Vision and Improved Learned-Trajectory Replay for Assistive-Feeding and Food-Plating Robots

Travers Rhodes

CMU-RI-TR-19-55

Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

July 2019

Thesis Committee:
Manuela M. Veloso, Chair
Henny Admoni
Devin Schwab

Abstract

Food manipulation offers an interesting frontier for robotics research because of the direct application of this research to real-world problems and the challenges involved in robust manipulation of deformable food items. In this work, we focus on the challenges associated with robots manipulating food for assistive feeding and meal preparation. This work focuses on how we can teach robots visual perception of the objects to manipulate, create error recovery and feedback systems, and improve on kinesthetic teaching of manipulation trajectories. This work includes several complete implementations of food manipulation robots for feeding and food plating on several robot platforms: a SoftBank Robotics Pepper, a Kinova MICO, a Niryo One, and a UR5.

Acknowledgements

This work would not have been possible without the continued support and inspiration of my advisor, Manuela Veloso, and the support of faculty members at Carnegie Mellon University and at Instituto Superior Técnico in Lisbon, including Henny Admoni, Oliver Kroemer, João Paulo Costeira, and Manuel Marques.

Thank you to all the members of the CORAL lab for practical and motivational help. Thanks also to the members of Oliver Kroemer's lab, including Sam Clarke and Kevin Zhang, and the members of Henny Admoni's lab, including Reuben Aronson and Ben Newman, for putting up with me when I showed up in their labs to use their robot arms. Thanks to the members of my Dissertation Writing Group, Pragna Mannam, Max Sieb, Abhijat Biswas, and Suddhu Suresh. Thanks also to Ceci Morales for assistance building my first robot from scratch. I want to especially thank Alexandre Candeias for his intellectual, engineering, and moral support throughout the FeedBot project.

Thank you to my family and friends for inspiring me to pursue research, including my father, C. Harker Rhodes III, M.D., Ph.D., my uncle Ed Rhodes, Ph.D., my grandmothers Mae Rhodes and Janey Symington, Ph.D., and my college roommate Adrian Veres, Ph.D.

Finally, many thanks to my wife, C. Taylor Poor, J.D., for encouraging and approving of my academic pursuits.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Food manipulation offers an interesting frontier for robotics research because of the direct application of this research to real-world problems and the challenges involved in robust manipulation of deformable food items. In this introduction, we explain how robots are used for assistive feeding and meal preparation, as well what makes it difficult for robots to manipulate food. Finally, we introduce our approach to the problem of robots manipulating food and enumerate the contributions we have made toward the advancement of domestic food manipulation robots.

1.1 Motivation

This work is motivated by two distinct and complementary real-world challenges that have the potential to be solved by robots. These challenges are assistive feeding, where a person with an upper-extremity disabilities is fed by a robot, and food plating, where a robot transfers and arranges prepared food onto a plate.

1.1.1 Assistive-Feeding Robots

The primary real-world challenge that is addressed in this thesis is that of feeding-assistance robots for people living with upper-extremity disabilities. In the United States, about three in every 1000 children have cerebral palsy [1]. Since at least 1963, when James Reswick developed the Case Research Arm Aid—Mark I, researchers have been trying to create robots to assist in everyday activities for people living with disabilities [2]. In 1987, Mike Topping began work on HANDY-1, a robot designed to help people with cerebral palsy at mealtimes, and his product was eventually commercialized [3, 4]. Compared to a caretaker, a feeding robot like HANDY-1 allows the user to have more control over the choice of food for each bite and the pace of the meal, allows for more consistent feeding, and makes the meal more engaging for the user [3].

There are currently a number of commercial robots available which assist in the

Figure 1.1: The author using a custom-built Niryo One feeding robot running algorithms presented in this work

feeding task, including Bestic [5], Meal Buddy [6], Mealtime Partner [7], My Spoon [8], and Obi [9]. For these products, a human agent (generally a caretaker) sets up the robot so that its feeding trajectory reaches the appropriate feeding location for the user. For example, for the Obi, the caretaker kinesthetically shows the robot the target feeding location by holding down a button and then manually moving the spoon to the desired feeding position. The Obi then computes a trajectory from the food to that target location [9].

Our research aims to improve the capabilities of autonomous feeding robots to allow a caretaker to specify the full trajectory of the robot arm rather than just the final position. Specifying the full feeding trajectory could enable a caretaker to expand the capabilities of the feeding robot to perform additional tasks, like combining foods before each bite or dipping each bite in sauce before moving the food to the feeding location. Additionally, by adding better feedback to the system, we hope to improve the robustness of these feeding robots. Current commercial robots do not detect whether the food is successfully acquired before bringing the spoon to the feeding location, and the feeding location does not respond to changes in the position of the user's head during feeding. Our implementation on a custom-built Niryo One robot [10] is shown in Figure 1.1 and our implementation on a Kinova MICO robot [11] is shown in Figure 3.2.

1.1.2 Food-Plating Robots

Another real-world challenge of food manipulation that we explore in this thesis is that of food-plating robots. While automation has long been used in food processing and packaging, and while a range of appliances simplify the food preparation process in homes, the less structured environment of transferring prepared food to a serving dish is still an open research area. In particular, we are interested in using a generic food

manipulation tool like a fork to perform the plating task. We want to use a fork from a research perspective because it encourages complicated manipulation strategies, and from a practical perspective because there are benefits to a more general manipulator to allow applicability to a wider range of tasks.

There are multiple current commercial approaches to robotic picking and placing of food that do not require complicated manipulation strategies. Most of these approaches use custom grippers designed for specific types of food. For food slices, the robot can slide support surfaces underneath the food from both sides, and stabilize the slices from above. This principle is used in the commercially available "Meat Gripper" built by Applied Robotics [12], which can pick up and deposit sliced meats and cheeses. An attractive gripper for food slices, in particular for tomato and cucumber slices, is developed by Davis et al. [13]. It is based on the Bernoulli principle, and shows great promise for food picking, similar to the success seen by suction grasping for packaged items. However, such an end-effector could not also be used for a task like assistive feeding.

Finally, forks have previously been used successfully by Gallenberger et al. [14] and Herlant [15] to pick up food and bring it to a user's mouth for autonomous feeding tasks. In those works, a linear skewering motion (either vertical or angled) was used to successfully pick up the food and bring it to the user's mouth. In this thesis, we use kinesthetic teaching on a light, easily operated input robot to define the full pickup trajectory, and consider how best to fit a cloned trajectory to a potentially larger, more robust robot's arm kinematics.

1.2 Objectives and Approach

In this thesis, we consider the problem of robots manipulating food in uncontrolled domestic environments and tackle challenges of perception, error recovery, and trajectory learning.

For perception, we only provide loose constraints on the household environment when the robot interacts with food. In the food manipulation challenges we investigate, the robot is interacting collaboratively with a human, whether the human is presenting food to the robot to plate, or whether the robot is presenting food to the human to eat. We consider in the plating task, for example, that the food may be placed in a general, not precisely defined, location when presented to the robot, so the robot needs to be able to perceive the food and adapt to the placement of the food. In this work, we describe the process we use to localize an object on the table, our procedure for teaching robots to detect the pixel location of an object, our proposal for using "proxy classes" to quickly train a deep-network object detector to detect new objects, and our use of an RGB-D camera for object localization when the object is not on a table.

There is intrinsic variability in the physical properties between particular instances

of food. Some grains of rice may stick more closely together than others. Some tomato slices may be more or less ripe or have different internal structures which may affect physics properties and may be difficult for the robot to detect before interacting with the tomato slice. While it seems likely that one could improve the perception of the robot to better detect and predict the physics properties of the food that the robot is manipulating, we nevertheless believe it is impossible to do so perfectly, so in our work we include feedback systems and autonomous error-recovery in the robot behaviors. We build a feeding robot and then show how our use of a spoon-facing camera to provide feedback on the amount of food acquired by the robot improves the performance of the feeding robot.

We commit to teaching our robots plating and feeding skills by learning from demonstration and by replaying trained trajectories. In our Niryo One and Kinova MICO feeding robots, we consider the case where a demonstration trajectory is presented as a visual recording of the poses of a fork used by a human. In our Pepper rice serving robot, we consider the case where a demonstration trajectory is kinesthetically trained on the robot by a human physically moving the joints of that robot through the desired motions. Finally, in our caprese salad plating robot, we consider the case where a demonstration trajectory is trained kinesthetically on one robot (a Niryo One robot), and then the resulting trajectory is transferred to different type of robot with a different kinematic layout (a UR5 robot). Though the robot can succeed by copying human-demonstrated trajectories, we allow the robot to modify and improve learned trajectories to work more efficiently on the particular kinematic structure of the robot's arm. For the Pepper robot, the learning is supervised by human feedback on task success after the robot performs the action. For the UR5, the learning is automated to detect and prevent collisions or motions outside the workspace and performed in simulation.

1.3 Contributions

This thesis makes the following contributions to the field of artificial intelligence for robots:

- We propose the use of "proxy classes" to teach robots novel objects in household scenes
- We show that an assistive-feeding robot's use of visual feedback improves its efficiency in performing food acquisition
- We introduce the Parameterized Similar Path Search (PSPS) algorithm, through which a robot can improve on a trajectory taught using kinesthetic teaching

- We show how a robot can take advantage of task symmetries by rotating a recorded trajectory to accommodate its kinematics
- We present implementations of feeding and plating algorithms on real Pepper, Niryo One, Kinova MICO, and UR5 robots

1.4 Reader's Guide to the Thesis

The rest of this work proceeds as follows. Chapter 2 discusses how we teach robots visual perception skills for food manipulation. Chapter 3 discusses our feeding robot and shows the efficiency of error recovery and feedback systems on that robot. Chapter 4 discusses kinesthetic teaching of food manipulation motions and introduces the PSPS algorithm to improve those learned motions. Chapter 5 discusses our caprese salad plating robot and shows how it can modify trained trajectories to expand its workspace. Chapter 6 concludes this work and presents suggestions for future work.

Chapter 2

Visual Perception for Food Manipulation

We teach our robots to sense their surroundings and understand where relevant objects are located so that they can succeed in unconstrained household environments. In this chapter, we present the vision-based approaches our robots use to identify and localize objects. First, we describe the process we use to localize some object on the table, given that we know its pixel coordinates in a camera image. Then, we explain our procedure for teaching robots to detect the pixel location of an object in an image using color segmentation, which approach we use in our caprese salad plating robot presented in Chapter 5. We also present our proposal for using "proxy classes" to quickly train a deep-network object detector to detect new objects [16]. Finally, we discuss the use of an RGB-D camera for object localization when the object is not known to be constrained to lie on a table, which we use to locate the mouth of a user when feeding them [17].

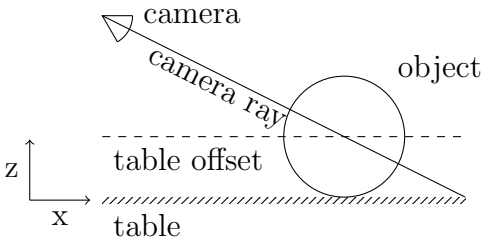
2.1 Problem Statement

In general, we consider our food manipulation environment to be "unstructured" because we do not require the food to be placed in a very specific position in order for the robot to interact with it. There is a qualitative difference between a food processing machine into which one inserts food, and the more generic food manipulation robot that is itself able to find and pick up food presented to it, or is itself able to detect where the food should be fed. One might imagine a mobile robot with a large workspace collecting food to manipulate and prepare, though in this work we assume that the food presented to the robot is placed somewhere in a predefined polygonal region on a table in front of the robot. This provides for the robot's vision system the constraint that the food is located somewhere on the plane of the table. With that assumption, we can use a single RGB camera mounted in a known position somewhere

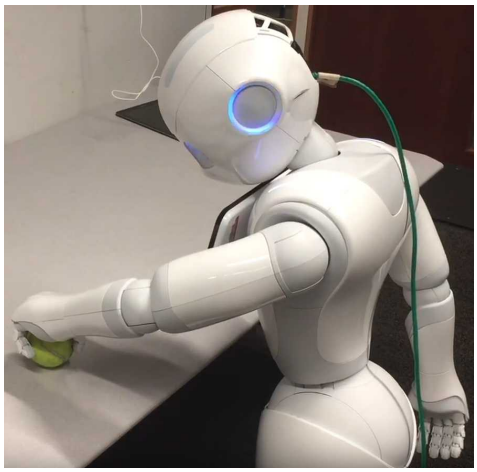
on the kinematic chain of the robot manipulator, or fixed in the world frame, to detect and localize the food for manipulation.

2.2 Object Localization on a Table

If an object is detected in a camera image, and it is known that the object is resting on a known table plane, the process we use to localize the object in three dimensions is as follows. First, we use the `image_geometry` ROS package to parse the camera's intrinsic parameters according to a pinhole camera model and to convert from a pixel location in the camera image to a ray in the camera frame. For this work, we assume we know the relative location of the table plane to the robot camera. Thus, we can use the `robot_state_publisher` and `tf` ROS packages to convert camera pixel ray to be in the same frame as the table plane. We intersect the camera pixel ray with the table ray to get the object location in three dimensions.



