification to determine whether the robot will place the item in the buffer or bin.

7.2 Task Evaluation

To evaluate our multisensing system, we conduct three grocery packing experiments: (1) a baseline experiment blind to each object's properties, (2) a vision-only experiment that only considers the item's size dimensions, and (3) a multimodal experiment that combines proprioception, tactile, and vision. For each of these experiments, we run three trials of packing a full order of ten items, for a total of nine experiments. We then evaluate the safety-consciousness of the different types of experiments with the use of a packing rubric, focusing on the extent to which the system's decision making was safe.

- **Baseline:** Vision and proprioceptive outputs are used only to ensure objects are grasped. All objects are packed immediately and in the center of the box.

- **Vision-Only:** Vision and proprioceptive outputs are used to ensure objects are grasped. In addition, vision provides an estimate of object shape (length and width) that is used to calculate the location where the object is packed. Items with object size greater than a threshold (determined experimentally to be .008 square meters using the calibration items in Figure 6-2), approximated as the top-down area calculated by $area = length * width$, are packed immediately, and smaller items are packed later. Items in the buffer are packed by order of decreasing size.

- **Multimodal:** Vision and proprioception are used to ensure objects are grasped. The length of the object estimated by vision is updated by the proprioceptive data from the gripper when it has closed around the object. A combined weighted heuristic of size and stiffness, discussed in section IV. B. 2) and determined by the tactile, proprioceptive, and vision systems, is used online to determine the order in which objects are packed. Vision and the size estimate are again used to determine the packing location for the object.

Three trials are performed for each experiment with different object sequences of test objects. These trials are as follows:

- **Trial 1:** Kale, ice cream, crackers, seaweed, pot roast, baking soda, muffin, chips, gum, soup

- **Trial 2:** Bread, kale, stroopwafels, pot roast, muffin, cheese, chips, sprinkles, gum, crackers

- **Trial 3:** Cheese, muffin, crackers, pot roast, soup, chips, stroopwafels, Pringles, ice cream, seaweed

7.2.1 Packing Rubric

For evaluation of the system, target objects were given two labels: delicate or not delicate, and heavy or not heavy. We define a human-legible rubric for evaluation taking into account the ambiguous qualities that contribute toward "good" grocery packing [27]. Each occurrence of a heavy item dropped on a delicate item was penalized during scoring of the experiments.

A lower penalty score on a packed box indicates a more safe and optimal bin packing.

Table 7.1: Scoring rules and associated penalties for packed bins

Criterion	Indicators	Penalty
Partial	Item occludes bin edge (top-down view)	3
Potentially Damaging	Heavy item packed on delicate item	2

7.3 Results

Overall, the results shown in Table 7.2 indicate that the vision-only-based system performs far better than the blind system to safely pack grocery items, and on average, the multimodal system outperforms both the vision-only and blind systems. As expected, the baseline experiment results in poor packing, with an average of six potentially damaging occurrences of a heavy item dropped on a fragile item per trial. The vision experiment produces improved packing performance, with an average of three potentially damaging occurrences of a heavy item dropped on a fragile item per trial. Average potentially damaging occurrences per trial drop to less than one for the multimodal experiement.

Table 7.2: Experiment penalty scores

Trial	Partial	Potentially Damaging	Total
Baseline 1	0	7	14
Baseline 2	0	7	14
Baseline 3	0	4	8
Vision 1	0	6	12
Vision 2	0	2	4
Vision 3	0	1	2
Multimodal 1	0	0	0
Multimodal 2	1	2	7
Multimodal 3	1	0	3

7.3.1 Baseline Results

The baseline experiments had a grasping success rate of 100%, with ten grasps per trial for a total of 30 successful grasps. Since the baseline scenario packs the item in the middle of the box, no items are placed in the buffer. One box is packed with ten grocery items per trial, for a total of three packed boxes. While there were no failures with respect to grasping, the packing score reflected poorly, as the system was not considering the size nor overall delicacy of the grocery item when packing its order. Overall, the baseline experiment results in poor packing, with an average of six potentially damaging occurrences of a heavy item dropped on a fragile item per trial.

7.3.2 Vision-Only Results

There were a total of 42 successful grasps out of 42 for the three vision-only trials. While there are still ten items per trial, the buffer held four ’delicate’ items, as defined by the size factor mentioned in the task evaluation. Again, there were a total of three packed boxes that were considered full. One unexpected placement for Trial 1 was the placement of the soup can in the buffer, where it was placed in a different orientation than when picked up from the conveyor belt. This was caused from a change of center of mass while grasping the object. However, since the fingers are soft, they adapted to the altered orientation and still placed the item carefully onto the buffer, and picked it up safely when packing into the bin. Overall, the vision experiment produces improved packing performance, with an average of three potentially damaging occurrences of a heavy item dropped on a fragile item per trial.

7.3.3 Multimodal Results

For the multimodal final experiment, there were 45 successful grasps out of 45. The buffer for this experiment held five items to be considered ’delicate’: bread, kale, stroopwafels, muffin and cheese, which are all soft to the human touch. Three boxes were packed fully, where nondelicate items were packed first then delicate items were picked from the buffer to the grocery bin.

In Trial 2, the first item was not detected due to the user misplacing the item to occlude the entire conveyor belt from the RGB-D camera, and the robot did not attempt to grasp the item. A second attempt was made with the correct placement, which resulted in a successful grasp. Since color segmentation of the belt was enabled, covering the entire width of the conveyor belt will cause an error from the system.

All penalties incurred during the multimodal tasks embody intricacies of real-world grocery packing. The two penalties for potentially damaging packed items for Multimodal 2 arise from the bag of stroopwafels being packed on top of other delicate items. This case of packing an item that is both delicate and heavy illuminates the complexity of the grocery packing task; prioritizing the safety of this item could mean potentially damaging other objects, yet packing it first could result in the object being damaged. The partial packing penalty incurred for both Multimodal 2 and Multimodal 3 trials are due to a slight occlusion of the bin edge by the plastic bag of the bread. If the bread, a large object, had been packed earlier in an emptier bin (such as in Baseline 2 and Vision 2), it would likely land entirely inside the bin at the cost of being crushed under subsequently packed items. Intuitively, if a bin were being packed for a human to carry (eg. from shopping cart to car), placing delicate, bulky items at the top of the bin would likely be prioritized over strictly fitting all items within the walls of the bin.

Figure 7-2: Packed bins for each of three trials for baseline, vision, and multimodal experiments.

Chapter 8

Discussion & Future Work

In this thesis, we have achieved a soft robotic system that leverages multimodal sensing input to pack groceries towards a human-legible metric of "well-packed". We have shown that the combined multimodal system achieves greater performance than baseline and vision-only systems and uncover tradeoffs in packing optimality inherent in the complex nature of the grocery packing task. With this, we present an end-to-end grocery packing system that can autonomously handle dynamic item grasping, packing, and real-time decision-making. When the box is full, it can be swapped out for another box without full pausing the system. The robot can continue packing, as the RGB-D sensors track the items in the conveyor belt and box continuously in real time. By utilizing the combination of tactile sensing and proprioception, the system can detect the fragility of an item and ensure a firm grasp but one that could damage the grocery item.

Combining the cost-efficient color segmentation method to detect and track the items down the conveyor belt, and using the constant speed of the belt to calculate the motion planning to grasp and pick up the moving item allows for a reliable item grasping subsystem. By running the tracking subsystem as its own ROS node, the items can get tracked and have their estimated 3D pose calculated continuously. The same can be said about the ROS node for the packing subsystem, where the RGB-D sensor continuously evaluates the lowest packing area available for the item the robot is currently grasping. By running two convolutions, we can determine a packing pose considering the size of the item.

A modular multimodal system such as the system we currently have could possibly be extended to real world deployment with adjustments to the design of the gripper and tactile sensors, along with a larger-scale and more realistic setup. In addition, the speed and accuracy of the grocery packing module would go up if we were to add more robot agents into the environent - two robot arms packing groceries down the conveyor belt simultaneously.

8.1 Lessons Learned

8.1.1 Hardware

The initial obstacle was understanding how to use the UR5 robot since it was my first time working with a robot arm hardware. In particular, the challenging aspects consisted of understanding the coordinate frames of the different joints of the arm and their relationship to the external RGB-Depth cameras and how the commands get translated from joint goal commands to pose goal commands. Another restriction was the number of objects we had in the environment that we did not want to robot to bump into: the box, tables, conveyor belt, and of course, the humans. I learned how to tell the robot to exit the search if it found a path that was out of bounds or contained too many way points, indicating a long and inefficient path to the goal pose.

Another obstacle in the hardware system, especially during long experiments, was the adapter between the Dynamixel servo and the gripper. This was a vital connection as it allowed the servo to open and close the gripper, and it was important that this connection did not break or slip significantly while the grocery packing system was active. When resolving this issue, it was especially important to understand the stress points of hardware and why the parts broke - this adapter for the Dynamixel servo particularly kept breaking since the 3D printed part was too thin and sheared off. Our fix was to redesign the hardware to avoid high torsion on the adapter, as the belt mechanism in our gripper module led to the 3D-printed adapter to be the stressor point. After long use, I learend it is important to turn off or reboot the hardware system, especially the Dynamixel Servos, since certain features were overridden. This also applies to the software side of the system, since the cache can build up and sometimes result in strange system behavior.

Lastly, depth data from the cameras are messy! From the software side, I applied additional filtering and averaging to the 2D camera streams to ensure that the conversion from 2D pixel to 3D world coordinate was not erroneous, and to ignore depth values that did not logically make sense, such as 0 or nan.

8.1.2 Algorithms

It is important to consider the main priorities of the specific system when implementing decision making of the system. For example, in the current system, the conveyor belt is short and linear, meaning that it is a priority for the robot to not let any of the items fall off the belt. If the belt were round, the priority would change. Another important part of the algorithm is to remember the hardware integrating into it - for example for the packing, I initially did not consider the maximum open and close dimensions of the gripper and assumed its size to be static when implementing the packing algorithm. The robot must account for how far the gripper may open and make sure the pose chosen will not cause the robot to bump into other items or the box.

Another thing I learned was efficient motion planning for a path with many environment obstacles added as a safety mechanism. For example, there are different algorithms to use for searching through the state space - we used RRTConnect but the original setting was at RRT, which only searches through random tree from the start pose rather than both the start and end poses. In addition, we can also select the segment length, which determines how fine-grained we want the search to be. Overall, despite using the assistance of MoveIt! for motion planning, there needed to be a comprehensive understanding of how the UR-5 moves and why it moves in that particular way.

8.1.3 Experiments

When running experiments, I learned that it is critical to stay organized and record all data, particularly when there are many moving parts in the system for each experiment. For example, in addition to the footage filmed on the camera, we also kept rosbags with the action history of the system, and took photos of the resulting packed bin.

Another critical aspect of the system I realized when getting ready to conduct the experiments was to have a modular or easy-to-read codebase, in case the setup ends up changing or different experiments need to get run in sequence. One method I used was to store rosparams, where the user can alter the settings for the 1) experiment mode, 2) running with the gripper on, and 3) move the robot. Since the sun casted different shadows on the conveyor belt in the lab room every day, having a method to check the camera stream and calibrate the box and color segmentation was crucial.

