CS 397 Final Report

Semester: Spring 2020
Advisor: Professor Kris Hauser
Rohan Prasad (netid: rprasad3)

I. Introduction

Robots have become increasingly prevalent in the modern era. As society moves into automating many jobs that have been traditionally performed by humans, the world has seen the introduction of robots into previously human-exclusive fields. Hospitals are no exception to this automation, and now more than ever as the world is poised for industrial disruption post COVID-19, robotic introduction into the field of nursing has many benefits.

This project serves to create a tele-operated nursing robot (abbreviated as TRINA) to participate in the Xprize ANA Avatar Competition, where teams across the world will aim to create an autonomous nursing assistant, able to assist nurses and doctors with some of the more mundane tasks associated with hospital processes.

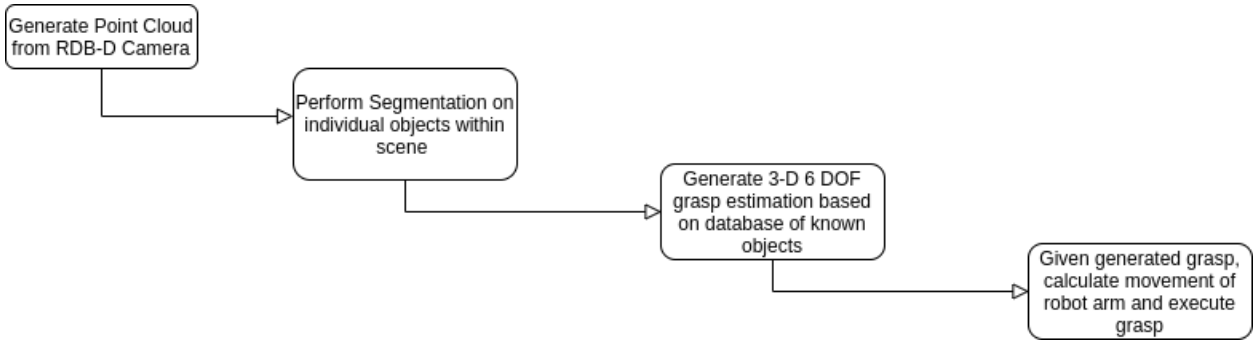
As a member of the autonomy subteam, my task this semester was to identify and define an end to end computer vision pipeline, and implement a specific segment of that pipeline, to be further discussed in this report.

This report serves as documentation and summary of my progress during the - albeit short - spring semester, and will aid either myself or others who continue this project in coming semesters. The spring 2020 semester was notably moved online due to COVID-19 halfway through, which explains the transition to a fully virtual mode of operation towards the latter portion of the report.

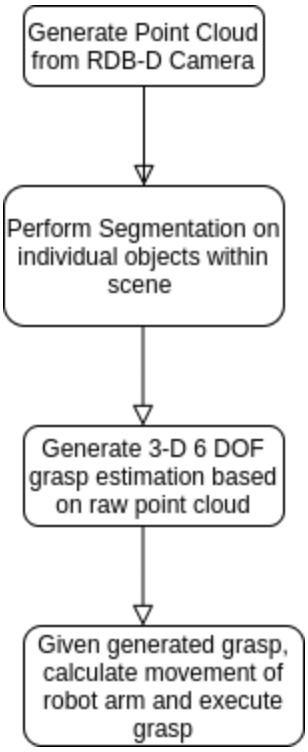
II. Project Description

A. Computer vision pipeline

The initial computer vision pipeline was determined as the following. An RGB-D camera is to be used to get a point cloud of a certain scene, or read and process scenes as the robot moves. From this point cloud, objects should be segmented for further processing. Then, using a model to calculate grasp generations, known objects could be determined and a grasp would be generated from prior knowledge of how to grasp that object. Finally, given the generated grasp, the robot's arm and gripper would be moved using the motionAPI, completing the grasp of a given object.



After further consideration, we determined that generating grasps from a dataset of known objects would be less reasonable to implement in a true hospital environment; all it takes is one object for the robot to fail, and the viability of robots in a high-intensity environment such as a hospital drops significantly. Due to this, we decided to modify the pipeline to generate grasps based on a raw point cloud. With this functionality, regardless of the object, the robot would be able to identify a grasp and hopefully execute it, which allows for a widespread application of the robot, rather than being limited to a few hospital items. The updated pipeline is as follows.



III. My assignment in this pipeline was to develop an interfaceable and accessible means to generate 3-D 6 DOF grasps from raw point clouds. Initially, I tried to interface with the Intel Euclid camera. At first, this seemed to be ideal, as this camera allowed for various streams, not just a raw depth stream. After accessing the camera stream, it was determined that the Euclid's dependence on ROS and it's subscriber/publisher scheme even for stream accessing wasn't ideal, and I began using the zed and realsense cameras already established in the lab.