(a) Projection diagram



(b) Algorithm running on Pepper so it can grab a tennis ball

Figure 2.1: To account for the height of an object when localizing the object on a table, the camera ray should be intersected with the table offset plane (not the table plane) in order to compute the correct x location of the object center

If the detected object center has a known height above the table, the procedure can be adjusted for this height. For example, in order to have the Pepper robot pick up a tennis ball from a table, we noted that the center of the tennis ball was roughly 3cm above the table. Thus, we intersected the camera pixel ray with a plane 3cm above the table in order to locate the center of the tennis ball. If the camera is not directly above the object, this adjustment corrects the x, y locations of the object as well as its z location. Without this adjustment, the robot will project the object center onto a point on the table *behind* the object (Figure 2.1).

(a) UR5 wrist camera image (b) Region of interest

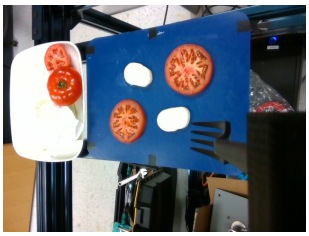
Figure 2.2: The vision system using the UR5 wrist-mounted camera to localize food items on the table

Additionally, in order to prevent the camera from fixating on objects in the background scene, if the robot is given a programmed polygon on the table where food will be presented to it, we can focus the robot on the relevant region as follows. First, we back-project the polygonal region of interest on the table into the camera frame, using the `image_geometry` package. Next, we create a binary image mask to only include the relevant polygonal region using OpenCV's `fillConvexPoly` function [18]. Finally, we apply that mask during the image processing procedure to prevent the detection of objects located in the background. Figure 2.2 gives an example of this mask. It shows the region of interest computed based on a pre-defined rectangle of interest inside the blue cutting board on the table.

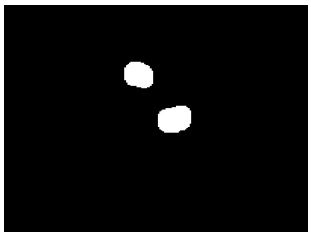
2.3 Color Segmentation for Object Recognition

We now address the problem of identifying discrete food object pixels in an RGB camera image. If the food objects have a known color distribution that doesn't overlap with the color distribution of the table, as is the case when making a caprese salad on a blue cutting board, we can use color segmentation to localize the tomato and cheese pixels in the image.

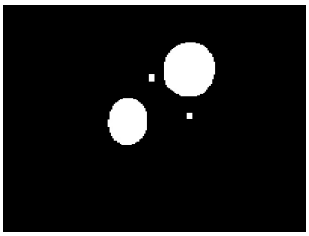
For ease of teaching the robot to identify the objects, the user can provide separate reduced (4x4 pixel or 16x16 pixel) PNG images containing pixel samples from the different food items and from the background. Providing these images is the way the programmer teaches the robot what color the objects to detect are. We then use a nearest neighbors approach using Euclidean distance in the RGB space to label each pixel with whichever image (food item or background) it matches most closely in color. For each food item, this gives us a binary mask of that food item's pixels. We use an erosion and dilation procedure to clean these binary pixel identifications, and we group connected components in order to identify separate instances of our food items on the table. In general, we pick the center point of the largest connected component



(a) Wrist camera image



(b) Cheese pixels



(c) Tomato pixels

Figure 2.3: Cheese and tomato pixels are identified within the masked region of interest based on taught color samples of cheese, tomato, and background. Since we then segment connected components and choose the largest component, the small noise components in the tomato identification are not a concern

for a given class of food as the target point for that class of food. This process is displayed in Figure 2.3. The robot is trained to differentiate tomatoes and cheese from the background color by being given color samples from cheese, tomatoes, and the background cutting board.

2.4 Deep-Network Object Recognition

While color segmentation can work for simple cases, more advanced techniques give more flexibility, allowing for identification based on the shape of an object, or its texture. Deep networks show great promise in the field of object recognition, but have one downside in that they require many samples. By contrast, you only need a single, exceptionally low-resolution PNG for each object and the background in order to train the color segmentation detector.

We investigate the possibility of taking a network that is pre-trained on some categories and then applying that network to novel objects. For example, we analyze the performance of YOLO and Faster R-CNN networks trained on the COCO dataset and apply those networks to detect objects that do and do not map intuitively to categories in the COCO label set, to simulate real-world scenarios where the robot is asked to identify classes of objects on which it has not been trained [19, 20, 21]. We propose a learning algorithm where the robot associates novel objects with the class label(s) they most resemble.

To empirically measure whether the networks are effective in distinguishing a variety of arbitrary, everyday objects, we instruct the robot to take images of seven objects in 50 different configurations (examples in Figure 2.4), and we hand-label the center point of each object in each image. Table 2.1 presents the first, second, and third quartiles of the confidence scores for the four classes with the highest median confidence for each

Table 2.1: A summary of class scores for Faster R-CNN and confidence levels for YOLO across 50 images

	Faster R-CNN				YOLO		
	Q1	Q2	Q3		Q1	Q2	Q3
Mug							
cup	99.86	99.91	99.96	cup	88.53	90.65	91.97
refrigerator	2.45	5.22	8.83	sink	0.09	0.29	1.22
bottle	0.87	2.40	4.79	bed	0.10	0.29	0.54
vase	0.00	0.81	1.58	laptop	0.09	0.21	0.39
Can							
bottle	76.49	85.17	90.01	bottle	24.29	40.71	61.73
cup	13.36	33.31	44.99	cup	1.04	3.69	16.03
refrigerator	1.73	4.42	8.60	bed	0.06	0.20	0.41
vase	0.67	1.78	3.41	sink	0.08	0.17	1.22
Tennis ball							
sports ball	69.71	85.90	94.79	sports ball	3.57	27.84	64.23
apple	7.42	15.05	33.26	apple	3.78	15.17	38.29
refrigerator	1.73	4.42	8.46	orange	3.55	10.51	20.95
orange	0.13	1.11	2.04	bed	0.07	0.20	0.41
Block							
refrigerator	1.93	3.84	8.46	cell phone	0.42	1.35	3.54
cell phone	0.00	2.09	10.10	book	0.23	0.82	2.35
book	0.70	1.40	3.33	remote	0.09	0.26	0.85
tie	0.13	1.02	2.23	knife	0.08	0.23	0.82
Golf ball							
sports ball	36.04	50.65	60.14	sports ball	13.52	27.39	50.61
mouse	6.61	10.89	15.95	mouse	2.98	12.72	25.91
apple	7.26	9.32	13.11	apple	0.30	1.44	5.92
refrigerator	3.04	5.82	8.90	orange	0.11	0.47	1.14
Red ball							
apple	50.99	60.11	72.49	apple	30.29	44.43	57.34
sports ball	6.95	11.44	16.31	orange	14.06	25.13	43.51
mouse	4.17	6.15	12.64	sports ball	0.57	2.10	8.00
refrigerator	1.73	4.42	8.46	mouse	0.24	0.51	1.18
Marker							
bottle	76.74	86.95	91.60	bottle	17.71	29.44	45.89
toothbrush	11.03	20.92	46.98	toothbrush	7.75	20.68	36.86
refrigerator	2.83	5.46	8.83	remote	0.29	1.15	2.77
cup	0.68	1.71	3.24	spoon	0.20	0.55	1.35

Figure 2.4: Sample images of everyday objects from our object recognition experiments

object.[1]

We first consider the objects which intuitively belong to a COCO class (i.e., mug: "cup," tennis ball: "sports ball," golf ball: "sports ball"). For those objects, we find that the median assigned confidence is highest for the matching class labels, for both YOLO and Faster R-CNN.

We next consider objects like can, red ball, and marker, which do not have an intuitive label in the COCO set.[2] We find that the robot nevertheless consistently assigns certain classes to those objects regardless of orientation. We call those classes "proxy classes." We define a proxy class for an object to be a class label that is consistently triggered more strongly by that object than by other objects in the environment. For example, Table 2.1 shows that the "toothbrush" class (the second most strongly triggered class by whiteboard markers) is a proxy class for markers. The robot is able to identify markers as the only objects in our images that "look like" toothbrushes. Of course, if the environment also contains an actual toothbrush, or some other object that "looks like" a toothbrush, then the "toothbrush" class would not be sufficient to distinguish markers from that other object.

In cases where a single proxy class is not sufficient, combinations of classes can be used to differentiate objects. We propose an algorithm in which the robot searches for objects that "look like" a specific combination of target classes. For example, the robot might be shown a can and learn that the neural networks classify the "can" object as a combination of "bottle" and "cup" classes. It could then search for objects that look like both a "bottle" and a "cup" and thereby be able to identify cans and also to distinguish them from bottles and cups.

We propose that by combining information across classes in this manner, any of our everyday objects can be differentiated from the others. We believe that algorithms

[1]The "dining table" class is removed from Table 2.1 since both neural nets assign that class a large bounding box around all the objects (correctly identifying that these objects are resting on a table).

[2]The 80 COCO class labels can be found at `http://cocodataset.org`.

of this type will be useful in the scenario where a robot is asked to interact with a novel generic object class on which it has not previously been trained. In this way, a food manipulation robot could modify a pre-trained deep-network object recognition algorithm to detect novel food items, even if it has not been previously trained on that precise food category.

2.5 3D Object Localization using Point Clouds

If the object is not located on a known plane, then more information than a pixel location is needed in order to localize the object. This occurs, for instance, when localizing the mouth point to which food should be brought when feeding a person. For this type of localization, one approach we found effective was Discriminative Optimization, a 3D point-cloud registration algorithm.

With Discriminative Optimization, a model of the point cloud that should be identified is stored. In the case of feeding a person, a model of the person's head is recorded as a 3D point cloud. This model is then processed so that for any new point cloud presented to the algorithm, an approximate gradient can quickly be computed that rotates and translates the new point cloud to more closely align with the model point cloud. This approximate gradient descent is repeated a fixed number of times to give an approximate alignment of the new point cloud to the model point cloud.

In the case of identifying a feeding point location for a feeding robot, if the mouth point of the person is known relative to the model point cloud, then the Discriminative Optimization algorithm (or a similar point-cloud-matching algorithm, like Iterative Closest Point) can be used to identify the 3D location of the head model in the world-frame, and the mouth point is then a known location in the head model frame.

As long as the transformation from the RGB-D camera to the base of the robot is known, this procedure can be used to localize the mouth of the person being fed in the robot's frame of reference. Thus, as long as the camera is properly calibrated, the camera that is tracking the mouth of the user does not need to be physically attached to the robot, but could be placed in any convenient place away from the robot facing the user. Since most RGB-D cameras have a minimal sensory range, it might not be beneficial to put an RGB-D camera on the wrist of the robot, since during feeding the wrist of the robot is likely to get closer to the user's face than the minimal sensing range of the depth camera.

2.6 Summary

In this chapter, we presented vision-based approaches we implemented on our robots in order to perceive and localize objects in their environment. We explained how our robots can localize objects on a plane, how we teach our robots to identify objects using

color segmentation, and our proposed "proxy classes" so that the robot can quickly identify new objects using a deep-learning framework. Finally, we explained our use of Discriminative Optimization for 3D point-cloud registration to localize objects not on a table.

Chapter 3

Vision-based Feedback Systems and Error Recovery

This chapter introduces our feeding robot and presents the work we did using visual feedback and error-recovery behaviors to improve its efficiency [17]. We show how a spoon-facing camera can detect the amount of food on the spoon the robot is holding, and how the robot can use this information to modify its behavior to bring food more efficiently to the user's mouth. This work explains our implementation and experiments on a real Kinova MICO robot. We also applied these same algorithms to smaller Niryo One arms (Figure 3.1), and one of those robots is now in regular use assisting one of our collaborators with cerebral palsy and will be used in their lab for future studies.

3.1 Food Acquisition

We propose a real-time system that uses depth images to track the user's mouth in 3D space and a separate vision system to provide useful feedback for how much food the

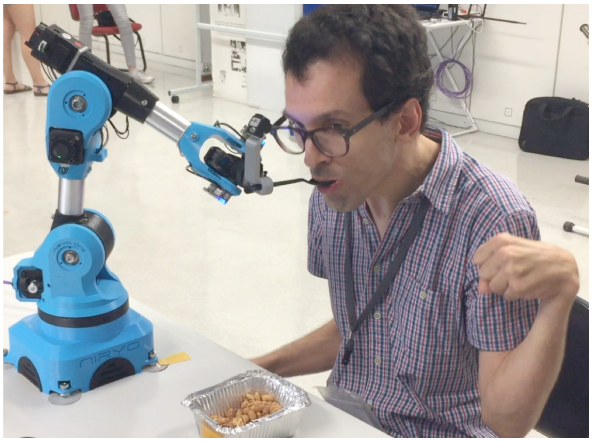


Figure 3.1: Our Niryo One assistive-feeding robot

(a) Preparing (b) Scooping (c) Carrying (d) Eating

Figure 3.2: Our assistive-feeding robot includes a MICO robot arm, an RGB-D camera, and an RGB camera. Figure (a) also labels the axis orientations of the robot coordinate system. A video of our robot is available at `https://youtu.be/X7McqWk1AK8`

robot acquired on its spoon. Real experiments show that the visual feedback module significantly improves the feeding system's performance.

3.1.1 Feeding System

For this work, we consider a feeding system composed of two parts: 1) a vision system that is responsible for detecting if there is food on the spoon and the location of the user's mouth and 2) a control system that is responsible for transforming visual perceptions into tasks executed by the robot arm.