8.2 Future Work

Future work includes the expansion of tactile, vision, and proprioceptive data to further explore physical properties of unknown objects. For example, exploration of soft sensor array configurations towards better ensuring adequate contact between the soft fingers and target object could occur, to remove the lower threshold that existed for the item delicacy classifier. Next, estimation of geometric regularity from visual data was implemented in our research but not used in experiments, and could be used if we alter the packing algorithm to consider 'tight' packing. Another interesting direction is to build upon the multimodal sensing for determining the delicacy of an item, and use secondary estimates to calculate object size and stiffness from proprioceptive data.

In the experimental results, a packing rubric was used to evaluate the safety of the items packed in the bin. Assigning penalties and rewards to this packing rubric gives way to reinforcement learning applications, where the system could potentially learn the delicacy classification of the item based on the extra measurement from proprioceptive feedback.

If extended to a real world deployment, the system would require a slightly altered setup - one larger in scale, both in terms of the surface area of the conveyor belt and buffer zone to handle more items.

Appendix A

Grocery Packing Demo Instructions

1. Turn on the UR-5 Robot cart. Turn on the UR controller, then initialize and start the robot. Check the robot's IP address and confirm that it matches with the address listed in `moveit_UR5.launch`, found in the `moveit_planner` ROS package.

2. Ensure the cables for the Ethernet, gripper, and both RGB-Depth cameras are connected to the laptop.

3. Turn the conveyor belt on by flipping the switch, and turn on the power for the gripper (Dynamixel Servos).

4. Open the terminal window on the laptop, and run `roslaunch moveit_planner grocery_demo.launch`.

5. Once all the programs are loaded, the terminal will say "Press to start". Press the enter key to activate the grocery packing system, and the user will not need to interact with the terminal until their desired termination of the system. The robot will stay idle until the first grocery item is detected on the belt. Place each grocery item in the center of the conveyor belt such that it does not occlude the entire width of the belt. When the robot moves back to its idle position, place the next grocery item on the belt.

6. Here is the param file for running the grocery system. To switch between the different experiment modes discussed in the thesis, use '1' for baseline, '2' for vision-only,

and '3' for multimodal.

```
show_packing: false
show_obj_detection: false
gripper: true
move_robot: true
mode: 3 # 1: baseline, 2: vision-only, 3: multimodal
calibrate: false
```

Appendix B

Code

Grocery Packing module

```
#!/usr/bin/env python

from collections import namedtuple
import numpy as np
import rospy

from moveit_planner.UR5Arm import *
from moveit_planner.object_utils import ObjectUtils
from geometry_msgs.msg import *
from moveit_msgs.msg import *
from moveit_msgs.srv import *
from trajectory_msgs.msg import *
from std_msgs.msg import Float32MultiArray, Int8, Int32MultiArray, Int32, Float32
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState

CENTER_TOPIC = '/boxcoord' #to subscribe object center coordinates to
BIN_JOINT_GOAL = [4.4839324951171875,-1.0680277983294886, 1.0908217430114746,-1.593987766896383, -1.5700305143939417,
    2.913686513900757] # fixed joint state location above box - allows for faster performance in RRT Connect planning

# dynamic picking params
TRAVEL_TIME = 4 # seconds allowed for robot to plan + travel to object meeting point
MIN_Y = -.43 # meters; defines area on conveyor belt we allow the robot to move to
MAX_Y = .2 # meters
BUFFER_ZONE_Z_HEIGHT = .3 #.37
GRIPPER = rospy.get_param("gripper", False)
EXPERIMENT_MODE = rospy.get_param("mode", 1)
TACTILE_TOPIC = "/tactile_out"
WILL_EXECUTE = rospy.get_param("move_robot", True)
CALIBRATION_FREQ = 3

ADC_OFFSET = 1.25
ADC_COEFF_VOLTS_PER_TICK = 2.048/16777215.0
ADC_COEFF_MV_PER_TICK = ADC_COEFF_VOLTS_PER_TICK * 1000

NUM_TACTILE_SENSORS = 6
AREA_THRESHOLD = 0.008 # determined from vision characterization

```

```
# ROBOT ACTIONS
PICK_FROM_BELT = 0
PLACE_IN_BUFFER = 1
PICK_FROM_BUFFER = 2
PACK_IN_BIN = 3
HOME = 4
STANDBY = -1

# EXPERIMENT MODES
BASELINE = 1
VISION = 2
MULTIMODAL = 3

# CLASSIFIER PARAMS - calculated from calibration experiments
Y_INTERCEPT = 135.0
SLOPE = (Y_INTERCEPT -90.9)/-0.05
CONTACT_THRESHOLD = 21

class GroceryPacking():
    def __init__(self):
        self.tactile_zeros = np.zeros((6,))
        self.calibrate_sensor_count = 0

        self.gripper_pub = rospy.Publisher("/move_gripper", Int8, queue_size=0) # 1: open, 2: close, 3:auxOpen, 4:
            auxClose
        self.buffer_pub = rospy.Publisher("/buffer_flag", Point, queue_size=0)
        self.tactile_sub = rospy.Subscriber(TACTILE_TOPIC, Int32MultiArray, callback=self.tactile_cb)
        self.slip_pub = rospy.Publisher("/slip_amt", Int32, queue_size=0)
        self.normalized_tactile_pub = rospy.Publisher("/normalized_tactile_out", Float32MultiArray, queue_size=10)
        self.toggle_pub = rospy.Publisher("/toggle", Int8, queue_size=0)
        self.score_pub = rospy.Publisher("/score", Float32MultiArray, queue_size=0)
        self.gripper_dist_sub = rospy.Subscriber("/gripper_width", Float32, callback=self.gripper_dist_cb)
        self.torque_pub = rospy.Publisher("/torque_on", Int8, queue_size=0)
        self.size_check = rospy.Publisher("/size_check", Int8, queue_size=0)

        self.execute = {PICK_FROM_BELT: self.PickFromConveyorBelt, PLACE_IN_BUFFER: self.PlaceInBufferZone,
            PICK_FROM_BUFFER: self.PickFromBufferZone, PACK_IN_BIN: self.PlaceInBox, HOME: self.GoToNeutralPose}
        self.arm = UR5Arm()
        self.utils = ObjectUtils()
        self.result = None
        self.reset()
        self.custom_offset = 0
        self.normalized_tactile_vals = [0,0,0,0,0,0]
        self.missed = False

        self.arm.printOrientation()

    def gripper_dist_cb(self, data):
        dist_cm = data.data
        if self.current_item is not None:
            self.current_item.proprio_width = dist_cm

    def tactile_cb(self, data):
        """
        Callback for tactile data stream of NUM_TACTILE_SENSORS raw values. We convert to mV and
        normalize the data. If sensor value is too low (aka no contact point), disregard when
        averaging values. Assumption: harder objects give higher readings.
        """
        output = data.data # array of 6 sensor values
        normalized_data = np.zeros((NUM_TACTILE_SENSORS,))

```

```
        count = 0
        for i in range(len(output)):
            normalized = (output[i] + ADC_OFFSET/ADC_COEFF_VOLTS_PER_TICK) * ADC_COEFF_MV_PER_TICK
            if -200 < normalized < 10000: # within reasonable range
                normalized_data[i] += normalized
                count += 1

        self.normalized_tactile_vals = normalized_data - self.tactile_zeros

        # publish normalized tactile out
        msg = Float32MultiArray()
        msg.data = list(self.normalized_tactile_vals)
        self.normalized_tactile_pub.publish(msg)

    def average_tactile_out(self, num_seconds, zero=False):
        # publish tactile calibration start
        toggle_msg = Int8()
        toggle_msg.data = 7
        self.toggle_pub.publish(toggle_msg)

        tactile_zeros = self.tactile_zeros if zero else 0

        count = 0
        total = 0.0
        now = rospy.Time.now()

        while (rospy.Time.now().to_sec() - now.to_sec()) < num_seconds:
            total += (self.normalized_tactile_vals + tactile_zeros)
            count += 1

        # publish tactile calibration end
        self.toggle_pub.publish(toggle_msg)

        return total / float(count)

    def clean_tactile_data(self, data):
        THRESHOLD = 5.0

        output = data[data > THRESHOLD]
        if len(output) == 0:
            return 0.0, 0
        return sum(output) / float(len(output)), max(output)

    def reset(self):
        self.clear_buffer_occupancy()
        self.current_item = None
        self.packed_items = []
        self.current_action = HOME
        self.history = [self.current_action]
        self.GoToNeutralPose()

        self.item_count = 0

    def run(self):
        """
        Executes the sequence.
        """
        print "starting run"

        self.current_action = self.get_next_event(self.result)
        # self.current_action = self.grab_object()
```

```
        rospy.loginfo("NEXT ACTION: {}".format(self.current_action))

        # If a valid action exists, execute it
        if self.current_action != STANDBY:
            self.result = self.execute[self.current_action]()

            # publishing current action
            msg = Int8()
            msg.data = self.current_action
            self.toggle_pub.publish(msg)

        self.history.append(self.current_action) # keep track of past actions

    def clear_buffer_occupancy(self):
        """
        This resets the buffer occupancy. The buffer currently has 5 hardcoded locations, where the dict
        key is the buffer id and the value is [position wrt robot, Item object]
        """

        self.buffer_occupancy = {0: [(-.42,  .1, -.1), None],
                                 1: [(-.42, -.1, -.1), None],
                                 2: [(-.42, -.3, -.1), None],
                                 3: [(-.42, -.5, -.1), None],
                                 4: [(-.42, -.7, -.1), None]}

    def add_packed_item(self, item):
        self.packed_items.append(item)

    def update_item_for_buffer(self):
        """
        Determine the next item to pick from the buffer and communicate the item's properties
        to the packing logic (by publishing to a new topic).
        """
        updated_size = Point()
        buffer_item, _, _ = self.get_least_delicate_in_buffer()
        updated_size.x, updated_size.y, updated_size.z = buffer_item.width, buffer_item.length, 1
        self.buffer_pub.publish(updated_size)
        print("publishing buffer flag with new size ({}, {})".format(buffer_item.width, buffer_item.length))

    def get_next_event(self, result=None):
        """
        Packing algorithm logic.
        1. If item on conveyor belt, pick it up. Else check if items are in buffer.
        2. If item is delicate, place in buffer. Else pack in bin.
        3. Go home/standby.
        """

        if result is False: # if solution not found on belt, go home
            return HOME

        if self.current_action == HOME or self.current_action == -1: # at home

            if self.utils.center_pixels != (0,0): # Item on belt
                if self.ItemIsReachable():
                    print("center is currently", self.utils.center_pixels)
                    return PICK_FROM_BELT # pick from conveyor belt

                return STANDBY # wait until item is within reach. Prioritizes item on belt over buffer item.

            else:
```

```
                for buffer_id, (buffer_loc, item) in self.buffer_occupancy.items():
                    if item is not None:
                        print("Picking from: ",self.get_least_delicate_in_buffer())
                        return PICK_FROM_BUFFER # pick from buffer
                return STANDBY # do nothing

        if self.current_action == PICK_FROM_BELT:
            if self.current_item is None:
                return STANDBY
            elif self.IsBufferFull(): # pack in bin no matter what if buffer is full
                return PACK_IN_BIN
            elif self.current_item.isDelicate():
                return PLACE_IN_BUFFER # place in buffer
            else:
                return PACK_IN_BIN # place in bin

        if self.current_action == PLACE_IN_BUFFER: # go home after placing item in buffer
            return HOME

        if self.current_action == PICK_FROM_BUFFER: # pack item after picking from buffer
            return PACK_IN_BIN

        if self.current_action == PACK_IN_BIN: # go home after placing item in bin
            reset_buffer_flag = Point()
            self.buffer_pub.publish(reset_buffer_flag)
            print("Resetting buffer flag")
            self.publish_slip_val(100)  # account for gripper slip
            return HOME

    def grab_object(self):
        if self.current_action == HOME:
            return PICK_FROM_BELT
        elif self.current_action == PICK_FROM_BELT:
            rospy.sleep(1.0)
            if GRIPPER:
                self.OpenGripper()
            return HOME

    def GoToNeutralPose(self):
        """
        Robot always starts and ends in the neutral position to avoid occluding the items on the conveyor belt and packing box.
        """
        print_message("4 Go To Neutral Pose")

        # open gripper
        if GRIPPER:
            print("opening gripper from GroceryPacking node!")
            self.OpenGripperFull()

            if EXPERIMENT_MODE == MULTIMODAL and self.calibrate_sensor_count % CALIBRATION_FREQ == 0: # go to calibrate state
                self.FingerClose()

        if self.history[-1] == PACK_IN_BIN and WILL_EXECUTE:
            success = self.arm.LoadandExecutePlan("binToHome", self.arm.arm_group)
            if success:
                return

        robot_pose = self.arm.defineWaypoint()