3.1.2 Vision System

To see what is on the spoon, we place a tiny RGB camera on the end-effector of the robot arm. After a simple calibration step of holding a light-colored background behind the spoon, we compute a mask for the image that only contains the spoon. We can use this masked image for two purposes, to detect if there is enough food to serve the user, and to detect if the user has eaten the food off the spoon. To detect if there is enough food on the spoon we use a detection algorithm that we can tune to specify the required amount of food. To detect if the user has eaten the food, we use a classifier with two classes: "food" and "no food."

To tune our food-detection threshold, we gathered a dataset containing 387 data points of masked images of the spoon and the corresponding weight of food present on the spoon. Examples are shown in Figure 3.3. We use peanuts and fried rice as our test foods. We use the histogram of colors present in the spoon image to inform the choice of the threshold for "enough food" on the spoon.

(a) Empty spoon (b) Peanuts (c) Fried rice

Figure 3.3: Sample masked images of the assistive-feeding robot's plastic spoon, showing the spoon empty, with nuts, and with rice

The robot uses an RGB-D camera to detect the location of the user's mouth. First, we train a model of the user's head, and then the robot uses Discriminative Optimization to perform 3D point-cloud registration to detect the user's head in the scene, as explained in Section 2.5. The mouth point is assumed to be in a static location on the head model.

3.1.3 Control System

Gradient-based planning

The goal of our planning algorithm is to move the tip of the spoon from its current location to the user's mouth. Because the mouth of the user may be moving, we need our planning algorithm to be able to quickly re-compute its plan based on updated mouth-target positions. For this reason, we choose a planning algorithm that iteratively moves the tip of the spoon one step in a straight line toward the current location of the mouth. We also want to ensure that the spoon does not dump its contents during transit, so we control the orientation of the spoon during transit.

A flexible approach to allow quick re-planning and to constrain orientation is a gradient-based approach. At each timestep, we compute an intermediate Cartesian end-effector target that is a certain distance, parameterized by a parameter "translationStepSize," from the current location. The intermediate Cartesian target is the mouth location itself if the mouth location is within "translationStepSize."

We similarly compute an intermediate orientation target by computing the rotation necessary to convert from the current orientation of the end-effector to the target orientation. We define the target orientation to lie along the y-axis direction shown in Figure 3.2. We set our intermediate orientation target to be the orientation that is at most a certain angle rotation, parameterized by a parameter "rotationStepSize," away from the current orientation. During transit from the plate to the user's mouth, the orientation is never more than a very small angle away from the desired final feeding orientation, so the target orientation is always the desired final orientation.

We use OpenRave [22] to compute the Cartesian Jacobian, which is a $3 \times$ DOF ma-

trix showing how changes in each joint angle locally move the end-effector in Cartesian space. We also use OpenRave to compute the angular velocity Jacobian, which is a $3 \times$ DOF matrix showing how changes in each joint angle locally rotate the end-effector, where rotations are given in angle-axis representation.

We write our desired translation as a length 3 vector. We concatenate that vector with the angle-axis representation of our desired rotation to give us a length 6 vector representing our desired change in end-effector pose. Our "translationStepSize" and "rotationStepSize" parameters guarantee that our intermediate target pose is "close" to our current pose, so we can reasonably linearize our problem to be Equation 3.1, where J_c and J_r are the Cartesian and rotation Jacobian defined above, Θ is the change in the joint angles we are calculating, and Δ_c and Δ_r are the desired translation and rotation to move the end-effector to the intermediate target. In our experiments, we use a 6DOF robot, so we can use ordinary least-squares regression to compute the required joint changes necessary to satisfy that equation. For a robot with more degrees of freedom, regularization could be used to prefer solutions with small joint angle changes.

$$\begin{bmatrix} J_c \\ J_r \end{bmatrix} \Theta = \begin{bmatrix} \Delta_c \\ \Delta_r \end{bmatrix} \tag{3.1}$$

We find that this simple architecture works for the majority of our feeding tests. Since this gradient-based algorithm is greedy, we do note that it is possible for the robot arm to follow trajectories into local minima where joint constraints prevent the robot arm from continuing all the way to the user's mouth. In our setup, we find that this generally happens if the trajectory brings the robot end-effector too close to the robot base. Placing the user and plate so that the path between them stays more than a certain distance from the robot base circumvents this issue and leads to successful trajectories, though we also implement a more robust solution that modifies planned trajectories to keep away from the robot base while moving.

Learning from Demonstration

In addition to defining trajectories from the plate to the mouth and back, we also need to train the robot to acquire food from the plate. To do so, we use learning from demonstration to imitate human utensil trajectories. In this manner, we do not need to perform the complex task of modeling different food types in order to plan food acquisition strategies. The ability of learning from demonstration to plan trajectories without an explicit model of the environment dynamics [23] is an attractive benefit for us in this context. "Learning from demonstration" in the robotics context usually means some mechanism of automatically acquiring knowledge from human demonstration, but for this work it was sufficient to pick and imitate a single trajectory from a collection of trajectories for assistive feeding. In [24], Bhattacharjee et al. collected detailed fork trajectories in a simulated assistive feeding environment. We

visualize several of those trajectories in Rviz and select and truncate a single fork trajectory that follows a simple scooping motion in acquiring coleslaw. Our robot imitates that single fork trajectory to acquire food with a spoon. We find that single scooping trajectory to be sufficient for scooping up the different food types we tested, but we expect that there are food types and serving plate configurations for which a more customized trajectory is required.

To have our robot spoon replicate the training trajectories, we want the robot to move its arm in such a way that the tip of the spoon follows the demonstrated tip of the fork in both position and orientation. In addition, we want the robot to be able to translate the demonstration trajectory target by various offsets so that the robot can scoop food from different parts of the serving plate. To accomplish this, we use the same gradient-based planner described above, where now the target position and orientation of our gradient-based controller are generated by playing back the recorded utensil tip poses, offset by the desired translation. We slow down the played-back demonstration to one fifth of the demonstration speed due to speed constraints of the robot arm and to ensure safety.

3.2 Experimental Setup and Results

3.2.1 Hardware Setup

To test our food acquisition feedback algorithm on an actual robot, we use a 6DOF Kinova MICO robot affixed rigidly to a dining table. We place a serving bowl of food in front of the robot and have the robot hold a spoon in its end-effector. We attach an iDS-xs RGB camera to the end-effector pointed toward the bowl of the spoon. The camera publishes a ROS topic which is used for food acquisition feedback.

3.2.2 Ablation Study for Food Acquisition

Setup

To analyze the importance of various components of the food acquisition system, we perform an ablation study in the feeding task where we measure the efficiency of the feeding robot when it does and does not use certain feedback and error recovery strategies. We define the efficiency to be the mass of food that is delivered to the feeding location as a function of the total distance that the tip of the spoon has traveled. We report the mass as a function of distance traveled rather than the time taken because we want to negate any effect that setting the robot to a faster or slower speed would have on the results.

The feedback and error recovery strategies that we alter in our ablation study are 1) whether or not the robot re-scoops if the spoon-facing camera detects that not enough

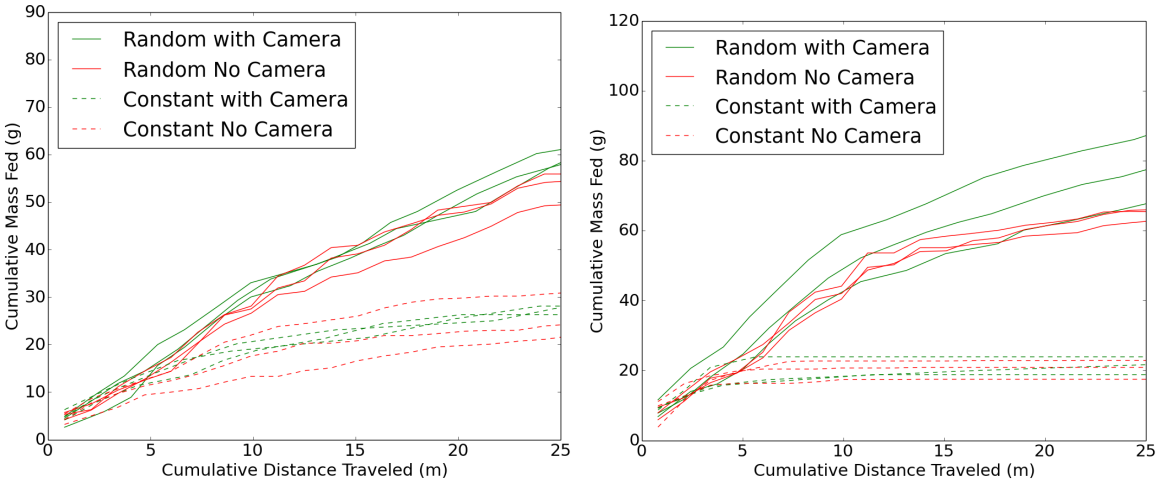
food was acquired and 2) whether the robot always scoops from the same position on the serving plate or whether it scoops from a uniformly random position within a 6cm x 3cm rectangle on the plate. We perform the ablation experiment on two different kinds of food: peanuts and fried rice. In this way, we can determine if the importance of system components depends on the type of food.

For consistency in results, we use the same random seed for all trials that randomize the scooping location on the plate. To speed up data collection, the robot dumps the food directly onto the scale after food acquisition. Then, in our data analysis we add in the distance to the user's mouth and back. Cutting out the travel time between the plate and the user's mouth cuts the data acquisition time by a factor of three. On average, the distance traveled by the spoon tip in a single scooping motion is 32cm and the average distance to and from the user's mouth is 49cm in each direction. We use these average distance values when representing the amount of food served as a function of distance.

Results

In our ablation study for food acquisition, we find that vision feedback significantly increases the amount of food brought to the user in each bite. For rice, when randomizing the scooping location, the average amount of rice served per bite in the first 20 bites across three trials is 4.8g with camera feedback and 3.2g without camera feedback (paired t-test p-value 9e-6). Likewise for nuts it is 3.8g with camera feedback and 2.7g without (paired t-test p-value 1e-5). There is an added distance that the end-effector travels in re-scooping when using camera feedback. Despite this added cost in distance, Figure 3.4 shows that even when looking at the amount of food served as a function of the total distance that the spoon tip travels (including the added distance required to re-scoop), after 25 meters, random scooping with camera feedback consistently outperforms random scooping without camera feedback.

For both rice and nuts we find that randomization of the scooping location is a useful technique in food acquisition. Figure 3.4 also shows that randomization coupled with visual feedback on the success of a scoop is the most effective. Most interestingly, we find that the importance of our system components depends on the type of food used. We find that randomizing the location of scoops is more important when serving rice instead of nuts. This could be related to the fact that nuts were observed to "settle" after each scoop, whereas fried rice will tend not to fill in the hole left after scooping. Thus, scooping the same place for fried rice will quickly result in very little (almost no) mass acquired by the spoon in subsequent scoops.



(a) Feeding nuts

(b) Feeding rice

Figure 3.4: Charts showing the amount of food fed (in grams) as a function of the distance the tip of the spoon has traveled, including the distance traveled in re-scooping. The chart includes results for scooping location randomization (solid lines) compared to a constant scooping location (dotted lines), and it includes results with (green lines) and without (red lines) camera feedback on whether food was successfully acquired. We ran three trials for each test type

3.3 Summary

In this chapter we presented our assistive-feeding robots, which include a Kinova MICO robot and a Niryo One robot, and explained the implementation of our feeding algorithm in detail. We also showed how a feedback system to detect the presence of food on the spoon of the robot improves the efficiency of the assistive-feeding robot. We performed our feedback efficacy experiments on a real Kinova MICO robot, and implemented our feeding algorithm on both a Kinova MICO robot and a Niryo One robot.

Chapter 4

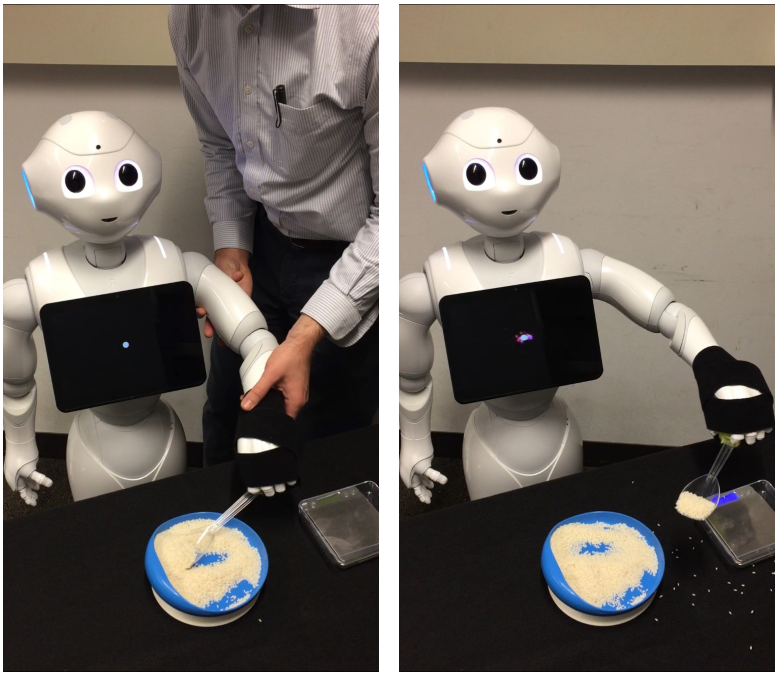
Robot-focused Trajectory Improvement

Our research aims to improve the capabilities of autonomous feeding robots to allow a caretaker to specify the full trajectory of the robot arm rather than just the final position. Specifying the full feeding trajectory could also enable a caretaker to expand the capabilities of the feeding robot to perform additional tasks, like combining foods before each bite or dipping each bite in sauce before moving the food to the feeding location. We commit to achieving this by using learning from demonstration, in particular kinesthetic teaching, where a teacher physically moves a robot through a desired trajectory. In this chapter we present our work on a robot using trial and error and feedback from a human teacher to improve on a kinesthetically trained trajectory [25]. We introduce the Parameterized Similar Path Search (PSPS) algorithm and show how the robot can use PSPS to improve the trained trajectory over its own cost function while still completing the desired task. This work was implemented on a real Pepper robot.

4.1 Problem Statement and Formalization

We address the problem of how an assistive robot can improve on a trajectory learned through kinesthetic teaching. The robot is kinesthetically trained on some task, as in Fig. 4.1a, and our problem is to construct a learning algorithm whereby the robot can improve on that task, as in Fig. 4.1b.

For an assistive feeding robot with m degrees of freedom, let $\mathcal{J}$ be the set of valid static configurations $\mathcal{J} = \{(j_1, j_2, \ldots, j_m) \in \mathbb{R}^m\}$ where j_i is the parameterization of the i^{th} degree of freedom. For example, if the i^{th} degree of freedom corresponds to a revolute joint, then j_i would measure the angle of that joint. We consider a trajectory to be a set of pairs of joint states and times $\{(\boldsymbol{j}_1, t_1), \ldots, (\boldsymbol{j}_k, t_k)\}$, with $\boldsymbol{j}_i \in \mathcal{J}$, $t_i \in \mathbb{R}$, and $t_i < t_{i+1}$. We call the set of all trajectories Φ. Since our robot is trained kinesthetically,



(a) Pepper learning from kinesthetic training

(b) Pepper improving learned trajectory using PSPS

Figure 4.1: Images of the Pepper robot learning the rice serving task presented in Section 4.3. A video of these experiments can be found at `https://youtu.be/jMyzH8Yu2mM`

the training demonstration is an element of the set Φ.

We assume there exists some cost function $C : \Phi \to \mathbb{R}$, known to the robot, that gives a measure of the cost of any trajectory. This cost function could be any cost, like the total path length, the smoothness of the path, or the energy needed to achieve that trajectory.

We assume that the teacher knows some task-success-evaluation function $S : \Phi \to \{0, 1\}$, unknown to the robot, that tells whether a trajectory satisfactorily completes the task, in which case S returns 1, or not, in which case S returns 0. The task-success-evaluation function is evaluated by the teacher, and the teacher decides whether the robot's attempt to complete the task was successful. For example, if the task is to scoop up a dumpling, dip it in sauce, and move it to the feeding location, then the success function only will return true if the robot accomplishes all those tasks to the satisfaction of the teacher. The robot is able to ask the teacher to evaluate the success function $S(\phi)$ for any trajectory ϕ. While we allow the robot to ask the teacher to evaluate any trajectory, that trajectory evaluation does involve the robot actually executing the trajectory. Since novel exploratory trajectories can be dangerous to execute on a real robot, in our work the robot gradually changes and improves its trajectory around a known viable trajectory rather than searching across the whole space of trajectories.

Each time the robot asks the teacher to evaluate the success function is a learning step. Pursuant to [26], we consider enhancing this learning problem by having the teacher also provide an estimate of the time(s) when the robot trajectory failed the task. Though our experiments find satisfactory results without including that component, we consider further research in that area to be a possible avenue of future work.

We define our problem as follows: The robot is shown a single successful trajectory $\phi_d \in \Phi$. The robot asks the teacher to evaluate the success of N different trajectories. The robot's learning problem is to find, after these learning steps, a learned trajectory ϕ_l for which $S(\phi_l) = 1$ and which makes $C(\phi_d) - C(\phi_l)$ as large as possible.

We note that this formalization can be extended naturally to handle a stochastic success function S_s, where $S_s(\phi)$ returns 1 with probability $p(\phi)$ and 0 with probability $1 - p(\phi)$, by having the robot attempt to maximize $\mathbf{E}[S_s(\phi_l)(C(\phi_d) - C(\phi_l))]$. This maximization can be empirically estimated by evaluating $S_s(\phi_l)$ multiple times.

4.2 Parameterized Similar Path Search (PSPS)

The PSPS algorithm is based on the assumption that similar trajectories are likely to have similar success values. Thus, we want the algorithm to focus its search on trajectories that are similar to the training trajectory. We also note that, because exploratory trajectories on a real robot can be dangerous, having the learning algorithm focus on trajectories similar to the kinesthetically trained trajectory is also relevant from a safety perspective.

We therefore define the PSPS algorithm as follows. First, the robot defines a function $d : \Phi \to \mathbb{R} \geq 0$ that measures how different a new trajectory is from the training trajectory.[1] Second, the robot picks some distance parameter $\delta_{cur} \in \mathbb{R}$ and searches over the set of trajectories $\{\phi \mid d(\phi) < \delta_{cur}\}$ for the trajectory in that set that minimizes the cost $C(\phi)$. The robot then performs a learning step and asks the teacher if that trajectory is successful. If the trajectory is successful, the robot increases δ_{cur} and repeats. If the trajectory is not successful, the robot decreases δ_{cur} and repeats.

If the robot is given information about particular parts of the trajectory that led to failure, the robot may also update the distance function d to make the trajectory more closely align to the training trajectory for those failing timesteps.

[1] For linguistic convenience, we call d a "distance" function in this paper. However, since d only defines a notion of distance *from the training trajectory*, d does not satisfy the mathematical definition of a metric.

Algorithm 1 Parameterized Similar Path Search Algorithm

Initialize $\phi_{current\_best}$ to the trained trajectory
Initialize $\delta_{min} \leftarrow 0$
Initialize δ_{cur} to some arbitrary value
Initialize *isDeltaMaxKnown* $\leftarrow$ False
iteration $\leftarrow 0$
while iteration $< N$ **do**
    if *isDeltaMaxKnown* **then**
        $\delta_{cur} \leftarrow (\delta_{min} + \delta_{max})/2$
    else
        $\delta_{cur} \leftarrow \delta_{cur} * 2$
    end if
    Define similar trajectories $\Phi_\delta = \{\phi \mid d(\phi) < \delta_{cur}\}$
    $\phi_{test} \leftarrow \arg\min_{\phi \in \Phi_\delta} C(\phi)$
    Ask teacher to evaluate *success* $\leftarrow S(\phi_{test})$
    if *success* **then**
        $\phi_{current\_best} \leftarrow \phi_{test}$
        $\delta_{min} \leftarrow \delta_{cur}$
    else
        $\delta_{max} \leftarrow \delta_{cur}$
        isDeltaMaxKnown $\leftarrow$ True
    end if
    if The robot receives additional feedback **then**
        The robot updates d based on that feedback
    end if
end while
return $\phi_{current\_best}$

This active learning algorithm focuses the robot's search on trajectories that are likely to improve the cost function C. We expect this algorithm to be robust to the choice of the initial δ_{cur}, as it performs an exponential search in the δ parameter. We note that increasing the δ will always lead to an equal or lower-cost trajectory, but may lead to trajectories that do not accomplish the desired task. For safety reasons, to avoid the robot executing dangerous trajectories, we suggest that δ_{cur} be initialized to a small value and that the feasibility of the initial δ_{cur} be tested in simulation before this algorithm is run on a new robot.

4.3 Experimental Evaluation and Results

We run experiments to better understand the feasibility of this algorithm and its real-world performance. We are interested in evaluating how much this algorithm is able to optimize sample learned trajectories and whether it is necessary for the teacher to provide information on the time of trajectory failure (lines 22-23 in Algorithm 1) to make the PSPS algorithm effective.

We ran the PSPS algorithm in both the 2D simulated feeding world and on a real robot serving rice.

4.3.1 2D Rice Serving Task Simulation Setup

We first test this algorithm in simulation in a low-dimensional environment so it is easier to visualize its behavior. We consider a two-dimensional world where the robot state J is the x, y coordinates of a spoon. Inspired by our feeding problem, the task constraints are to generate a trajectory that passes from the start location to inside the rice outlined in blue in the bowl on the right (to pick up the food) and then to the green area in the bowl on the left (to deposit the food) without hitting the table or sides of the bowls. The locations of the bowls and table in this simulation, along with the training trajectory curve demonstrated by human input through a mouse, are displayed in Fig. 4.2.

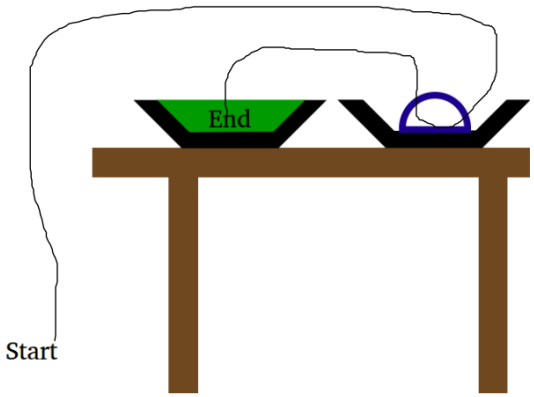


Figure 4.2: The feeding task trajectory input by teacher going from start, to rice bowl (blue outline), to target bowl area (green)

In this simulation, we set the goal to minimize the roughness (sum of squared second derivative) of the trajectory, while ensuring that the square root of the weighted average square distance between the new path and the old path over all timestamps is less than the constraint distance. This choice of smoothness function combined with the distance

function is known [27] to be solved by natural cubic splines, and we are able to use the MATLAB `spaps` function [28] at each step of the PSPS algorithm to optimize the cost function given the constraint.

In this simulation, the mechanics of the PSPS algorithm interaction between teacher and simulated robot is as follows. The simulated robot performs the learning optimization step of the PSPS algorithm by computing a similar spline trajectory to the training trajectory within some constraint distance δ. Then, the simulated robot draws that new trajectory on the screen for the evaluation of the teacher. The teacher inspects that trajectory and reports back whether the trajectory meets the teacher's task constraints or not.

4.3.2 2D Serving Task Simulation Results

The constraint is that the sum of square Euclidean distances between training and tested paths must be no more than $\delta^2 * n$ where n is 544, the number of points in the trajectory. The map giving the layout of the table and bowls is 500x800 pixels, and we begin with $\delta = 5$ pixels. We constrain the spoon trajectory to match the start and end locations closely by multiplying the start- and end-point square errors by 100 before including those errors in the constraint sum.

Fig. 4.3 shows the result of running PSPS in the 2D serving simulation. The training trajectory (here trained by drawing with a mouse on the image) is shown as a dotted black line. The simulated robot spoon first tries a similar trajectory using PSPS and $\delta = 5$ pixels. Since that trajectory gets marked as successful by the teacher, we plot it in green in Fig. 4.3. Since that trajectory was successful, the learning agent then increases δ and iterates, following PSPS. We plot the tested trajectories in Fig. 4.3, with green showing successful trajectories and red showing failures. After six learning iterations, the best trajectory is plotted in blue.

From this experiment we observe that this algorithm can indeed optimize over the training trajectory while satisfying the trajectory constraints without the additional necessity of adding waypoint constraints along the path of the trajectory. For this example case, we find that it is not critical for the PSPS algorithm to receive information about the timing of failures in order to nevertheless find an improved trajectory over the training trajectory.

4.3.3 Pepper Robot Rice Serving Task Setup

We also explore the usefulness of this algorithm on a real robot for a realistic plating task. We use the Pepper robot from SoftBank Robotics and attach a spoon to its left hand. The joint configuration of Pepper is shown in Fig. 4.4.

We task Pepper with scooping up rice from a blue container and dropping the scoop of rice in a target bowl. For this task, we assume there is a separate process

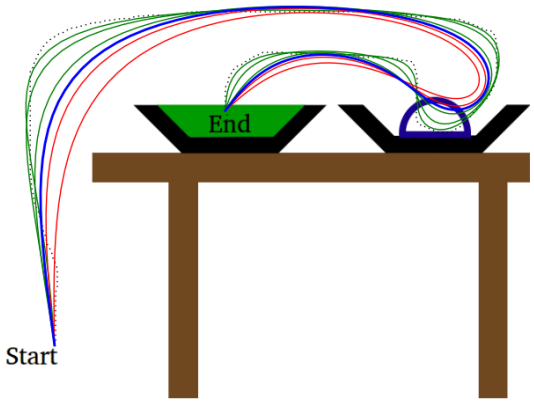


Figure 4.3: Spoon trajectories attempted for this task. Red indicates the human teacher marked that attempt as a failure, green indicates success. The learned trajectory after six iterations is in blue. The black dotted line is the demonstration.

that ensures there is rice ready to be scooped. Future work may include learning preparatory scraping motions with the spoon to position the rice so that it can be easily scooped, and may include incorporating vision to find the rice, stochasticity, or other modifications to ensure that the robot continues to find rice as the bowl is emptied. When vision is incorporated in future work, we hope to experiment with augmenting the learning algorithm to detect the type of food and to learn different trajectories for different types of food. Additionally, we hope to do user studies on the usefulness of the feeding trajectories. Before PSPS is deployed in a user study, safety features to detect anomalies and shut off the robot to prevent injury (like those presented in [30]) should be included. For now, for this task, our robot runs blind (no visual input) and we reset the bowl to be equally full of rice between each attempt so that repeated scoops of rice from the same part of the bowl are equally successful.