```

```
        success = self.arm.MoveToPoseGoal(robot_pose, self.arm.arm_group, event_name="Neutral Pose", willExecute=WILL_EXECUTE)

        print("Going to Neutral Pose with x: {} y: {} z: {}".format(robot_pose.position.x, robot_pose.position.y, robot_pose.position.z))

        # Multimodal addition: calibrate the sensors every 3 turns
        if EXPERIMENT_MODE == MULTIMODAL:
            if self.calibrate_sensor_count % CALIBRATION_FREQ == 0:
                print("Recalibrating tactile out...")
                self.tactile_zeros = self.average_tactile_out(2.0, zero=True)
                print("Finished calibrating with new offset of {}".format(self.tactile_zeros))
                if GRIPPER:
                    self.OpenGripperFull()

            self.calibrate_sensor_count += 1

        return robot_pose, success

    def ItemIsReachable(self):
        """
        Returns true if the robot's length can reach the item on the conveyor belt.
        """
        target_loc = self.utils.get_future_loc(TRAVEL_TIME).y

        if target_loc < MIN_Y:
            print("Item is too far out of reach - will try again.")
            return False

        if target_loc > MAX_Y:  # we will not get to the item in time
            print("Robot will not get to item in time! Aborting")
            return False

        return True


    def PickFromConveyorBelt(self):
        """
        Event 0: If the target item is reachable, calculate a reasonable location for the robot to pick up the item.
        """
        print_message("0 Pick From Conveyor Belt")

        # If item is not recognized, set dimensions to (0,0); otherwise set to its bounding box width and length

        while np.isnan(self.utils.item_width) or np.isnan(self.utils.item_length):
            size = (self.utils.item_width, self.utils.item_length)

        size = (self.utils.item_width, self.utils.item_length)

        rospy.sleep(.5)

        robot_pose = self.arm.arm_group.get_current_pose("ee_link").pose

        # move to anticipated item location with z buffer
        travel_time = TRAVEL_TIME
        target_loc = self.utils.get_future_loc(travel_time)
        if target_loc.y > MAX_Y: # we will not get to the item in time
            print("Robot will not get to item in time! Aborting")
            return False

        # check if the item will still be out of our reach in TRAVEL_TIME; if so, set to wait at edge of boundary
```

```
        if target_loc.y < MIN_Y:
            target_loc.y = MIN_Y
            travel_time = self.utils.travel_time_to(target_loc)

        self.item_count += 1

        # create new Item object for the item we are picking up
        self.current_item = Item(id=self.item_count,
                                 size=size,
                                 packing_pose=self.arm.getPackingLoc())

        rospy.loginfo("New item: id {}, size ({}, {}), packing_pose ({}, {}, {})".format(self.current_item.id, self.current_item.width, self.current_item.length, self.current_item.packing_pose.position.x, self.current_item.packing_pose.position.y, self.current_item.packing_pose.position.z ))

        # Robot pose for grasping item off the conveyor belt
        robot_pose.position.x -= .3  # assuming that item is placed in center of conveyor belt, which is 30 cm away from the robot
        robot_pose.position.y = target_loc.y +.01  # Note: this constant is an offset dependent on the lighting and shadows.
        robot_pose.position.z = .17  # z location above conveyor belt

        eta = rospy.Time.now() + rospy.Duration.from_sec(travel_time)
        # print("Target intercept location: {} with eta {}".format(robot_pose, eta))

        success = self.arm.MoveToObject(robot_pose, self.arm.arm_group, eta, willExecute=WILL_EXECUTE, event_name="Pick from Belt")

        # If robot could not execute the plan
        if not success:
            print("Did not move to item location on conveyor belt")

        else:
            print("Successfully moved to item location on conveyor belt")
            self.item_in_conveyor_belt = False

            # Close gripper
            if GRIPPER:
                self.CloseGripper()
                print("GP CloseGripper done!")

            # Raise arm after grasping item
            self.RaiseArm()
            print("Successfully picked up item")

            if EXPERIMENT_MODE == MULTIMODAL:
                # aux close a little more
                if GRIPPER:
                    self.FingerCloseExtra()

                print('##### press for tactile calibration #####')
                input = raw_input()
                # take one second to analyze tactile output at this time, determine score.
                initial_tactile_out = self.average_tactile_out(1.0)
                avg_value, self.current_item.tactile_out = self.clean_tactile_data(initial_tactile_out)
                print("Finished calibrating with new val of {}".format(self.current_item.tactile_out))
                print("average value is", avg_value)

                self.current_item.setScore() # updates score with tactile info
                score_msg = Float32MultiArray()
                score_msg.data = [self.current_item.width * self.current_item.length, self.current_item.tactile_out]
```

```
                self.score_pub.publish(score_msg)
                print("New score for item {} is {}".format(self.current_item.id, self.current_item.score))

            # SUBGOAL
            self.picked_from_conveyor_belt = True

    def PlaceInBufferZone(self):
        """
        Event 1: Place the item in an available spot in the buffer zone.
        """
        print_message("1 Placing in Buffer Zone")

        print("Current item is: {}".format(self.current_item.id))

        for buffer_no, (buffer_loc, item) in self.buffer_occupancy.items():
            if item is None:
                self.buffer_occupancy[buffer_no][1] = self.current_item  # update buffer occupancy
                self.item_in_buffer = True
                updated_buffer_loc = buffer_loc[0], buffer_loc[1], BUFFER_ZONE_Z_HEIGHT

                success = self.arm.MoveToCustomPose(*updated_buffer_loc, willExecute=WILL_EXECUTE)
                print("Moving to buffer zone {} with loc {}, success? {}".format(buffer_no, updated_buffer_loc, success)
                    )

                # Lower arm to the buffer zone location
                robot_pose = self.arm.arm_group.get_current_pose("ee_link").pose
                robot_pose.position.z = .09 # hardcoded z location for the buffer zone tables

                self.arm.MoveToPoseGoal(robot_pose, self.arm.arm_group, event_name="Placing object", willExecute=
                    WILL_EXECUTE)

                break

        # Open gripper and place the item
        if GRIPPER:
            self.OpenGripperParallel()

        # SUBGOAL
        self.item_in_buffer = True
        rospy.loginfo("Item {} added to buffer zone {}".format(self.current_item.id, buffer_no))

        # Raise robot arm after placing item in buffer
        self.RaiseArm()

        # note: hardcoded fix for buffer zone 4 - it is furthest from the robot, so the motion planning
        # gets stuck sometimes. fix is to have the robot move to an intermediate pose first.
        if buffer_no == 4:
            success = self.arm.MoveToCustomPose(-.4, -.5, BUFFER_ZONE_Z_HEIGHT, willExecute=WILL_EXECUTE)

    def get_least_delicate_in_buffer(self):
        least_delicate = None
        chosen_buffer_no = None
        chosen_buffer_loc = None

        print("Current state of buffer", self.buffer_occupancy)
        for buffer_no, (buffer_loc, item) in self.buffer_occupancy.items():
            if item is not None and (least_delicate is None or item.score > least_delicate.score):
                least_delicate = item
                chosen_buffer_no = buffer_no
                chosen_buffer_loc = buffer_loc
                print("Least delicate item is", least_delicate.id, least_delicate.score, chosen_buffer_no)
```

```
        return least_delicate, chosen_buffer_no, chosen_buffer_loc

    def IsBufferFull(self):
        for buffer_no, (buffer_loc, item) in self.buffer_occupancy.items():
            if item is None:
                return False

        return True

    def PickFromBufferZone(self):
        """
        Event 2: Pick up an item from the buffer zone. If LEARNING_MODE = True, this will be a hardcoded item given in lof_experiment_params.yaml.
                 If LEARNING_MODE = False, this will be the first item that matches the correct properties (is_hard, is_heavy).
        """
        print_message("2 Picking from Buffer Zone")

        print("Buffer:", self.buffer_occupancy)

        self.update_item_for_buffer()
        self.current_item, buffer_no, buffer_loc = self.get_least_delicate_in_buffer()  # set current item
        self.current_item.packing_pose = self.arm.getPackingLoc() # update item's packing pose

        success = self.arm.MoveToCustomPose(buffer_loc[0], buffer_loc[1], BUFFER_ZONE_Z_HEIGHT, willExecute=WILL_EXECUTE)
        print("Moving to buffer zone {} with loc {}, success? {}".format(buffer_no, (buffer_loc[0], buffer_loc[1], BUFFER_ZONE_Z_HEIGHT), success))
        rospy.loginfo("Removed item {} from buffer zone {}".format(self.current_item.id, buffer_no))

        # Lower arm to buffer location
        lowered_z = 0.07 # currently set for buffer table height
        print("Custom offset is {}".format(self.custom_offset))
        success = self.arm.MoveToCustomPose(buffer_loc[0], buffer_loc[1] + self.custom_offset, lowered_z, willExecute=WILL_EXECUTE)

        if GRIPPER:
            self.CloseGripperParallel()

        # Raise arm after grasping item from buffer
        success = self.arm.MoveToCustomPose(buffer_loc[0], buffer_loc[1], BUFFER_ZONE_Z_HEIGHT, willExecute=WILL_EXECUTE)

        # SUBGOAL
        self.picked_from_buffer = True if success else False
        self.buffer_occupancy[buffer_no][1] = None  # clear buffer memory; area is now unoccupied

        occupied = self.buffer_occupancy.values()
        self.item_in_buffer = False if occupied.count(None) == len(occupied) else True

        # note: hardcoded fix for buffer zone 4 - it is furthest from the robot, so the motion planning
        # gets stuck sometimes. fix is to have the robot move to an intermediate pose first.
        if buffer_no == 4:
            success = self.arm.MoveToCustomPose(-.4, -.5, BUFFER_ZONE_Z_HEIGHT, willExecute=WILL_EXECUTE)

    def PlaceInBox(self):
        """
        Event 3: Place the item in an appropriate location inside the packing box/bin. Afterwards, the robot will move to the neutral position.
        """
```

```
        # Go to center of bin before packing
        self.arm.MoveToJointGoal(BIN_JOINT_GOAL, self.arm.arm_group, willExecute=WILL_EXECUTE)

        if EXPERIMENT_MODE == BASELINE:
            # Baseline: pack items in center of bin regardless of properties
            self.MoveToBoxCenter()

            # open gripper
            if GRIPPER:
                self.OpenGripperFull()

        else:
            packing_z_buffer = .07
            packing_pose = self.current_item.packing_pose

            # move to z buffer above packing location
            packing_pose.position.z += packing_z_buffer
            success = self.arm.MoveToPoseGoal(packing_pose, self.arm.arm_group, willExecute=WILL_EXECUTE)

            packing_pose.position.z -= packing_z_buffer

            # # Lower arm to the packing bin location
            success = self.arm.MoveToPoseGoal(packing_pose, self.arm.arm_group, event_name="Place in box", willExecute=WILL_EXECUTE)
            print_message("3 Placing in Box at x: {} y: {} z: {}".format(round(packing_pose.position.x, 2), round(packing_pose.position.y, 2), round(packing_pose.position.z, 2)))
            print("Successfully moved to packing spot?", success)

            # open gripper and place item in packing bin
            if GRIPPER:
                self.OpenGripper()

            # Raise arm above packing location to avoid moving other items
            alt_packing_pose = self.arm.arm_group.get_current_pose("ee_link").pose
            alt_packing_pose.position.z += packing_z_buffer
            success = self.arm.MoveToPoseGoal(alt_packing_pose, self.arm.arm_group, willExecute=WILL_EXECUTE)


        # SUBGOAL
        self.put_item_in_bin = True
        self.add_packed_item(self.current_item)

        packed_str = ""
        for p in self.packed_items:
            packed_str += "id: {}, area: {}*{}, score: {}, loc: ({}, {}, {})\n".format(p.id, p.width, p.length, p.score, p.packing_pose.position.x, p.packing_pose.position.y, p.packing_pose.position.z)

        print(packed_str)

        # Go to center of bin; repeated twice since this planning execution is not always accurate - need two tries
        self.arm.MoveToJointGoal(BIN_JOINT_GOAL, self.arm.arm_group, willExecute=WILL_EXECUTE)
        self.arm.MoveToJointGoal(BIN_JOINT_GOAL, self.arm.arm_group, willExecute=WILL_EXECUTE)

    def RaiseArm(self):
        """
        Raise robot arm in z direction
        """
        robot_pose = self.arm.arm_group.get_current_pose("ee_link").pose
        robot_pose.position.z = BUFFER_ZONE_Z_HEIGHT
        self.arm.MoveToPoseGoal(robot_pose, self.arm.arm_group, willExecute=WILL_EXECUTE)