We kinesthetically demonstrate a single serving trajectory for the robot that travels from the starting position to the bowl with rice in it, scoops up some rice, and deposits the rice in the target bowl. We weigh the amount of rice deposited in the target bowl after execution, and the trajectory is considered to be a failure by the teacher if the test trajectory deposits less than 50 percent of the rice that was deposited in the bowl by the training trajectory. In our experiment, the training trajectory deposited 29.8 grams of rice onto the target bowl, so whenever the robot asked us to judge a trajectory, we deemed the trajectory a failure if it deposited less than 14.9 grams of rice in the target bowl. Fig. 4.1a shows the Pepper robot being taught a sample serving trajectory kinesthetically and Fig. 4.1b shows the Pepper robot learning improvements to the training trajectory.

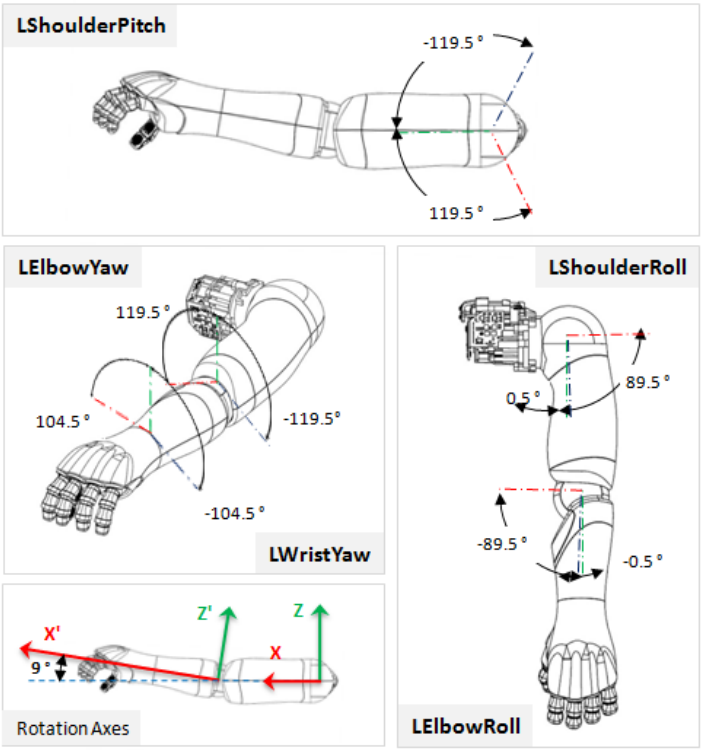


Figure 4.4: The arrangement of joints defining the joint space of Pepper's left arm [29]

We use the same cost/constraint pair that we used in the 2D simulation, with the cost function minimizing the roughness (sum of squared second derivative) of the trajectory, and the distance between two trajectories being measured by the square root of the average square distance between them. We measure both the roughness of trajectories and the distance between trajectories in the joint space of the robot's arm. We use the left arm of the Pepper robot, and the joint configuration is displayed in Fig. 4.4.

4.3.4 Pepper Robot Rice Serving Task Results

We initialize δ to 0.1 radians, after confirming in a simulated environment that that would not lead to dangerous robot trajectories. After six iterations of the PSPS algorithm, the robot had bounded δ to between 0.15 and 0.15625 radians. A plot of the amount of rice deposited in the destination bowl as a function of the δ value used in trajectory calculation is given in Fig. 4.5.

From this plot, we find that the geometric search for the correct δ during the PSPS algorithm is effective. We find that for this example, the data follow the expected trend that trajectories that are above a certain difference from the training trajectory will tend to fail to satisfy the teacher's task constraints.

To compute how much PSPS was able to optimize the trained trajectory, we com-

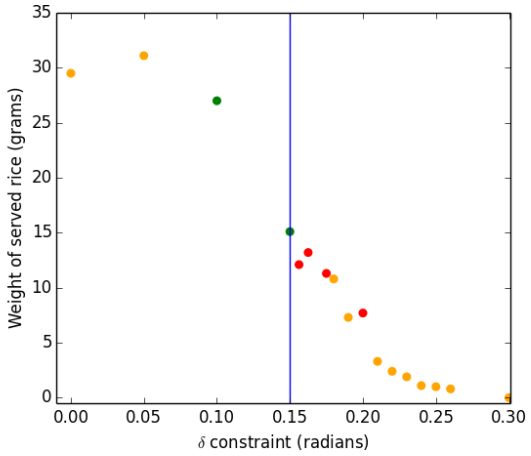


Figure 4.5: The weight of rice transferred to the target bowl at each learning step in the PSPS algorithm, parameterized by the distance constraint parameter δ. Red points are those marked by the teacher as failures. Green points succeeded. Orange points show tests that were run outside of the PSPS algorithm to confirm that the PSPS algorithm had found the correct threshold δ value of 0.15 radians (vertical line)

pute, using finite differences, the total sum of squared second derivatives (the cost function) for the training trajectory and the solution found by PSPS. The training trajectory had a total cost of 9.78 radians/second^2. PSPS's learned trajectory had a total cost of 0.461 radians/second^2, showing a marked improvement over the cost function. To visualize the improvement of the PSPS algorithm's result over the roughness cost, we plot the learned ($\delta = 0.15$) trajectory for each joint in Fig. 4.6. We see from Fig. 4.6 that the PSPS algorithm has indeed been able to improve the smoothness of the joint trajectories, while, as noted in Fig. 4.5, still satisfying the task constraints.

From Fig. 4.6, not only do we see confirmation that the PSPS algorithm is able to smooth out the trajectory, but we also see evidence for why that is the case. We hypothesized that a robot could improve on a kinesthetically taught trajectory if the teacher were insufficiently coordinated to move the joints simultaneously along the desired trajectories. The regions of roughly constant value in various joints in Fig. 4.6 suggest the difficulty in manipulating a high-DOF robot kinesthetically. In particular, they show how generally only a few joints can be easily controlled at a time by someone physically moving the robot to teach it a trajectory, and that frictional forces will tend to keep joints not explicitly manipulated at a constant value. PSPS allows the robot to smooth out the times when a joint is active into regions where the joint was not actively manipulated.

We also note that joint LElbowYaw (red line) was hardly moved during kinesthetic training, and that our experiment in PSPS also kept that joint fixed. The fact that PSPS maintained the constancy of that joint is related to the fact that the cost function

we used for this experiment was defined *in joint space*. This suggests future work where we experiment with either more complicated cost functions or with cost functions defined in configuration space, as that may enable the robot to improve trajectories by manipulating joints that had been held constant.

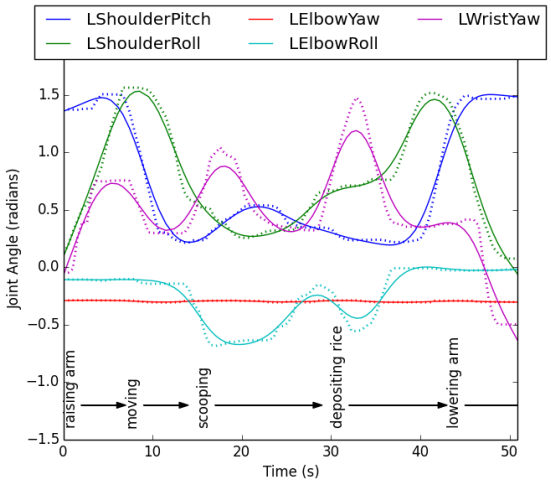


Figure 4.6: A comparison of the demonstration joint trajectories (dotted) and the learned joint trajectories (solid lines). Pepper's joints are defined in Fig. 4.4. General descriptions of motion components are given at the bottom of the chart

4.4 Summary

In this chapter, we considered the challenge of a robot modifying a kinesthetically trained trajectory to better suit the robot's kinematics and cost function. We introduced the PSPS algorithm and showed how it uses feedback from a human teacher to decrease the cost to the robot of performing the trajectory while still completing the assigned task. Our experiments were performed on a real Pepper robot.

Chapter 5

Caprese Salad Case Study

In this chapter, we present our caprese salad plating robot, a UR5 robot with a fork attached to its end-effector. We explain how the robot performs the caprese salad plating task using kinesthetic learning and recorded trajectory replay. The food-acquisition task is approximately rotationally symmetric around the center of the food item. We show how the robot is able to expand its workspace by rotating the recorded pickup trajectories and replaying them in simulation to compute feasible, lower-cost trajectories.

5.1 Caprese Salad Plating

One of the current difficulties in manipulating food is the lack of an accurate model for deformable food. For example, consider the task of plating a caprese salad by picking and placing alternating slices of tomatoes and cheese using a single fork. One strategy for performing the acquisition of slices is to pierce each slice using a vertical approach, tilt the fork to a horizontal position (bending the tomato or cheese slice in the process), and lift the fork with the prongs held horizontally.

This complicated manipulation is shown in Figure 5.1, where the fork is tilted to a horizontal position before being lifted off the blue cutting board. In the example shown, the tomato slice remains on the fork when lifted vertically. We note that in this example, holding the fork prongs vertically (instead of horizontally) releases the tomato from the fork. This has two ramifications. First, we can use this behavior to deposit the tomato slice at the desired location by moving the fork to the desired location before tilting the fork prongs to vertical. Second, this manipulation problem with a ripe tomato slice is complicated enough that vertical skewering and vertical carrying is not sufficient.

While it would be possible to use a more complicated end-effector design to simplify the manipulation process, we are interested in using a multi-purpose tool like a fork that can be also used for other tasks as well. Additionally, we are specifically interested in

(a) Approach with prongs vertical

(b) Rotate fork

(c) Lift with prongs horizontal

Figure 5.1: UR5 robot performing a complicated tomato slice acquisition motion as part of a plating task. A video of the robot plating the caprese salad is available at `https://youtu.be/BsJ47_J4oec`

complicated manipulation strategies for deformable, pierceable, and bendable objects because these manipulation strategies are not as well understood.

Without a deformable, pierceable, bendable physics model of a tomato slice, it is not possible for an learning algorithm to develop this type of pickup strategy in simulation. Instead of having the robot learn this pickup strategy from scratch, we give the robot a single demonstration trajectory and have the robot mimic the training trajectory as closely as possible. This trajectory replay is a type of behavioral cloning, in which a robot learns a policy that imitates training trajectories without learning a reward function [31]. We find that trajectory replay is able to accomplish the desired task using only a single demonstration in the real world. This suggests the power of trajectory replay for manipulation tasks of hard-to-model objects like food.

We implement our trajectory replay of a single demonstration in the following way. First, a training trajectory is recorded on a small Niryo One robot [10] with a fork, and the trajectory of the fork tip is computed in task space based on joint angle recordings, as shown in Figure 5.2. We chose to record on a Niryo One robot because the Niryo One robot is light, small, and easy for a single person to manipulate dextrously. We also tested recording fork trajectories directly on the UR5 robot, but found it was cumbersome to move the relatively large links of the robot.

Second, a vision system identifies the location of a tomato slice relative to the base of a UR5 robot. We use a RealSense camera mounted on the wrist of the UR5 to identify the location of the largest food items on the table. Here, we use the color segmentation strategy explained in Section 2.3.

Finally, the task-space trajectory is translated to align with the tomato slice, and

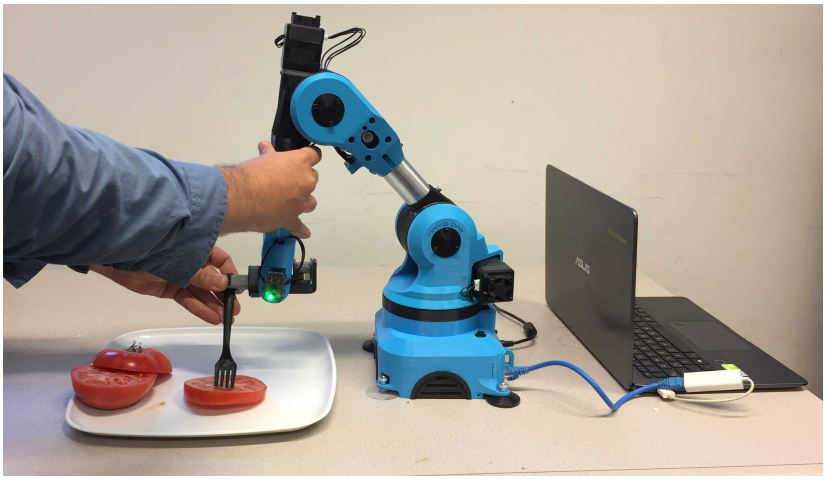
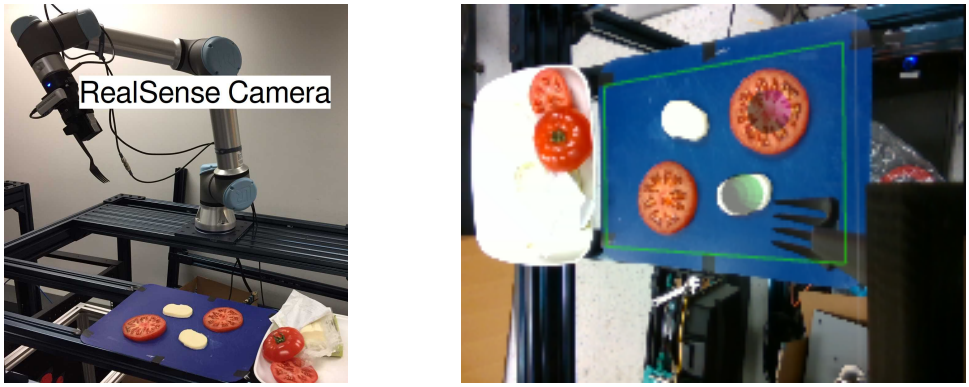