```

```
    def MoveToBoxCenter(self):
        robot_pose = self.arm.arm_group.get_current_pose("ee_link").pose
        robot_pose.position.z = BUFFER_ZONE_Z_HEIGHT - .05
        self.arm.MoveToPoseGoal(robot_pose, self.arm.arm_group, willExecute=WILL_EXECUTE)

    def publish_slip_val(self, val):
        slip_msg = Int32()
        slip_msg.data = val
        print("Publishing slip value of {}".format(val))
        self.slip_pub.publish(slip_msg)

    ## GRIPPER FUNCTIONS ##

    def check_gripper_success(self):
        gripper_msg = rospy.wait_for_message("/gripper_finished", Int8)

        if gripper_msg.data == 0:
            print("Robot missed item. Canceling operation...")
            self.missed = True
        else:
            self.missed = False


    def OpenGripper(self):
        print("Publishing gripper open")
        option = Int8()
        option.data = 1
        self.gripper_pub.publish(option)

        rospy.wait_for_message("/gripper_finished", Int8)
        self.missed = False

    def CloseGripper(self):
        print("Publishing gripper close")
        option = Int8()
        option.data = 2
        self.gripper_pub.publish(option)

        self.check_gripper_success()

        print("Exited while loop")

    def OpenGripperFull(self):
        print("Publishing gripper open fully")
        option = Int8()
        option.data = 3
        self.gripper_pub.publish(option)

        rospy.wait_for_message("/gripper_finished", Int8)
        self.missed = False

    def CloseGripperParallel(self):
        print("Publishing gripper close")
        option = Int8()
        option.data = 4
        self.gripper_pub.publish(option)

        self.check_gripper_success()

    def OpenGripperParallel(self):
        print("Publishing gripper open")
```

```
        option = Int8()
        option.data = 5
        self.gripper_pub.publish(option)

        rospy.wait_for_message("/gripper_finished", Int8)
        self.missed = False

    def FingerClose(self):
        print("Publishing aux close")
        option = Int8()
        option.data = 6
        self.gripper_pub.publish(option)

        try:
            rospy.wait_for_message("/gripper_finished", Int8, 2.0)
        except:
            pass

    def FingerCloseExtra(self):
        print("Publishing aux close extra")
        option = Int8()
        option.data = 7
        self.gripper_pub.publish(option)
        try:
            rospy.wait_for_message("/gripper_finished", Int8, 2.0)
        except:
            print("Took too long to receive ack")

class Item():
    """
    Item class: can be formed once the properties are known. When the 'pick from conveyor' is signaled,
    send the vision-related information & tactile information to this class, which will create a new Item object.
    """
    def __init__(self, id, size, packing_pose):
        self.id = id
        self.width = size[0]
        self.length = size[1]
        self.packing_pose = packing_pose
        self.score = 0
        self.tactile_out = 0
        self.proprio_width = None
        self.setScore()

    def __repr__(self):
        return "Item " + str(self.id)

    def setScore(self):
        if EXPERIMENT_MODE == BASELINE: # score is irrelevant
            self.score = 0

        elif EXPERIMENT_MODE == VISION:
            self.score = self.width * self.length
            print("Setting score for item {} to: {}".format(self.id, self.score))

        elif EXPERIMENT_MODE == MULTIMODAL:
            final_width = self.width

            print("Final width is {}, original: {}, proprio: {}".format(final_width, self.width, self.proprio_width))
            area = final_width * self.length

            self.score = -(SLOPE * area + Y_INTERCEPT - self.tactile_out)
```

```python

            print("Setting score for item {} to: {}".format(self.id, self.score))

    def isDelicate(self):
        if EXPERIMENT_MODE == BASELINE: # control group, item is never considered delicate.
            return False

        elif EXPERIMENT_MODE == VISION:
            rospy.loginfo("Area of current item is {}".format(self.score))
            if self.score < AREA_THRESHOLD:
                return True

        elif EXPERIMENT_MODE == MULTIMODAL:
            if CONTACT_THRESHOLD - self.tactile_out > 0:  # if > 0, pack in bin
                print("under low threshold")
                return False

            if self.score < 0:
                print("is delicate")
                return True

        return False

def print_message(message):
    print(message)
    rospy.loginfo(message)

def main(args):
    rospy.init_node('icra_demo', anonymous=True)
    g = GroceryPacking()
    time.sleep(0.5)

    try:
        print('##### press any key to start #####')
        input = raw_input()
        while not rospy.is_shutdown():
            override = None
            action = g.get_next_event()
            print("Tentative action is {}".format(action))

            g.run()

            rospy.sleep(.5)


    except KeyboardInterrupt:
        print "grasping_demo Shutting down"


if __name__ == '__main__':
    main(sys.argv)
```