Figure 5.2: In kinesthetic teaching, the robot is manually moved through the motion of picking up a tomato slice



(a) UR5 with wrist-mounted camera (b) Wrist camera view with identified food

Figure 5.3: The vision system using the wrist-mounted camera to localize food items on the table

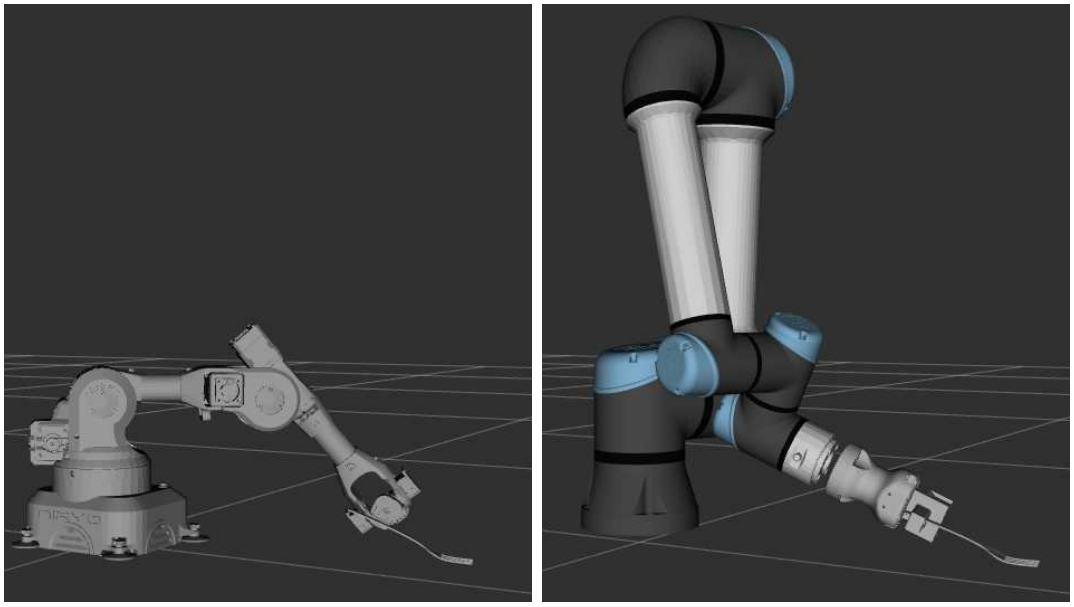


Figure 5.4: The fork can be translated directly toward the base of the Niryo One robot pictured on the left, but the resulting fork positions would lead to self collision on the much larger UR5 robot pictured on the right

we use a continuous inverse kinematics solution to give target joint positions for the UR5 robot to follow. We find that the same recorded trajectory is very often able to successfully pick up the tomato slice and also the mozzarella slice. We did not find it necessary to record a separate trajectory for tomato and mozzarella slices.

However, one downside of this implementation of direct behavioral cloning is the lack of modification to account for the kinematics of the robot. In particular, exactly following the demonstration trajectory may not be feasible for certain tomato slice locations with the given robot kinematics, or may lead the robot to travel through awkward joint trajectories including possibly near singularities with high joint velocities. Figure 5.4 shows how some reasonable fork poses on a Niryo One robot would lead to self-collisions on the larger UR5 robot.

In order to make the power of behavioral cloning more flexible, we present a search-based method to consider rotations of the original trajectory in order to minimize the cost of the trajectory for the particular robot kinematics.

5.2 Approach

We want to modify the demonstrated trajectory to better fit the kinematics of the robot. For this work, we allow the robot to modify the demonstrated trajectory by rotating the trajectory around a vertical axis centered at the tomato slice. We allow this type of rotation as we assume that those rotated trajectories will have an equal

success rate in picking up rotationally symmetric slices. Our goal is to find rotated trajectories that have the lowest cost to perform.

Our algorithm works by taking the desired trajectory to replicate and rotating it around a vertical line passing through the center of the tomato slice. We perform a grid search over N different rotations, evenly dividing the possible rotation angles in $[0, 2\pi)$. For each of these trajectories, we use an analytic inverse kinematics solver [32] to provide possible joint configurations at the start of the rotated trajectory. We then plan through the trajectory and check if that trajectory is collision free. We use MoveIt! [33] to easily handle collision checking, including checking whether the trajectory would collide with the table that the robot arm is mounted on.

We performed our experiments on a UR5 robot, which has six degrees of freedom. Therefore, given a desired trajectory for the fork tip pose, and an initial joint configuration at the start, the joint space trajectory of the UR5 robot is uniquely defined. The above only holds so long as the robot does not hit a joint singularity, but we note that the set of exact joint singularities within the workspace has probability zero for feasible trajectories in task space.

If a trajectory is collision-free, we compute for that trajectory the cost of the trajectory. For this work, we use the Euclidean length of the joint-space path as the cost of the trajectory. This encourages the robot to choose paths that reduce the overall distance traveled by the joints, which in turn reduces the average speed that the joints would need to travel. We consider our cost function to be a measure of the "comfort" of the robot in performing the trajectory. Other more complicated cost functions could include penalty terms for the static torque on the robot joints due to gravity throughout the trajectory or dynamic terms that penalize joint acceleration during the trajectory.

5.3 Results

We find that this approach is able to perform the expected optimizations. We consider a recorded trajectory for moving a fork down, rotating it to pick up a tomato slice, and lifting it (Figure 5.1). If we do not rotate the recorded trajectory, the fork prongs point in the negative X direction when lifting the tomato. The cost of that unrotated trajectory, for different locations of the tomato slice, relative to the base of the robot at (0,0), is shown in Figure 5.5.

When we do not allow for rotations, we find many locations for the tomato slice where the robot is not able to perform the desired trajectory because the resulting trajectory would head outside of the robot's workspace, or would lead to a self-collision. For example, in Figure 5.5, the infeasible area directly to the left of the origin is due to self collisions for poses similar to that displayed on the UR5 in Figure 5.4.

However, when we do allow for the robot to change the behavioral cloned trajec-

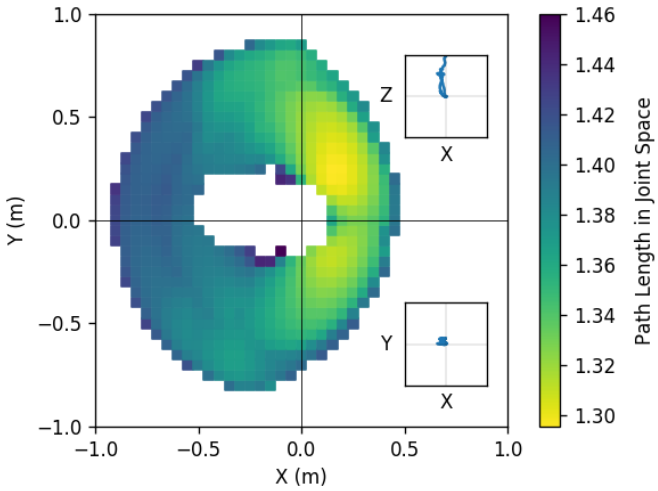


Figure 5.5: We plot the cost of performing the recorded trajectory at various locations without rotating the trajectory. For the given X-Y location on the table, we plot the minimal joint space path length required to perform the recorded trajectory at that location. The recorded trajectory points the fork prongs in the negative X direction. The insets show, to scale, the fork tip path during the pickup trajectory. The base of the robot is at (0,0). No color means that the resulting trajectory is not feasible (either extends outside the reach of the robot, or results in collision)

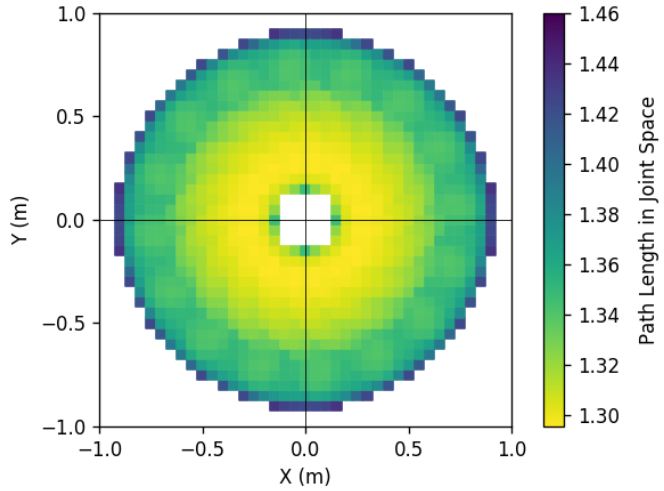


Figure 5.6: We plot the cost of performing the recorded trajectory at various locations, allowing the robot to rotate the trajectory. For the given X-Y location on the table, we plot the minimal joint space path length required to perform the recorded trajectory (rotated by some multiple of $\pi/8$ radians) at that location. The base of the robot is at (0,0). No color means that the resulting trajectory is not feasible (either extends outside the reach of the robot, or results in collision)

tory by rotating the trajectory, we expand the effective workspace of the robot and reduce the cost of performing the trajectory. Additionally, since the UR5 robot's first joint allows for rotations around the vertical axis, we expect that if we allow vertical rotations of the target trajectory as well, then the resulting feasible workspace should also be rotationally symmetric around the vertical axis centered at the robot center. In particular, if we move the target tomato-slice location in a circle centered at the robot base, we expect the robot to be able to rotate the solution trajectory a corresponding amount and find a solution to the new target location with the same cost. For computational efficiency, we discretize rotations of the original trajectory into multiples of $\pi/8$ radians (dividing the circle into 16 components). Accounting for this approximation, we find that the resulting costs do indeed appear to be rotationally symmetric.

We find that the robot can rotate a recorded motion to effectively increase its workspace in order to perform the given task. We also note that in locations that were part of the workspace without rotation, the robot can still choose to rotate the trajectory in order to reduce the cost of performing the trajectory. Using our approach, a new robot with new kinematic properties can efficiently figure out a good rotation of the original trajectory to use to complete the assigned task.

5.4 Summary

In this chapter, we presented our caprese salad plating robot. The demonstration trajectory was trained kinesthetically on one robot (a Niryo One robot with a fork), and then the resulting trajectory is transferred to the UR5 robot. We found that the robot was able to successfully perform the caprese salad plating task. We further showed that the robot could rotate the recorded trajectories and replay them in simulation to compute feasible, lower-cost trajectories.

Chapter 6

Conclusion and Future Work

In this work, we presented our approach to the problem of robots manipulating food, addressing in particular questions about how to train a robot in visual identification, the importance of feedback and error correction strategies, and how to improve kinesthetically trained food manipulation trajectories. As part of this work, we developed vision-based feeding robots and food-plating robots in the real world using Pepper, Niryo One, Kinova MICO, and UR5 robots.

6.1 Conclusion

We explored strategies through which a food manipulation robot is able to use vision to identify the location types of food items that may be presented. If the object is being localized on a plane, the robot can use RGB image-processing techniques to identify the 3D location of the object. We showed the use of color segmentation to train a robot to identify objects. We introduced the idea of "proxy classes," which could be used to quickly train a robot how to recognize a new object based on similar objects it resembles. If the object is not constrained to lie on a plane, we showed how the robot can use an RGB-D camera and a point-cloud registration algorithm to identify the location of the object.

In our research on error correction and feedback, we presented our complete feeding robot implemented on both a Kinova MICO robot and a Niryo One robot. On the Kinova MICO robot, we highlighted the importance of visual feedback to allow more robust feeding execution. By introducing a vision-based approach we can make the system more intelligent and also more autonomous. The developed system is fully capable of feeding a user using a spoon with different types of food like rice or peanuts, and uses visual feedback to ensure spoons are full of food when presented to the user.

We also presented the Parameterized Similar Path Search (PSPS) algorithm, a novel approach to improve kinesthetic trajectory learning for feeding robots, in which the robot iteratively improves the trajectory demonstrated by the caretaker over a

known cost function and requests evaluations from the caretaker to check compliance with unknown task constraints. We analyzed the performance of PSPS in a two-dimensional simulation as well as on a real Pepper robot performing a rice serving task. We found that the PSPS algorithm is able to successfully improve the training trajectory based on feedback on whether the full tested paths satisfied the caretaker's task constraints, even without receiving feedback from the caretaker on the specific times failing trajectories violated task constraints.

Finally, we presented our caprese salad plating robot. We showed how the robot was trained using kinesthetic teaching on a different robot, and how our vision module allowed the robot to detect the desired food items on the table. We showed how the robot could expand its workspace and better accomodate its kinematics by rotating the recorded trajectory in simulation to compute feasible, lower-cost trajectories.

Throughout this work, we approached these problems with an eye to real-world implementation on robots. We were able to successfully manipulate food items ranging from rice and peanuts to large tomato and cheese slices. Our implementations were generic enough to be able to be successfully used on several real-world robots: Pepper, Niryo One, Kinova MICO, and the UR5.

6.2 Future Work

We have shown how visual perception allows our robots to plan manipulations in their environment. In future work, we expect that, once contact is made with the object, then additional sensory modalities can help localize the object during manipulation, such as force, tactile-vibration, and acoustic sensing. Given the effectiveness in the spoon-facing vision feedback in identifying the mass of food on the spoon, future work could investigate using that vision feedback system to control the depth to which the spoon digs into the food, so that the feeding system can alter its serving strategy to accommodate the gradually decreasing level of food in the serving bowl. Additionally, we could more tightly close the feedback loop by gathering feedback throughout the food acquisition motion, rather than just at the end to check for success or failure.

Another area of future work would be to kinesthetically teach multiple trajectories to the robot and have the robot incorporate them into a food manipulation repertoire. This research area could investigate how to average similar trajectories or how to choose which trajectory is most likely to be successful on a given food item.

The PSPS algorithm, while useful independently, also suggests future avenues of research in which results from several runs of the PSPS algorithm could then be fed into learning from demonstration algorithms in order to generalize these lower-cost trajectories onto new environments.

Another area of future work would be to consider more complicated applications of trajectory modification in task space within the framework of the PSPS algorithm.

In this work, we either considered joint space modifications (Chapter 4) or simple rotations of the training trajectory (Chapter 5). More complicated trajectory modification approaches might use STOMP [34] to perform the optimization step of the PSPS algorithm.

We showed how, for rotationally-symmetric tasks, the robot can take advantage of that task symmetry to increase its workspace and decrease task cost by rotating the recorded trajectory. Future work could explore cases where the probability of task success varies based on how much the trajectory is rotated. In that case, we expect to find a tradeoff between the probability of task success and the reduced cost of the rotated trajectory.

Appendix A

Customized Robotic Platform for Manipulation

In order to run experiments and incorporate vision into a feeding robot, we decided to start our research by building our own feeding robot with visual feedback. An image of our complete feeding robot setup with visual feedback is shown in Figure A.1.

We wanted an inexpensive system we could modify as needed, and which was relatively portable, so it could be eventually be transported to potential users in real-world environments. We chose to base our robot on the open-source Niryo One robot arm, and successfully designed and implemented a feeding robot arm that we could use as a test platform for new robot feeding advances. We contribute our original end-effector design that is attached to the Niryo robot arm, as well as the lessons we learned while building a custom open source feeding robot. After success with this custom-built robot described in this appendix, we purchased additional, pre-built Niryo One robots and added our custom end-effector to those.

A.1 Related Work

Similar reviews of the design and creation of feeding robots have been addressed in [35], [4], [36], etc. This appendix extends on those prior works by giving our own learnings generated in the design and construction process, as well as in improvements in the feeding robot design itself by means of a camera to detect successful food acquisition.

A.2 Hardware Design and Construction

The fundamental design of our feeding arm is the Niryo One arm, which is an open-source robotic arm designed by the Niryo corporation.[1] We 3D printed the 3D-printable

[1] `https://niryo.com/`.

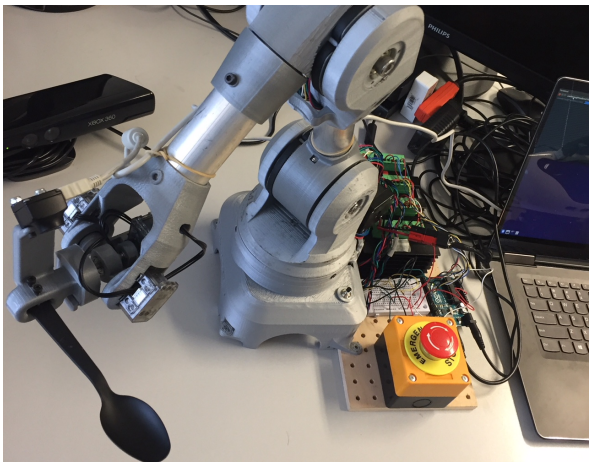


Figure A.1: The complete feeding robot setup

components in PLA plastic. The longest links are constructed from 1.5 inch hollow aluminum tubes which we cut and drilled to the desired length.[2] We found that compressing the tubes from circles into slight ellipses after inserting them into the plastic sockets gave them a very snug fit on the robot. While we were able to find GT2 timing belts online, we were not able to find a circular timing belt of the correct length, so we spliced two ends of a timing belt together by cutting a single step function into each of the two ends, and gluing and taping them together with small steel pins driven through the ridges in the strips to hold them together. We also needed to construct a support spring that is placed inside the shoulder joint of the robot. We did so by wrapping a 0.095 inch high carbon steel wire around 1.5 inch aluminum tube.

We designed and printed a custom end-effector in SolidWorks[3] to transform the Niryo One from a universal manipulator into a spoon-feeding-specific robot arm. The end-effector consists of an attachable mount to the robot arm, an adjustable holder for the spoon handle, and a spoon-facing camera mount. We base the attachable mount shape from STL example attachable mounts provided by Niryo. We were not able to find SolidWorks design files for the attachable mount directly. A bit of experimentation gave us a lightweight design that ensured that the spoon would be seen by the camera but also that the camera would not interfere with the use of the spoon when eating. The spoon holder is shown in Figure A.2.

We attach an RGB camera, in particular an IDS uEye XS camera,[4] to the camera mount on the spoon holder, and wire it via a usb cable along the outside of the arm of the robot to the laptop. We also attach an RGB-D camera, in particular a Kinect V1, to the laptop so that the spoon-feeding robot arm can have situational awareness of its surroundings. The Kinect is placed where it can see the robot so that the camera

[2]Thanks to Ceci Morales for assistance in the tool shop.

[3]`https://www.solidworks.com/`.

[4]`https://www.ids-imaging.us/`.

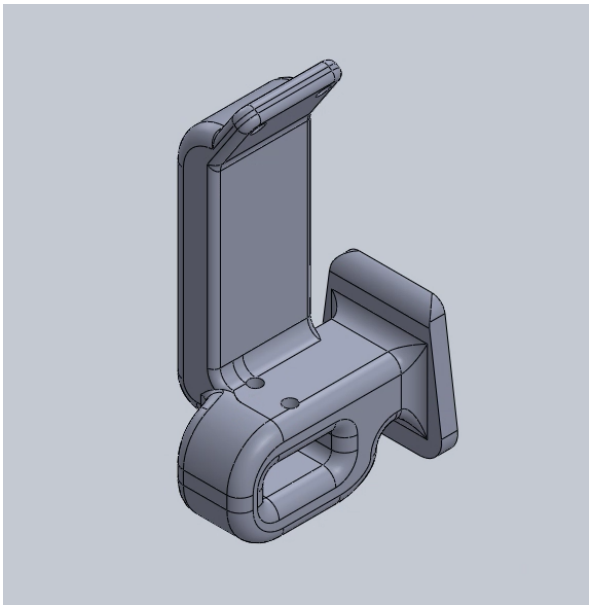


Figure A.2: Spoon holder with camera mount

frame of reference can be easily calibrated to the base of the robot.

A.3 Electronics Design and Construction

The actuation of the stepper motors and the servo motors is different. The servo motors we use are the small XL-320 motors from Dynamixel. They require a 5Vpower supply, and take command packets over a single TTL wire. Furthermore, the servo motors can be daisy chained, so there is a single wire from the Arduino that sends command packets to all the motors. The message in the command packet (syntax specified at the Robotis e-Manual[5]) specifies the target motor and the the requested action. We use an XL320 Arduino helper library[6] to actually construct and send these messages.

For safety purposes, we also want to connect an emergency stop button to the stepper motors to make it trivial to stop them suddenly. Given the types of forces we find when the robot is unladen, the servo motors remain stationary when turned off. We therefore chose for the emergency stop button to simply turn off the power to the servo motors when pushed, as seen in Figure A.3.

The stepper motors are controlled by coordinated pulses along pairs of electromagnetic coils in each motor. Like in the Niryo V1,[7] we pass each of these four wires from each motor all the way to the base of the robot. However, to prevent the overheating

[5] `http://emanual.robotis.com/docs/en/dxl/x/xl320/`.

[6] `https://github.com/hackerspace-adelaide/XL320`.

[7] See blog post on changes made to Niryo One in V2: `https://niryo.com/2017/10/17/niryo-one-introduction-new-upgraded-version-october-2017/`.

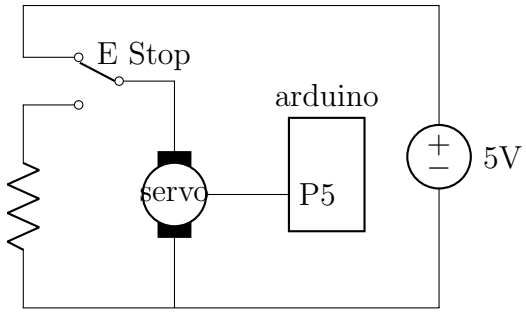


Figure A.3: Power-breaking emergency stop for servo motors

discussed in Niryo V1, we power the motors with stepper drivers with very large heat sinks. The stepper drivers are powered by the 12V power source, and we are able to customize the power consumption of each stepper driver through switches on the side. Different amounts of current on the different stepper drivers results in different max torque that the motor is able to supply. We choose 2A for the first three joints, and only draw 1.5A for the rotation of the forearm joint because we find it does not require as large torques.

For the emergency stop of the stepper motors, we chose not to cut the power to the stepper motors when the emergency stop was pressed. Relative to the force of gravity, the stepper motors have very little friction or similar reactive torque when unpowered. This causes the robot to fall quickly and potentially cause itself injuries, unless it is supported when the power is disconnected. Though the robot is relatively light, there is also some chance that falling could cause user injuries as well. We did not want the robot to fall suddenly when the emergency stop was pressed. Therefore, we chose to implement a software-based stop for the stepper motors when the emergency stop is pressed, as shown in Figure A.4. The Arduino reads the state of the emergency stop signal through an input pin, and stops sending stepper pulse signals when the emergency stop is pressed. A more robust hardware-based solution that still maintains power to the stepper motors would be to entirely cut power to the Arduino. However, since we connect the Arduino to a laptop through its USB port, it is not trivial to disconnect the power to the Arduino since the Arduino is able to draw power through the USB port.

We also wire a disable button to the enable pins of the stepper drivers, shown in Figure A.4. Pressing the disable button acts like cutting the power to the stepper drivers. The stepper motors lose their rigidity completely, and the user is able to re-orient the related joints.

Since our custom-built Niryo does not use encoders on the stepper motors (though the pre-built Niryo One robots we later purchased do), there is no feedback from the stepper motors on the position of the related joints. We instead rely on the user to initialize the joints in a starting position, and the stepper motors are then controlled

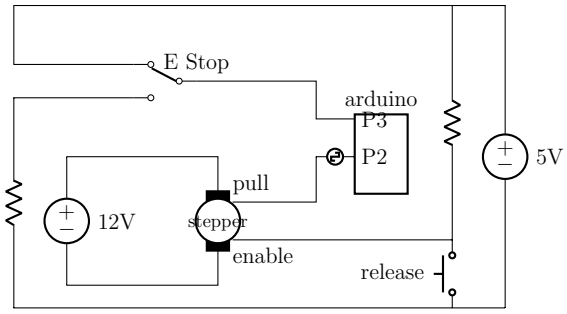


Figure A.4: Software-based emergency stop and disable button for stepper motors

in an open-loop fashion based on the known angular change whenever a pulse is sent to the stepper motor. We manually calibrate the gear ratio between the stepper motor and the joints themselves. This gear ratio tells us how many pulses are necessary to send to the stepper driver in order to move the robot joint a certain angle. We ran a few trials to ensure that the open loop control was reasonable by commanding the second joint to turn 10 degrees five times in the same direction, and then we reset the motor to the starting position, and then commanded the motor to turn 50 degrees in one motion. On visual inspection, comparing pencil marks that were roughly 1mm in width on the outer structure of the joint, the motor was able to accurately replicate its previous motor position on sequential trials. In order for the open loop stepper motor control to work, the robot must be turned on in a known initial starting position. The disable button is especially useful for easily resetting the robot to the starting position.

A.4 Firmware Design and Implementation

All motors are commanded from a single Arduino thread. We found that commands could be dropped if they were sent to the servo motors with too little space between the commands. We did not see any issue with dropped commands if commands were sent with a 10ms pause between them. However, simply adding a 10ms `delay` function call and then sending the next servo command will cause the stepper motors to stop moving during those 10ms. If commands are sent to the servo motors frequently, those pauses cause severe vibrations. Instead, whenever a command is requested for the servo motors, that command is added to a queue. Periodically, the Arduino's single thread of execution will check to see if both 1) there is a request for the servo motors on the queue and 2) it has been more than 10ms since the last request. If so, the next request will be pulled off the servo queue and sent to the motors. In this way, the execution thread is not delayed by the need to send servo commands with a gap between them.

The stepper motors are controlled via pulses from the single execution thread. In order to control all motors simultaneously, the single execution thread maintains a record of when the last pulse was sent to each stepper motor. It then iteratively

runs over the list of motors to check whether it has been more than a certain amount of time since the last pulse, and if so, it sends a pulse to that stepper motor and resets the last recorded pulse time. The time between pulses to each stepper motor depends on the speed that we want the motor to travel. In our firmware, the Arduino receives requested joint angles from the controller. It then compares the requested joint angles to the current joint angles, and determines the required speed to move each joint in order to have each joint achieve the desired joint angles after 150ms. We do put a maximum speed on each joint angle, so some joints may achieve the requested joint states after others. We also add a maximum acceleration for each joint, which leads to smoother motions, but is another reason that joints may not reach the target locations at exactly the same time. For our work, we believe that these motor control specifications are sufficiently smooth and sufficiently precise to satisfy our use case.