Object detection module

```python
#!/usr/bin/env python2

import rospy
from sensor_msgs.msg import Image, PointCloud2, CameraInfo
from geometry_msgs.msg import Point
from std_msgs.msg import Float32MultiArray
from cv_bridge import CvBridge, CvBridgeError
```

```python
from matplotlib import pyplot as plt
import cv2
import numpy as np
from ros_numpy import point_cloud2 as np_pc2
from centroid_tracker import CentroidTracker


CAMERA_NAME = rospy.get_param("object_detection", "/camera")
RGB_TOPIC = CAMERA_NAME + '/rgb/image_raw'
DEPTH_TOPIC = CAMERA_NAME + '/depth_registered/image_raw' # depth registered image topic; should align with rgb image (but not rectified)
CAM_INFO_TOPIC = CAMERA_NAME + '/depth/camera_info' #camera info for depth
CENTER_TOPIC = '/boxcoord' # to publish object center coordinates to
BBOX_TOPIC = '/bounding_box' # publish entire bounding box
DEPTHPUB_TOPIC = '/object_depth_image' # to publish processed depth image to
CLOUD_TOPIC = '/object_cloud' # to publish object point cloud to
REGULARITY_TOPIC = '/regularity_score' # to publish object area / bounding box area; currently publishes as a point (score, contour_area)
DISPLAY_IMAGES = rospy.get_param("show_obj_detection", False) # when testing, print out rgb-related image or not
DISPLAY_DEPTHS = False # display depth image of object
CLEAN_IMAGE = True # if True, will look for black area and zero out everything outside the black zone
BLACK_BOUNDARY = 95 #25 # v in hsv to determine what will be considered black
OBJ_DETECTION_THRESHOLD = 50
REGULARITY_THRESHOLD = .74
BELT_PIXEL_BOUNDS = (130, 220)

class ObjDetector():

    def __init__(self):
        self.rgb_sub = rospy.Subscriber(RGB_TOPIC, Image, self.rgb_callback)
        self.depth_sub = rospy.Subscriber(DEPTH_TOPIC, Image, self.depth_callback)
        self.center_pub = rospy.Publisher(CENTER_TOPIC, Point, queue_size=10) # publish the center of the object
        self.bbox_pub = rospy.Publisher(BBOX_TOPIC, Float32MultiArray, queue_size=10) # publish entire bounding box
        self.depth_pub = rospy.Publisher(DEPTHPUB_TOPIC, Image, queue_size=10)
        self.cloud_pub = rospy.Publisher(CLOUD_TOPIC,PointCloud2, queue_size=10) # publish the point cloud of the object
        self.regularity_pub = rospy.Publisher(REGULARITY_TOPIC, Point, queue_size=10)
        self.bridge = CvBridge() # converts between ROS and CV image
        self.bounding_box = ((0,0),(479,639)) # initiallize bounding box as the entire image (image is of shape (480,640))
        self.obj_depths = np.zeros((480, 640)) # dummy first depth reading

        self.camera_info_sub = rospy.Subscriber(CAM_INFO_TOPIC, CameraInfo, self.camera_callback)
        self.camera_info_pub = rospy.Publisher(CAMERA_NAME + '_info', CameraInfo, queue_size=10)
        self.K = np.zeros((3,3)) # calibration matrix for depthTo3d later; this may cause the first (couple?) point clouds to be incorrect, though
        self.ct = ct = CentroidTracker()
        self.center = (0,0)

    def image_print(self,img):
        """
        Helper function to print out images, for debugging. Pass them in as a list.
        Press any key to continue.
        """
        cv2.imshow("image", img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    def camera_callback(self, caminfo):
        self.camera_info_pub.publish(caminfo) # for rviz image visualization
        self.K = np.matrix(caminfo.K).reshape((3,3)) # resetting every time, but all relevant info should stay the same

    def rgb_callback(self,ros_image_msg):
```

```python
        # Convert from ROS image to OpenCV image
        try:
            cv_image = self.bridge.imgmsg_to_cv2(ros_image_msg, "bgr8")
        except CvBridgeError as e:
            rospy.loginfo(e)

        self.find_obj(cv_image)
        point = Point()  # create Point object to publish

        # If the object is not where we expect it to be, publish an empty point (based on y-value)
        if self.center[1] > BELT_PIXEL_BOUNDS[1] or self.center[1] < BELT_PIXEL_BOUNDS[0]:
            self.center = (0,0)
            self.center_pub.publish(point)
        else:
            # OpenCV gives column, row; convert to row, column
            point.x = self.center[0]
            point.y = self.center[1]
            point.z = self.obj_depths[self.center[1], self.center[0]]
            print("Center point", point.x, point.y, point.z)
            self.center_pub.publish(point)

    def depth_callback(self, ros_image_msg):
        # Need to figure out how to deal with depth data
        try:
            depth_image = self.bridge.imgmsg_to_cv2(ros_image_msg,"passthrough")
        except CvBridgeError as e:
            rospy.loginfo(e)
            return

        # Editing depth image clear out everything outside of bounding box
        p1, p2 = self.bounding_box
        obj_depths = depth_image.copy()
        obj_depths[:p1[1],:] = 0
        obj_depths[:,:p1[0]] = 0
        obj_depths[p2[1]:,:] = 0
        obj_depths[:,p2[0]:] = 0
        self.obj_depths = obj_depths

        # Generate mask for point cloud
        pc_mask = np.ones(depth_image.shape)
        pc_mask[:p1[1],:] = 0
        pc_mask[:,:p1[0]] = 0
        pc_mask[p2[1]:,:] = 0
        pc_mask[:,p2[0]:] = 0
        pc_array = cv2.rgbd.depthTo3d(depth_image, self.K, mask=pc_mask)

        # Splice into x's, y's and z's to put back together as numpy record array (each has shape (480,640))
        pc_x = pc_array[:,:,0]
        pc_y = pc_array[:,:,1]
        pc_z = pc_array[:,:,2]
        pc_recarray = np.core.records.fromarrays([pc_x,pc_y,pc_z],names='x,y,z')
        point_cloud = np_pc2.array_to_pointcloud2(pc_recarray,stamp=ros_image_msg.header.stamp,frame_id=ros_image_msg.header.frame_id)

        # Publish object point cloud
        self.cloud_pub.publish(point_cloud)

        # Publish object depth image
        depth_msg = self.bridge.cv2_to_imgmsg(obj_depths,encoding="passthrough")
        depth_msg.header.frame_id = ros_image_msg.header.frame_id
        depth_msg.header.stamp = ros_image_msg.header.stamp
```

```
        self.depth_pub.publish(depth_msg)

        if DISPLAY_DEPTHS:
            # convert to readable np.uint8 type grayscale to use cv2 to visualize as image
            obj_depths = np.array(obj_depths, dtype = np.uint8)
            temp = self.bridge.cv2_to_imgmsg(obj_depths, encoding="mono8")
            obj_depths = self.bridge.imgmsg_to_cv2(temp, "mono8")
            self.image_print(img)

    def find_obj(self, img):
        """
        Segment out the largest black area in the image, then find object against a black background.
        Input:
            img: np.3darray; the input image with a black area and object in the black area to be detected. BGR.
        Return:
            bounding_box: ((x1, y1), (x2, y2)); the bounding box of the object, unit in px
                (x1, y1) is the bottom left of the bbox and (x2, y2) is the top right of the bbox
            center: (x, y); the center of the bounding box

        Info: Tuned hardcoded black values (HSV): [0,0,80] [179,255,255]
                Pick colors here http://colorizer.org/
        """
        # Define range of non-black color in HSV
        lower = np.array([0,0,BLACK_BOUNDARY])
        upper = np.array([179,255,255])

        # Convert color space
        hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV)  # convert from BGR to HSV
        image = hsv
        kernel = np.ones((10,10),np.uint8)  # for mask processing

        if CLEAN_IMAGE:  # look for largest black area, then zero out surrondings
            # We want to look for black from [0,0,0] to lower bound on range
            clean_kernel = np.ones((2,2),np.uint8)
            # clean_mask = cv2.inRange(image,np.array([0,0,0]),np.array([179,130,BLACK_BOUNDARY]))
            clean_mask = cv2.inRange(image,np.array([0,0,0]),np.array([255,130,BLACK_BOUNDARY]))
            # clean_dilated = cv2.dilate(clean_mask,clean_kernel,iterations = 1)  # want to dilate because we're looking to be tolerant with getting the biggest black area
            clean_closing = cv2.morphologyEx(clean_mask,cv2.MORPH_OPEN, clean_kernel)
            # clean_masked = cv2.bitwise_and(image,image,mask=clean_closing) #apply mask to create image of only black pixels); for visualization
            clean_result, clean_contours, clean_hierarchy = cv2.findContours(clean_closing, cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)

            # Find rectangle
            clean_contr = None
            if len(clean_contours) >= 1: #use largest black object
                clean_contr = clean_contours[0]
                if len(clean_contours) > 1:
                    for c in clean_contours:
                        if cv2.contourArea(c) > cv2.contourArea(clean_contr):
                            clean_contr = c

            clean_x1,clean_x2,clean_w,clean_h = cv2.boundingRect(clean_contr) #not angled
            rect = cv2.minAreaRect(clean_contr)
            test = cv2.inRange(image,np.array([0,0,0]), np.array([0,0,0]))  # this is a stupid hack, but np.zeros(image, dtype=np.uint8 was throwing errors)
            cv2.drawContours(test, [np.int0(cv2.boxPoints(rect))], 0 , (255), -1)
            image = cv2.bitwise_and(image, image, mask=test)

            # Display area to keep
```

```
            if DISPLAY_IMAGES:
                self.image_print(cv2.rectangle(img,(clean_x1,clean_x2),(clean_x1+clean_w,clean_x2+clean_h),(0,100,255),2))
                self.image_print(image)

        mask = cv2.inRange(image,lower + np.array([0,0,20]),upper)  # create a mask by thresholding for only non-black values
        closing = cv2.morphologyEx(mask,cv2.MORPH_CLOSE,kernel)  # dilation, then erosion--worked best to smooth out the relevanat (object) areas

        # Find and draw contours of non-black areas
        result, contours, hierarchy = cv2.findContours(closing, cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
        todraw = cv2.drawContours(result,contours,-1,(0,255,0),3)

        closest_item_contr = None
        closest_center = (0,0)
        closest_bounding_box = ((0,0),(0,0))
        closest_item_dim = (0,0)
        original_corner = (0,0)

        for c in contours[:-1]:
            x, y, w, h = cv2.boundingRect(c)
            bounding_box = ((x,y),(x+w, y+h))

            center = (x + w/2, y + h/2)

            if center[0] > closest_center[0] and center[0] < 450 and BELT_PIXEL_BOUNDS[0] < center[1] < BELT_PIXEL_BOUNDS[1]:
                if w < 10 or h < 10 or w > 150:
                    # self.center = (0,0)
                    continue
                closest_item_contr = c
                closest_center = center
                closest_bounding_box = bounding_box
                closest_item_dim = (w, h)
                original_corner = (x, y)

            cv2.rectangle(img,bounding_box[0],bounding_box[1],(100,255,0),2) # draw rectangles around all detected objects

            area = cv2.contourArea(c)

            print("Regularity score is {} for item at {}".format(area/(w*h), center))
            print("Contour has center {} and area {}".format(center,area))

        if closest_item_contr is None: # don't continue track if robot is occluding conveyor belt
            self.center = (0,0)
            return

        self.bounding_box = closest_bounding_box
        old_center = self.center
        self.center = closest_center
        max_area = cv2.contourArea(closest_item_contr)

        box_area = closest_item_dim[0] * closest_item_dim[1]

        regularity_score = max_area / box_area

        rospy.loginfo("CENTER: {}, BOUNDING BOX: {}, ITEM_CONTOUR_AREA: {}, BOX_AREA: {}, REGULARITY: {}".format(self.center, self.bounding_box, max_area, box_area, regularity_score))

        self.track()

        # for detecting when there's no object
        if box_area > 7500:
```

```
            print("too large, resetting center")
            self.center = (0, 0)


        x, y = original_corner
        w, h = closest_item_dim
        corners = [x, y, np.clip(x+w, 0, 639), np.clip(y+h, 0, 479)]
        bbox = Float32MultiArray()
        bbox.data = [ corners[0], corners[1], corners[2], corners[3],
                self.obj_depths[corners[1], corners[0]],
                self.obj_depths[corners[3], corners[0]],
                self.obj_depths[corners[1], corners[2]],
                self.obj_depths[corners[3], corners[2]]]

        self.bbox_pub.publish(bbox)

        regularity_msg = Point()
        regularity_msg.x = regularity_score
        regularity_msg.y = max_area

        self.regularity_pub.publish(regularity_msg)


        if DISPLAY_IMAGES:
            result = cv2.rectangle(img,self.bounding_box[0],self.bounding_box[1],(100,255,0),2)

            self.image_print(result)
            self.image_print(closing)

        # Return bounding box, center of box
        return self.bounding_box, self.center

    def track(self):
        objects = self.ct.update([self.bounding_box])
        for obj_id, obj in objects.items():
            self.centroid = obj  # center found through tracking algorithm


if __name__ == '__main__':
    rospy.init_node('obj_detector')
    obj_detector = ObjDetector()
    rospy.spin()
```

Packing module

```
#!/usr/bin/env python2

import rospy
from sensor_msgs.msg import Image
from geometry_msgs.msg import Point, Pose, Quaternion
from std_msgs.msg import Float32MultiArray
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker

from cv_bridge import CvBridge, CvBridgeError
import cv2
from math import pi
from matplotlib import pyplot as plt
import numpy as np

CAMERA_NAME = rospy.get_param('packing', '/camera')
```

```python
START_TOPIC = CAMERA_NAME + '/rgb/image_raw'
DEPTH_TOPIC = CAMERA_NAME + "/depth_registered/image_raw"
PACK_LOC_TOPIC = "/packing_location"
OBJECTBOX_TOPIC = "/bounding_box"
DISPLAY_IMAGES = rospy.get_param("show_packing")
DISPLAY_DEPTHS = False
DEPTH_KERNEL = (40,40)

DEPTH_TESTING = False
DIST2PIXEL_FACTOR_WIDTH = 426.78
DIST2PIXEL_FACTOR_LENGTH = 445.93
ROTATION_EPSILON = 15  # if length & width are similar enough, no need to rotate
EXPERIMENT_MODE = rospy.get_param("mode")
CALIBRATE = rospy.get_param("calibrate")


class Packing():
    def __init__(self):
        self.start_sub = rospy.Subscriber(START_TOPIC, Image, self.packing_callback)
        self.pack_loc_pub = rospy.Publisher(PACK_LOC_TOPIC, Pose, queue_size=10)
        self.depth_sub = rospy.Subscriber(DEPTH_TOPIC, Image, self.depth_callback)
        self.debug_pub = rospy.Publisher("debug_depth", Image, queue_size=10)
        self.size_sub = rospy.Subscriber("/item_dim", Point, self.size_cb)
        self.calibration_pub = rospy.Publisher("/box_corners", Float32MultiArray, queue_size=10)

        self.bridge = CvBridge()  # Converts between ROS and CV image
        self.pack_loc = (0,0)
        self.mask = np.ones((480,640),np.uint8)
        self.count = 0
        self.obj_depths = np.zeros((480,640))
        self.depth = None
        self.depth_kernel = DEPTH_KERNEL
        self.written = 0
        self.width = 0  # Initial packing location is (0,0)
        self.length = 0
        self.rotated = False


    def size_cb(self, size):
        self.width = size.x * DIST2PIXEL_FACTOR_WIDTH
        self.length = size.y * DIST2PIXEL_FACTOR_LENGTH

    def image_print(self,img):
        """
        Helper function to print out images, for debugging. Pass them in as a list.
        Press any key to continue.
        """
        cv2.imshow("image", img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()


    def packing_callback(self,ros_image_msg):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(ros_image_msg,"bgr8")
        except CvBridgeError as e:
            rospy.loginfo(e)
            return

        # Do cumulative check
        if self.count <= 50:
```

```
            debug_msg = self.find_obj(cv_image)
        else:
            return


    def depth_callback(self,ros_image_msg):
        # Depth image units in mm
        try:
            depth_image = self.bridge.imgmsg_to_cv2(ros_image_msg,"passthrough")
        except CvBridgeError as e:
            rospy.loginfo(e)
            return

        # Set kernel dynamically to object size
        self.update_kernel()

        # Editing depth image clear out everything outside of bounding box
        obj_depths = depth_image.copy()

        nans = np.isnan(obj_depths)
        nans = nans.astype(int)
        nans = nans * 255

        obj_depths = np.nan_to_num(obj_depths)

        try:
            obj_depths = cv2.bitwise_and(obj_depths, obj_depths, mask=self.mask)
        except Exception as e:
            print("failed", e)

        # Check if box is full
        depths_copy = np.nan_to_num(depth_image.copy())
        depth_mask = self.mask.copy()
        obj_depths_copy = cv2.bitwise_and(depths_copy, depths_copy, mask=depth_mask)
        depth_mask[depth_mask == 0] = 1
        depth_mask[obj_depths_copy == np.nan] = 1
        depth_mask[obj_depths_copy == 0] = 1
        depth_mask[depth_mask == 255] = 0
        mx = np.ma.masked_array(obj_depths_copy, mask=depth_mask)

        if not CALIBRATE and (mx.min() < 1.08 or mx.mean() < 1.17):
                print("Box is full. Replace now! Press to continue.")
                inp = raw_input()

        self.obj_depths = obj_depths

        # Smooth to be less sensitive to outliers
        smoothed = cv2.blur(obj_depths,(1,1)) #issues here?

        # build "heatmaps"
        kernel_default = np.ones(self.depth_kernel)
        kernel_default = kernel_default/np.sum(kernel_default)
        convolved_default = cv2.filter2D(smoothed,-1, kernel_default) #-1 means keep the same data type as source
        kernel_rotated = np.ones((self.depth_kernel[1],self.depth_kernel[0]))
        kernel_rotated = kernel_rotated/np.sum(kernel_rotated)
        convolved_rotated = cv2.filter2D(smoothed,-1, kernel_rotated) #-1 means keep the same data type as source

        # locate optimal packing location
        # gripper_safety = 35 # increasing value widens the kernel --> takes into account the space taken by the fingers
        gripper_safety = 0
        if np.amax(convolved_default) > np.amax(convolved_rotated):
```

```
    pack_loc = np.unravel_index(np.argmax(convolved_default),convolved_default.shape)
    depth_roi = obj_depths[pack_loc[0] - self.depth_kernel[0]/2: pack_loc[0] + self.depth_kernel[0]/2, pack_loc[1] - (self.depth_kernel[1]/2 + gripper_safety): pack_loc[1] + self.depth_kernel[1]/2+gripper_safety]
    self.rotated = False
else:
    pack_loc = np.unravel_index(np.argmax(convolved_rotated),convolved_default.shape)
    depth_roi = obj_depths[pack_loc[0] - (self.depth_kernel[1]/2 + gripper_safety): pack_loc[0] + self.depth_kernel[1]/2 + gripper_safety, pack_loc[1] - self.depth_kernel[0]/2: pack_loc[1] + self.depth_kernel[0]/2]
    self.rotated = True

x_start, x_end = 284, 341
y_start, y_end = 186, 242

x_start, x_end = 0, 638
y_start, y_end = 0, 478

if CALIBRATE:
    try:
        test_depths = np.zeros((y_end - y_start, x_end-x_start))
        min_x, min_y = float('inf'), float('inf')
        max_x, max_y = float('-inf'), float('-inf')
        for r in range(y_start, y_end):
            for c in range(x_start, x_end):
                if 1 < convolved_default[r][c] < 1.3:
                    test_depths[r - y_start][c - x_start] = convolved_default[r][c]

                    min_x = min(c, min_x)
                    min_y = min(r, min_y)

                    max_x = max(c, max_x)
                    max_y = max(r, max_y)

        calibration_msg = Float32MultiArray()
        calibration_msg.data = [min_y, min_x, self.obj_depths[min_y, min_x],
            min_y, max_x, self.obj_depths[min_y, max_x],
            max_y, max_x, self.obj_depths[max_y, max_x],
            max_y, min_x, self.obj_depths[max_y, min_x]]

        self.calibration_pub.publish(calibration_msg)
    except:
        pass


self.pack_loc = pack_loc
self.depth = max(1.09, np.min(depth_roi)) # take most conservative estimate of highest point


if DISPLAY_DEPTHS:
    test = depth_image.copy()
    test[test<0] = np.amax(test)
    normed = test / np.sum(test)
    self.image_print(normed)
    rospy.loginfo(np.max(obj_depths))
    rospy.loginfo(np.min(obj_depths))

# Publish packing location
packing_pose = Pose()
packing_pose.position.x = self.pack_loc[0]
packing_pose.position.y = self.pack_loc[1]
packing_pose.position.z = self.depth# self.obj_depths[self.pack_loc]

```

```
        if EXPERIMENT_MODE == 1: # don't rotate if control group
            self.rotated = False

        if self.rotated:
            packing_pose.orientation = Quaternion(*quaternion_from_euler(pi/2, pi/2, -pi/2, 'rxyz'))
        else:
            packing_pose.orientation = Quaternion(*quaternion_from_euler(-pi/2, 0, pi/2, 'rxyz'))

        self.pack_loc_pub.publish(packing_pose)

        if DEPTH_TESTING:
            ## DEPTH TESTING JEANA ##
            corner_pixels = rospy.get_param("corner_pixels")

            if self.written < 5:
                print("PACKING PIXELS", packing_pose.x, packing_pose.y, packing_pose.z)
                self.written += 1
                to_plot = []
                for r in range(len(self.obj_depths)):
                    for c in range(len(self.obj_depths[0])):
                        if self.obj_depths[r,c] > 0:
                            to_plot.append((r, c, float(self.obj_depths[r,c])))

                rospy.set_param("to_plot", to_plot)
                corner_depths = []
                for pixel in corner_pixels:
                    corner_depths.append(float(self.obj_depths[pixel[0], pixel[1]]))
                rospy.set_param("corner_depths", corner_depths)
            ## END DEPTH TESTING ##


    def find_obj(self, img):
        """
        Load mask and segment out packing box, then find lowest point inside the box.

        Input:
            img: np.3darray; the input image with a black area and object in the black area to be detected. BGR.
        Return:
            bounding_box: ((x1, y1), (x2, y2)); the bounding box of the object, unit in px
                (x1, y1) is the bottom left of the bbox and (x2, y2) is the top right of the bbox
            center: (x, y); the center of the bounding box


        Info: Tuned hardcoded black values (HSV): [0,0,80] [179,255,255]
              Pick colors here http://colorizer.org/
        """
        self.count += 1

        if self.count == 1:
            self.read_mask()

        image = cv2.cvtColor(img,cv2.COLOR_BGR2HSV)

        if self.count == 5: # give depth side some time to find a packing loc
            rospy.loginfo("This is the calibrated masked image with coord " + str(self.pack_loc) + " and depth " + str(self.depth))
            todraw = cv2.bitwise_and(image, image, mask=self.mask)
            todraw = cv2.circle(todraw,(self.pack_loc[1], self.pack_loc[0]),10,(100,200,255),-1)
            self.image_print(todraw)

```

```
        if DISPLAY_IMAGES:
            print("displaying image")
            todraw = cv2.bitwise_and(image, image, mask=self.original_mask)
            todraw = cv2.circle(todraw,(self.pack_loc[1], self.pack_loc[0]),10,(100,200,255),-1) # opencv's indices are column, row - opposite from np
            if self.rotated:
                todraw = cv2.rectangle(todraw, (self.pack_loc[1] - self.depth_kernel[0]//2, self.pack_loc[0] - self.depth_kernel[1]//2),
                        (self.pack_loc[1] + self.depth_kernel[0]//2, self.pack_loc[0] + self.depth_kernel[1]//2),
                        (100,200,255), 2)
            else:
                todraw = cv2.rectangle(todraw, (self.pack_loc[1] - self.depth_kernel[1]//2, self.pack_loc[0] - self.depth_kernel[0]//2),
                        (self.pack_loc[1] + self.depth_kernel[1]//2, self.pack_loc[0] + self.depth_kernel[0]//2),
                        (100,200,255), 2)
            print("Rotated", self.rotated)
            self.image_print(todraw)

        return

    def update_kernel(self):
        # note: packing and object detection cameras are rotated pi/2 relative to each other
        if np.isnan(self.width) or np.isnan(self.length):
            self.depth_kernel = DEPTH_KERNEL
            return

        pixel_width = int(self.width + .5)
        pixel_length = int(self.length + .5)

        if (pixel_width == 0 or pixel_length == 0):
            return

        # minimum kernel size
        if pixel_length * pixel_width < 400:
            pixel_length = max(pixel_length,40)
            pixel_width = max(pixel_width,40)
        if self.depth_kernel != (pixel_length, pixel_width):
            self.depth_kernel = (pixel_length, pixel_width)


    def read_mask(self):
        self.mask = cv2.imread("/home/ada/manipulation_ws/src/vkchen_vision/calibration/packingBoxMask.png", cv2.IMREAD_GRAYSCALE)
        print("Mask shape is {}, should be (480, 640)".format(self.mask.shape))

        self.original_mask = cv2.imread("/home/ada/manipulation_ws/src/vkchen_vision/calibration/originalPackingBoxMask.png", cv2.IMREAD_GRAYSCALE)

    def set_count(self, count):
        self.count = count

    def get_count(self):
        return self.count


if __name__ == '__main__':
    rospy.init_node('packing')
    packing = Packing()

    try:
        while not rospy.is_shutdown():
```

```
            pass
        except KeyboardInterrupt:
            print "packing node shutting down"

        rospy.spin()
```