The syntax to send joint commands to the Arduino is inspired by the syntax of the iRobot Create[8]. In particular, we first send a "command id" byte, and then send a sequence of bytes which are parameters for that command. In our case we only ever send commands for moving the robot to the joint angles that follow, so our first byte is always the same command id. However, prefixing our commands with this "command id" allows extensibility in the future for us to communicate other commands. Following the "command id" byte corresponding to "move to the following joint positions", we send pairs of bytes (high byte first) that correspond, based on a pre-defined linear transformation, to the desired joint angles. The linear transformation simply maps the range of two bytes (0 to 65535) to the full range of possible joint angles we consider that might be requested for any of the joints (-2π to 2π). The firmware maintains an estimate for the current joint configuration.

A.5 Network Architecture

The components of the robot are in total connected as shown in Figure A.5. The Arduino controls the servo motors directly through byte commands sent over TTL, and controls the stepper motors by means of pulses sent to the stepper drivers. The laptop takes in information via USB from the RGB and RGB-D cameras. The laptop sends byte commands over USB to the Arduino to define the desired joint states, which is then interpreted by the Arduino firmware into motor commands.

[8]The iRobot Create 2 Open Interface is specified at `https://www.irobotweb.com/-/media/MainSite/Files/About/STEM/Create/2018-07-19_iRobot_Roomba_600_Open_Interface_Spec.pdf`

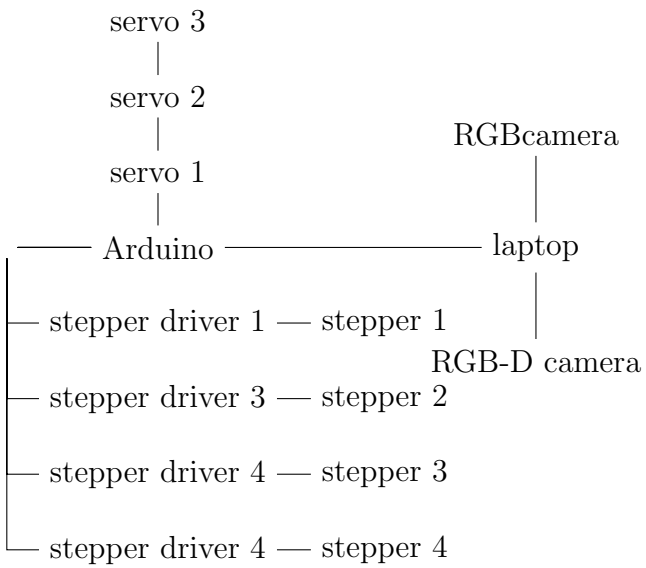


Figure A.5: Network architecture

A.6 Software Design and Construction

We use ROS[9] to organize our robotics code. The primary utility code we wrote for our robot is accessed through a service called `track_pose_service`. This service should be initialized when the robot is in the known initialization position. When initialized, this service moves the robot to a neutral position and then waits for requests. Requests to `track_pose_service` are either desired 6D poses of the end-effector spoon, or are requests to stop the motors from moving. When a request is sent, the `track_pose_service` computes a path from the current joint-state of the robot to a joint-state where the end-effector matches the desired pose. It does this by computing a the requested delta in cylindrical coordinates and a the requested rotational delta in angle-axis representation that would move the tip of the spoon to the desired pose. The reason we work in cylindrical coordinates is to try to avoid joint constraints due to the tip of the spoon getting too close to the base of the robot. By working in cylindrical coordinates, the default path between two points is a linear interpolation of the two points written in cartesian coordinates (r, θ, z). Thus, during the path of the end-effector, the end-effector never gets closer to the z-axis than at the start or end position of the path. Had we used traditional linear interpolation of cartesian coordinates (x, y, z), we might find cases where, when traveling from, say, $(-1, 0, 0)$ to $(1, 0, 0)$, the robot would attempt to move its end-effector through its center at $(0, 0, 0)$, which would lead to a violation of joint constraints. While this reduces the probability of the robot planning a path that violates joint constraints, it is nevertheless better if the code that calls the `track_pose_service` compute for itself a feasible path and

[9] `http://www.ros.org/`.

then send requested poses that are nearby waypoints on that path.

The requested change in the end-effector is converted to changes in joint-angles by means of the Jacobian. The code computes the Jacobian using MoveIt![10] and solves the linear equation relating the desired position and rotation deltas to joint angle deltas:

$$\begin{bmatrix} J_{\text{cyl}} J_c \\ J_r \end{bmatrix} \Delta_\theta = \begin{bmatrix} \Delta_{\text{cyl}} \\ \Delta_{\text{r}} \end{bmatrix} \tag{A.1}$$

where J_c is the cartesian Jacobian relating changes in (x, y, z) coordinates of the end-effector to changes in joint angles, J_{cyl} gives the relationship between changes in cartesian coordinates to changes in cylindrical coordinates, and J_r is the rotational Jacobian relating rotations written in angle-axis form to changes in joint angles. Δ_{cyl} gives the desired change in pose in cylindrical coordinates, and Δ_r gives the desired change in rotation in angle-axis coordinates. Δ_θ gives a linear estimate of the requisite joint angle change.

Given the resulting solution to this equation (solved using regularized least squares to penalize large joint movements), the robot and then move the joints a small fraction of those delta values, and then re-computes. Since this motion-planning process takes repeated small steps toward the target pose, it is no extra computational work to change the target pose, so this code is able to be very reactive to changes to the target pose.

We call the `track_pose_service` from a state machine that runs on a node called `spoon_feeder`, and the specific pose we track depends on the current state of the robot. For example, when the robot is picking up food, then the poses it sends to the `track_pose_service` to track are generated by replaying a recorded spoon trajectory (taken from [37, 24]) and then sending the current pose of the recording. We have a separate ROS service that plays recorded spoon trajectories called `play_trajectory_service`.

If the current robot state is moving the spoon to the mouth of the user, then the pose the robot requests is a function of the current location of the detected mouth of the user. The position of the user's mouth is detected by means of a ROS node we call `DO` (based on DiscriminativeOptimization, our approach to face-tracking). The robot picks a 6D pose in relation to the user's mouth for the spoon to move to, and sends that pose to the `track_pose_service`. The `DO` node takes in inputs from the RGB-D camera by means of a `libfreenect` node[11] to translate messages from the Kinect into ROS messages.

The state machine takes in inputs from the RGB camera to detect whether food has been acquired on the spoon. Images from the RGB camera are converted to ROS messages by means of a `ueye_cam` node[12].

[10] `https://moveit.ros.org/`.

[11] `https://github.com/OpenKinect/libfreenect`.

[12] `http://wiki.ros.org/ueye_cam`

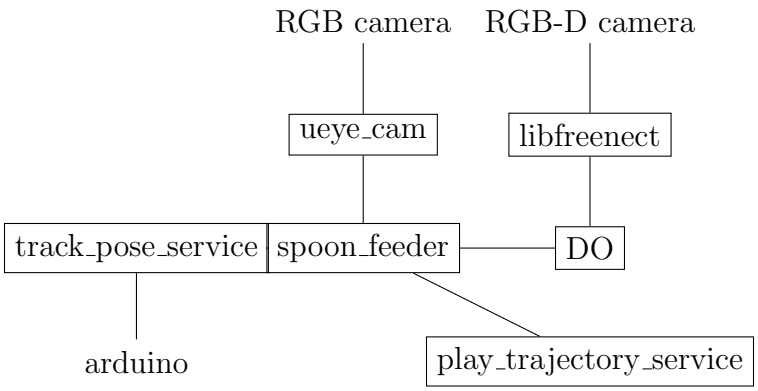


Figure A.6: Architecture of ROS nodes (in boxes) and connections to non-ROS software interfaces

In total, the software architecture is drawn in Figure A.6.

Bibliography

[1] D. Christensen *et al.*, "Prevalence of cerebral palsy, co-occurring autism spectrum disorders, and motor functioning–autism and developmental disabilities monitoring network, usa, 2008," *Developmental Medicine & Child Neurology*, vol. 56, no. 1, pp. 59–65, 2014.

[2] J. B. Reswick, "Development of feedback control prosthetic and orthotic devices," in *Advances in Biomedical Engineering, Volume 2*, pp. 139–217, Elsevier, 1972.

[3] M. Topping, "Early experience in the use of the handy 1 robotic aid to eating," *Robotica*, vol. 11, no. 6, p. 525527, 1993.

[4] M. Topping, "An overview of the development of handy 1, a rehabilitation robot to assist the severely disabled," *Journal of Intelligent & Robotic Systems*, vol. 34, pp. 253–263, 07 2002. Copyright - Kluwer Academic Publishers 2002; Last updated - 2014-08-23.

[5] Camanio Care, "Bestic." `http://www.camanio.com/us/products/bestic/`. Accessed: 2018-05-09.

[6] Performance Health, "Meal buddy." `https://www.performancehealth.com/meal-buddy-systems`. Accessed: 2018-02-18.

[7] Mealtime Partners, Inc., "The mealtime partner dining system description." `http://www.mealtimepartners.com/dining/mealtime-partner-dining-device.htm`. Accessed: 2018-02-18.

[8] Secom Co., Ltd., "My spoon meal-assistance robot." `https://www.secom.co.jp/english/myspoon/`. Accessed: 2018-02-18.

[9] Obi, "Obi, robotic feeding device designed for home care." `https://meetobi.com`. Accessed: 2018-05-09.

[10] Niryo, "Niryo one." `https://niryo.com/niryo-one`, 2019. Accessed: 2019-05-07.

[11] A. Campeau-Lecours, H. Lamontagne, S. Latour, P. Fauteux, V. Maheu, F. Boucher, C. Deguire, and L.-J. C. L'Ecuyer, "Kinova modular robot arms

for service robotics applications," in *Rapid Automation: Concepts, Methodologies, Tools, and Applications*, pp. 693–719, IGI Global, 2019.

[12] AppliedRobotics, "Applied robotics." `https://www.appliedrobotics.com`, 2019. Accessed: 2019-05-07.

[13] S. Davis, J. Gray, and D. G. Caldwell, "An end effector based on the bernoulli principle for handling sliced fruit and vegetables," *Robotics and Computer-Integrated Manufacturing*, vol. 24, no. 2, pp. 249–257, 2008.

[14] D. Gallenberger, T. Bhattacharjee, Y. Kim, and S. S. Srinivasa, "Transfer depends on acquisition: Analyzing manipulation strategies for robotic feeding," in *2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 267–276, IEEE, 2019.

[15] L. V. Herlant, *Algorithms, implementation, and studies on eating with a shared control robot arm*. PhD thesis, Carnegie Mellon University, 2018.

[16] M. de Jong, K. Zhang, A. M. Roth, T. Rhodes, R. Schmucker, C. Zhou, S. Ferreira, J. Cartucho, and M. Veloso, "Towards a robust interactive and learning social robot," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 883–891, International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[17] A. Candeias, T. Rhodes, M. Marques, J. P. Costeira, and M. Veloso, "Vision augmented robot feeding," in *European Conference on Computer Vision*, pp. 50–65, Springer, 2018.

[18] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime multi-person 2d pose estimation using part affinity fields," in *CVPR*, 2017.

[20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[21] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *arXiv preprint arXiv:1506.01497*, 2015.

[22] R. Diankov, *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[23] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[24] T. Bhattacharjee, H. Song, G. Lee, and S. S. Srinivasa, "Food manipulation: A cadence of haptic signals," *arXiv preprint arXiv:1804.08768*, 2018.

[25] T. Rhodes and M. Veloso, "Robot-driven trajectory improvement for feeding tasks," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2991–2996, IEEE, 2018.

[26] B. D. Argall, B. Browning, and M. Veloso, "Learning robot motion control with demonstration and advice-operators," in *2008 IEEE International Conference on Robots and Systems (IROS)*, pp. 399–404, IEEE, 2008.

[27] P. J. Green and B. W. Silverman, *Nonparametric regression and generalized linear models: a roughness penalty approach*. CRC Press, 1993.

[28] C. H. Reinsch, "Smoothing by spline functions," *Numerische mathematik*, vol. 10, no. 3, pp. 177–183, 1967.

[29] Aldebaran/SoftBank Robotics, "Pepper - documentation." `http://doc.aldebaran.com/2-5/home_pepper.html`. Accessed: 2018-02-27.

[30] D. Park, H. Kim, Y. Hoshi, Z. Erickson, A. Kapusta, and C. C. Kemp, "A multimodal execution monitor with anomaly classification for robot-assisted feeding," in *2016 IEEE International Conference on Robots and Systems (IROS)*, 2017.

[31] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters, *et al.*, "An algorithmic perspective on imitation learning," *Foundations and Trends® in Robotics*, vol. 7, no. 1-2, pp. 1–179, 2018.

[32] R. Diankov, *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[33] I. A. Sucan and S. Chitta, "Moveit!," *Online at http://moveit. ros. org*, 2013.

[34] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *2011 IEEE international conference on robotics and automation*, pp. 4569–4574, IEEE, 2011.

[35] W.-K. Song and J. Kim, "Novel assistive robot for self-feeding," in *Robotic Systems-Applications, Control and Programming*, InTech, 2012.

[36] S. Ishii, S. Tanaka, and F. Hiramatsu, "Meal assistance robot for severely handicapped people," in *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1308–1313, IEEE, 1995.

[37] T. Bhattacharjee, H. Song, G. Lee, and S. S. Srinivasa, "A dataset of food manipulation strategies," 2018.