Dynamixel Servo Control

```
#!/usr/bin/env python

import numpy as np
import os
import rospy
from std_msgs.msg import Int8, Float32MultiArray, Int32, Float32
import struct
from collections import namedtuple
from dynamixel_sdk import PortHandler, PacketHandler, COMM_SUCCESS, GroupSyncWrite, GroupSyncRead
from aux_gripper.DynamixelGripper import *


"""
https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_wizard2/#usb-latency-setting
https://learn.trossenrobotics.com/projects/194-setting-dynamixel-ax-and-mx-series-firmware-id-and-baud-with-roboplus
    -1-0.html

Change latency of USB port reading from Linux computer:
$ echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
$ cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
"""

SERVO_EPSILON = 50 # TODO: tune this // set to 1000 before
GOAL_VELOCITY_VALUE = 200
OPEN_VELOCITY = 125
VELOCITY_EPSILON = 15
LOAD_DELTA_THRESHOLD = 17
CONSECUTIVE_THRESH =  12
TRACK_LOAD_THRESH = 35
PARTIAL_OPEN_OFFSET = 1800

# GRIPPER ACTION MAPPINGS
## naming convention: [partial/full]_[open/close]_[which servos]
PARTIAL_OPEN = 1
CLOSE_ALL = 2
FULL_OPEN = 3
CLOSE_PARALLEL = 4
OPEN_PARALLEL = 5
CLOSE_AUX = 6
CLOSE_AUX_CUSTOM = 7


class GripperControl():

    def __init__(self):
        self.gripper = DynamixelGripper()

        # Publishers & Subscribers
        self.gripper_finished_pub = rospy.Publisher("/gripper_finished", Int8, queue_size=0)
        self.gripper_sub = rospy.Subscriber("/move_gripper", Int8, self.gripper_cb)
        self.slip_sub = rospy.Subscriber("/slip_amt", Int32, self.slip_cb)
        self.gripper_command_sub = rospy.Subscriber("/gripper_command", Int8, self.pos_command_cb)
        self.distance_pub = rospy.Publisher("/gripper_width", Float32, queue_size=0)
```

```
        self.torque_sub = rospy.Subscriber("/torque_on", Int8, self.torque_cb)

        self.option = None
        self.prev_load_33 = None
        self.command = False
        self.count = 0
        self.partial_open_count = 0
        self.prev_option = -1
        self.prev_prev_option = -1

        self.open_limit = self.gripper.track_servo.open
        self.close_limit = self.gripper.track_servo.close

        self.current_close = None


    def OpenGripper(self, pos=None):
        print("gripper opening!")
        rospy.loginfo("Opening gripper")
        # self.MoveToOpenPosition(pos)
        self.gripper.auxOpen() # TODO: add aux control back in
        self.gripper.setVelocity(-OPEN_VELOCITY)

    def OpenGripperParallel(self):
        print("gripper opening!")
        rospy.loginfo("Opening gripper")
        self.gripper.setVelocity(-OPEN_VELOCITY)

    def CloseGripper(self):
        print("gripper closing!")
        rospy.loginfo("Closing gripper")
        # self.gripper.auxClose()
        self.gripper.auxCloseCustom(400) # TODO: add aux control back in
        self.gripper.setVelocity(GOAL_VELOCITY_VALUE)

    def CloseGripperParallel(self):
        print("gripper closing!")
        rospy.loginfo("Closing gripper")
        self.gripper.setVelocity(GOAL_VELOCITY_VALUE)

    def slip_cb(self, slip_amt):
        new_val = slip_amt.data
        self.open_limit += new_val
        self.close_limit += new_val
        print("Slip detected. Adjusting with new interval [{}, {}]".format(self.open_limit, self.close_limit))

    def torque_cb(self, data):
        torque_on = data.data

        if torque_on: # enable torque
            self.gripper.enable_torque()

        else: # disable torque
            self.gripper.poe_exit_program()

    def gripper_cb(self, move_option):
        if move_option.data == PARTIAL_OPEN:
            print("received gripper option 1: open gripper partially")
            self.option = PARTIAL_OPEN

        elif move_option.data == CLOSE_ALL:
```

```python
            print("received gripper option 2: close gripper")
            self.option = CLOSE_ALL

        elif move_option.data == FULL_OPEN:
            print("received gripper option 3: open gripper fully")
            self.option = FULL_OPEN

        elif move_option.data == CLOSE_PARALLEL:
            print("received gripper option 4: close gripper parallel")
            self.option = CLOSE_PARALLEL

        elif move_option.data == OPEN_PARALLEL:
            print("received gripper option 5: open gripper parallel")
            self.option = OPEN_PARALLEL

        elif move_option.data == CLOSE_AUX:
            self.option = CLOSE_AUX
            print("received gripper option 6: aux close")

        elif move_option.data == CLOSE_AUX_CUSTOM:
            self.option = CLOSE_AUX_CUSTOM
            print("received gripper option 7: aux close custom")

        else:
            print("Invalid option. Will not move gripper.")
            self.option = None

    def publish_finished_flag(self, val):
        print("Finished gripper action!")
        # publish finished flag
        done_msg = Int8()
        done_msg.data = val
        for i in range(5):
            self.gripper_finished_pub.publish(done_msg)
            rospy.sleep(.2)

    def StopServo(self, message="", pos=None, load=None):
        self.gripper.setVelocity(0)
        self.option = None
        val = 1

        if load is not None and (self.option in (CLOSE_ALL, CLOSE_PARALLEL)):
            if load < 123:
                val = 0
                print("Gripper missed item!")

        if pos is not None:
            dist_msg = Float32()
            ticks = self.gripper.track_servo.close - pos
            cm = ticks * 0.014 / 500.0
            print("Item is {} m wide with {} ticks and close pos {}".format(cm, ticks, self.gripper.track_servo.close))
            dist_msg.data = cm
            self.distance_pub.publish(dist_msg)

        self.publish_finished_flag(val)

        self.command = False
        if len(message) > 0:
            rospy.loginfo(message)
```

```python
    def PositionThresholdReached(self, pos, vel):
        return self.OpenPositionLimitReached(pos, vel) and self.ClosePositionLimitReached(pos, vel)

    def OpenPositionLimitReached(self, pos, vel):
        # stop if gripper opens past position threshold
        if vel < -1 and (pos - SERVO_EPSILON < self.open_limit):
            message = "Gripper opens past position threshold at {}".format(self.open_limit)
            print(message)
            return True
        return False

    def OpenPartialReached(self, pos, vel):
        if vel < -1 and pos <= self.current_close - PARTIAL_OPEN_OFFSET:
            return True
        return False


    def ClosePositionLimitReached(self, pos, vel):
        # stop if gripper closes past position threshold
        if vel > 0 and (pos + SERVO_EPSILON > self.close_limit):
            message = "Full close reached at {}".format(self.close_limit)
            print(message)
            return True

        return False

    def readAndPublishServoData(self):
        ids, loads, vels, pos = self.gripper.readServos()

        printstr = "ids: {}, loads: {}, vels: {}, pos: {}".format(ids, loads, vels, pos)
        gripper_msg = Float32MultiArray()
        gripper_msg.data = [ids[0], loads[0], vels[0], pos[0],
                            ids[1], loads[1], vels[1], pos[1],
                            ids[2], loads[2], vels[2], pos[2]]

        # self.output_pub.publish(gripper_msg)

        return ids, loads, vels, pos


    ### POSITION CONTROL ###

    def pos_command_cb(self, option_data):
        option = option_data.data
        ### Position Control Menu ###
        print("receiving option {}".format(option))

        if option == 0:
            ids, loads, vels, pos = gp.readAndPublishServoData()
            printstr = "ids: {}, loads: {}, vels: {}, pos: {}".format(ids, loads, vels, pos)
            print(printstr)

        elif option == 1:
            print("option 1")
            self.gripper.parallelOpenInch()
        elif option == 2:
            print("option 2")
            self.gripper.parallelClose()
        elif option == 3:
            print("option 3")
            self.gripper.parallelOpen() # to change vals, modify TRACK_SERVO_INCH in DynamixelGripper.py
```

```
        elif option == 4:
            print("option 4")
            self.gripper.parallelCloseInch()

        rospy.sleep(1.) # wait 1 second
        self.publish_finished_flag(1)

PROGRAM_CURRENT_TEST = 0
PROGRAM_JEANA        = 1

program = PROGRAM_JEANA
TRACK_INDEX =  2 # index of track servo; originally 2

def main():
    gp = GripperControl()
    gp.command = False
    count = 0

    if rospy.is_shutdown():
        print("[poe] ros not running; exiting")
        exit()

    ids, loads, vels, pos = gp.readAndPublishServoData()
    print("ids: {}, loads: {}, vels: {}, pos: {}".format(*gp.readAndPublishServoData()))

    while not rospy.is_shutdown():
        if (program == PROGRAM_CURRENT_TEST):
            load = gp.gripper.readServos()[1][0] # fornow (assumes that gripperServo is index 0)
            print("{}".format(load))
        elif (program == PROGRAM_JEANA):
            if gp.command:
                ids, loads, vels, pos = gp.readAndPublishServoData()
                printstr = "ids: {}, loads: {}, vels: {}, pos: {}".format(ids, loads, vels, pos)
                print("LOAD: {}".format(loads[TRACK_INDEX]))

                ### TRACK SERVO CONTROL ###

                if (gp.option in (CLOSE_ALL, CLOSE_PARALLEL) and gp.ClosePositionLimitReached(pos[TRACK_INDEX], vels[TRACK_INDEX])) or \
                    (gp.option in (PARTIAL_OPEN, FULL_OPEN, OPEN_PARALLEL) and gp.OpenPositionLimitReached(pos[TRACK_INDEX], vels[TRACK_INDEX])):
                    load = loads[TRACK_INDEX] if gp.option in (CLOSE_ALL, CLOSE_PARALLEL) else None

                    gp.StopServo(pos=pos[TRACK_INDEX], load=load)
                    gp.command = False
                    print("Open/Close limit reached..")
                    print(printstr)

                elif gp.option == PARTIAL_OPEN:
                    if gp.OpenPartialReached(pos[TRACK_INDEX], vels[TRACK_INDEX]):
                        gp.StopServo()
                        gp.command = False
                        print(printstr)

                # Stop if significant load detected
                elif pos[TRACK_INDEX] > gp.gripper.track_servo.open + 800 and (loads[TRACK_INDEX] > TRACK_LOAD_THRESH) and abs(vels[TRACK_INDEX] - GOAL_VELOCITY_VALUE) < VELOCITY_EPSILON: # finger loads exceeded
                    message = "Load threshold reached - setting velocity to 0."
                    print(message)
                    gp.count += 1

```

```
                    print("Loads: {}, Present velocity is {} and goal velocity is {} with epsilon {}".format(loads, vels
                        [TRACK_INDEX], GOAL_VELOCITY_VALUE, VELOCITY_EPSILON))

                    if gp.count >= CONSECUTIVE_THRESH: #4 times
                        gp.StopServo(message=message, pos=pos[TRACK_INDEX]) #TODO: may need to add , load=loads[
                            TRACK_INDEX]
                        gp.command = False
                        print(printstr)

            else:
                if gp.option in (PARTIAL_OPEN, FULL_OPEN, OPEN_PARALLEL): # open
                    if not gp.OpenPositionLimitReached(pos[TRACK_INDEX], vels[TRACK_INDEX]):
                        if gp.option == OPEN_PARALLEL:
                            print("open gripper parallel")
                            gp.OpenGripperParallel()
                        else:
                            gp.OpenGripper()
                        gp.command = True
                        gp.count = 0
                        gp.partial_open_count = 0
                        gp.current_close = pos[TRACK_INDEX]
                    else:
                        gp.gripper.auxOpen()
                        gp.StopServo()
                        gp.command = False
                        print(printstr)

                elif gp.option == CLOSE_ALL: # close
                    gp.CloseGripper()
                    gp.command = True

                elif gp.option == CLOSE_PARALLEL:
                    gp.CloseGripperParallel()
                    gp.command = True

                elif gp.option == CLOSE_AUX:
                    gp.gripper.auxClose()
                    gp.command = False
                    gp.option == None

                elif gp.option == CLOSE_AUX_CUSTOM and gp.prev_option != CLOSE_AUX_CUSTOM and gp.prev_prev_option !=
                    CLOSE_AUX_CUSTOM:
                    gp.gripper.auxCloseCustom(500)
                    gp.command = False
                    gp.option == None
                    gp.publish_finished_flag(1)

                gp.prev_option = gp.option
                gp.prev_prev_option = gp.prev_option

    gp.gripper.poe_exit_program()

if __name__ == '__main__':
    rospy.init_node('read_servo', anonymous=True)
    main()
```

Bibliography

[1] Alexander C. Abad and Anuradha Ranasinghe. Visuotactile sensors with emphasis on gelsight sensor: A review. *IEEE Sensors Journal*, 20(14):7628–7638, 2020.

[2] Marichi Agarwal, Swagata Biswas, Chayan Sarkar, Sayan Paul, and Himadri Sekhar Paul. Jampacker: An efficient and reliable robotic bin packing system for cuboid objects. *IEEE Robotics and Automation Letters*, 6(2):319–326, 2021.

[3] Yesenia Aquilina and Michael A. Saliba. An automated supermarket checkout system utilizing a scara robot: preliminary prototype development. *Procedia Manufacturing*, 38:1558–1565, 2019. 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland, Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing.

[4] Yasemin Bekiroglu, Renaud Detry, and Danica Kragic. Learning tactile characterizations of object- and pose-specific grasps. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1554–1560, 2011.

[5] Jeannette Bohg, Matthew Johnson-Roberson, Mårten Björkman, and Danica Kragic. Strategies for multi-modal scene exploration. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4509–4515, 2010.

[6] Roberto Calandra, Andrew Owens, Dinesh Jayaraman, Justin Lin, Wenzhen Yuan, Jitendra Malik, Edward H. Adelson, and Sergey Levine. More than a feeling: Learning to grasp and regrasp using vision and touch. *IEEE Robotics and Automation Letters*, 3(4):3300–3307, Oct 2018.

[7] Lillian Chin, Felipe Barscevicius, Jeffrey Lipton, and Daniela Rus. Multiplexed manipulation: Versatile multimodal grasping via a hybrid soft gripper. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8949–8955. IEEE, 2020.

[8] Lillian Chin, Jeffrey Lipton, Robert MacCurdy, John Romanishin, Chetan Sharma, and Daniela Rus. Compliant electric actuators based on handed shearing auxetics. In *2018 IEEE International Conference on Soft Robotics (RoboSoft)*, pages 100–107. IEEE, 2018.

[9] Lillian Chin, Michelle C Yuen, Jeffrey Lipton, Luis H Trueba, Rebecca Kramer-Bottiglio, and Daniela Rus. A simple electric soft robotic gripper with high-deformation haptic feedback. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2765–2771. IEEE, 2019.

[10] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *arXiv preprint arXiv:1404.3785*, 2014.

[11] Michael Danielczuk, Matthew Matl, Saurabh Gupta, Andrew Li, Andrew Lee, Jeffrey Mahler, and Ken Goldberg. Segmenting unknown 3d objects from real depth images using mask r-cnn trained on synthetic data. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7283–7290, 2019.

[12] Daniele De Gregorio, Riccardo Zanella, Gianluca Palli, Salvatore Pirozzi, and Claudio Melchiorri. Integration of robotic vision and tactile sensing for wire-terminal insertion tasks. *IEEE Transactions on Automation Science and Engineering*, 16(2):585–598, 2019.

[13] Yang Gao, Lisa Anne Hendricks, Katherine J Kuchenbecker, and Trevor Darrell. Deep learning for tactile understanding from visual and haptic data. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 536–543. IEEE, 2016.

[14] Alexander M. Gruebele, Michael A. Lin, Dane Brouwer, Shenli Yuan, Andrew C. Zerbe, and Mark R. Cutkosky. A Stretchable Tactile Sleeve for Reaching Into Cluttered Spaces. *IEEE Robotics and Automation Letters*, 6(3):5308–5315, July 2021.

[15] Bianca S Homberg, Robert K Katzschmann, Mehmet R Dogar, and Daniela Rus. Haptic identification of objects using a modular soft robotic gripper. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 1698–1705. IEEE, 2015.

[16] Young-Dae Hong, Young-Joo Kim, and Ki-Baek Lee. Smart Pack: Online Autonomous Object-Packing System Using RGB-D Sensor Data. *Sensors*, 20(16):4448, January 2020.

[17] Ruizhen Hu, Juzhan Xu, Bin Chen, Minglun Gong, Hao Zhang, and Hui Huang. Tap-net. *ACM Transactions on Graphics*, 39(6):1–15, Nov 2020.

[18] Josie Hughes, Shuguang Li, and Daniela Rus. Sensorization of a continuum body gripper for high force and delicate object grasping. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6913–6919, 2020.

[19] Takuya Ikai, Shota Kamiya, and Masahiro Ohka. Robot control using natural instructions via visual and tactile sensations. *Journal of Computer Science*, 12(5):246–254, May. 2016.

[20] Naveen Kuppuswamy, Alex Alspach, Avinash Uttamchandani, Sam Creasey, Takuya Ikeda, and Russ Tedrake. Soft-bubble grippers for robust and perceptive manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9917–9924, October 2020.

[21] Michelle A. Lee, Matthew Tan, Yuke Zhu, and Jeannette Bohg. Detect, reject, correct: Crossmodal compensation of corrupted sensors, 2020.

[22] Michelle A Lee, Brent Yi, Roberto Martín-Martín, Silvio Savarese, and Jeannette Bohg. Multimodal sensor fusion with differentiable filters. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10444–10451. IEEE, 2020.

[23] Michelle A. Lee, Yuke Zhu, Krishnan Srinivasan, Parth Shah, Silvio Savarese, Li Fei-Fei, Animesh Garg, and Jeannette Bohg. Making sense of vision and touch: Self-supervised learning of multimodal representations for contact-rich tasks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8943–8950, 2019.

[24] Shuguang Li, John J. Stampfli, Helen Xu, Elian Malkin, Evelin Villegas Diaz, Daniela Rus, and Robert J. Wood. A vacuum-driven origami "magic-ball" soft gripper. *2019 International Conference on Robotics and Automation (ICRA)*, pages 7401–7408, 2019.

[25] Abhijit Makhal, Federico Thomas, and Alba Perez Gracia. Grasping unknown objects in clutter by superquadric representation. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 292–299, 2018.

[26] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.

[27] Kirstie Renae. 8 tips for bagging groceries, according to someone who does it every day. `insider.com/best-way-to-bag-groceries-2018-12`. Accessed: 2021-09-08.

[28] Rocco Antonio Romeo, Michele Gesino, Marco Maggiali, and Luca Fiorio. Combining Sensors Information to Enhance Pneumatic Grippers Performance. *Sensors*, 21(15):5020, July 2021.

[29] Philipp Schmidt, Nikolaus Vahrenkamp, Mirko Wächter, and Tamim Asfour. Grasping of unknown objects using deep convolutional neural networks based on depth images. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6831–6838, 2018.

[30] Max Schwarz, Anton Milan, Arul Selvam Periyasamy, and Sven Behnke. RGB-D object detection and semantic segmentation for autonomous manipulation in clutter. *The International Journal of Robotics Research*, 37(4-5):437–451, April 2018. Publisher: SAGE Publications Ltd STM.

[31] Benjamin Shih, Dylan Shah, Jinxing Li, Thomas G Thuruthel, Yong-Lae Park, Fumiya Iida, Zhenan Bao, Rebecca Kramer-Bottiglio, and Michael T Tolley. Electronic skins and machine learning for intelligent soft robots. *Science Robotics*, 5(41), 2020.

[32] Jun Shintake, Vito Cacucciolo, Dario Floreano, and Herbert Shea. Soft robotic grippers. *Advanced Materials*, 30(29):1707035, 2018.

[33] Rahul Shome, Wei N. Tang, Changkyu Song, Chaitanya Mitash, Hristiyan Kourtev, Jingjin Yu, Abdeslam Boularias, and Kostas E. Bekris. Towards robust product packing with a minimalistic end-effector. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 9007–9013, 2019.

[34] Benno Staub, Ajay Kumar Tanwani, Jeffrey Mahler, Michel Breyer, Michael Laskey, Yutaka Takaoka, Max Bajracharya, Roland Siegwart, and Ken Goldberg. Dex-net mm: Deep grasping for surface decluttering with a low-precision mobile manipulator. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 1373–1379, 2019.

[35] Ryan L Truby, Lillian Chin, and Daniela Rus. A recipe for electrically-driven soft robots via 3d printed handed shearing auxetics. *IEEE Robotics and Automation Letters*, 6(2):795–802, 2021.

[36] Ryan L. Truby, Robert K. Katzschmann, Jennifer A. Lewis, and Daniela Rus. Soft robotic fingers with embedded ionogel sensors and discrete actuation modes for somatosensitive manipulation. In *2019 2nd IEEE International Conference on Soft Robotics (RoboSoft)*, pages 322–329, 2019.

[37] Fan Wang and Kris Hauser. Stable Bin Packing of Non-convex 3D Objects with a Robot Manipulator. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8698–8704, May 2019.

[38] Fan Wang and Kris Hauser. Robot packing with known items and nondeterministic arrival order. *IEEE Transactions on Automation Science and Engineering*, pages 1–15, 2020.

[39] Hongbo Wang, Massimo Totaro, and Lucia Beccai. Toward Perceptive Soft Robots: Progress and Challenges. *Advanced Science*, 5(9):1800541, 2018.

[40] Akihiko Yamaguchi and Christopher G. Atkeson. Combining finger vision and optical tactile sensing: Reducing and handling errors while cutting vegetables. In *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pages 1045–1051, 2016.

[41] Akiya Yasuda, Gustavo Alfonso Garcia Ricardez, Jun Takamatsu, and Tsukasa Ogasawara. Packing planning and execution considering arrangement rules. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 100–106, 2020.

[42] Kuan-Ting Yu and Alberto Rodriguez. Realtime state estimation with tactile and visual sensing for inserting a suction-held object. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1628–1635. IEEE, 2018.