A. Open3D file conversion

Oftentimes in point cloud datasets, images are found in image format (.object files). In order to address these images and use the equivalent in point cloud format, Open3D was used to facilitate

a conversion between file types, and ensure the image is able to be sampled. Below is the python script to convert .obj files into either .ply or .pcd files. It should be called in the command line followed by the filepath of the .obj, and the desired output format.

```
import open3d as o3d
import os, sys
import numpy as np


def check_path(path, should_exist):
        """ Check that a path (file or folder) exists or not and return it.
        """
        path = os.path.normpath(path)
        if should_exist != os.path.exists(path):
                msg = "path " + ("does not" if should_exist else "already") + " exist: " + path
                raise argparse.ArgumentTypeError(msg)
        return path

filepath = ""
if len(sys.argv) > 1:
   filepath = str(sys.argv[1])

if(filepath.endswith('.obj')):
        filepath = filepath[:-4]

fp = check_path(filepath, true)

mesh = o3d.io.read_triangle_mesh(filepath + ".obj")
pcd = o3d.geometry.PointCloud()
pcd.points = mesh.vertices
pcd.colors = mesh.vertex_colors
pcd.normals = mesh.vertex_normals

mesh.paint_uniform_color([1, 0.706, 0])
o3d.visualization.draw_geometries([mesh])
print(np.asarray(mesh.vertices))


o3d.io.write_point_cloud(filepath + ".pcd", pcd)
```

B. Model Determination

After determining the input metric for the images that the robot would provide and what could be tested upon, I read a collection of survey papers and papers about grasp generations. At first, I tried a few models which generated grasps from a collection of objects, which was very accurate on some select objects, but were very inaccurate on objects not recognized with the model. This might be remedied by adding certain images to the test set, but an issue arises with the determination of more items that should be added and trained upon when determining a new/unrecognized object.

The first raw point cloud to grasp generation model that was considered was a hybrid model. This model trained on the YCB dataset of point clouds, and trained for both the 3-D grasps based on specific objects, and also on raw point clouds. This model, however, didn't follow a ROS based model, and was written and interfaceable with python 3, which clashes with the motion API that was prior written to control the robot. Due to this, I decided against using this model in preliminary testing of grasp generations.

I determined the ideal model for lab purposes was one that was operable on cluttered scenes and compatible with parallel jaw grippers (read: 2 finger robot hand). This model took advantage of various dependencies that were also compatible with the motionAPI, allowing for easy integration with pre-written code. Below is an image of the dependency list that had to be edited to select the distribution to install.

```
#include <vector>
#include <highgui.h>
#include <Eigen/Dense>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/opencv.hpp>

#include <opencv2/core/core.hpp>


#include <gpd/candidate/hand_set.h>
#include <gpd/descriptor/image_geometry.h>
```

After building and training this model with PCL, which provided processing for point clouds, I was able to modify the generated grasps to improve the quality of grasps through a file which was incorporated into the model. This config file incorporated a variety of physical preferences as well as virtual preferences. Below is the file with what I believe are fairly ideal values for the generator. This will be modified when the robot is available for testing, but for

now, these values were deemed appropriate.

```
# Grasp candidate generation
#   num_samples: number of samples to be drawn from the point cloud
#   num_threads: number of CPU threads to be used
#   nn_radius: neighborhood search radius for the local reference frame estimation
#   num_orientations: number of robot hand orientations to evaluate
#   num_finger_placements: number of finger placements to evaluate
#   hand_axes: axes about which the point neighborhood gets rotated (0: approach, 1: binormal, 2: axis)
#              (see https://raw.githubusercontent.com/atenpas/gpd2/master/readme/hand_frame.png)
#   deepen_hand: if the hand is pushed forward onto the object
#   friction_coeff: angle of friction cone in degrees
#   min_viable: minimum number of points required on each side to be antipodal
num_samples = 30
num_threads = 4
nn_radius = 0.01
num_orientations = 8
num_finger_placements = 10
hand_axes = 2
deepen_hand = 1
friction_coeff = 20
min_viable = 6

# Filtering of candidates
#   min_aperture: the minimum gripper width
#   max_aperture: the maximum gripper width
#   workspace_grasps: dimensions of a cube centered at origin of point cloud; should be smaller than <workspace>
min_aperture = 0.0
max_aperture = 0.085
workspace_grasps = -1 1 -1 1 -1 1

# Filtering of candidates based on their approach direction
#   filter_approach_direction: turn filtering on/off
#   direction: direction to compare against
#   thresh_rad: angle in radians above which grasps are filtered
filter_approach_direction = 0
direction = 1 0 0
thresh_rad = 2.0

# Clustering of grasps
#   min_inliers: minimum number of inliers per cluster; set to 0 to turn off clustering
min_inliers = 0

# Grasp selection
#   num_selected: number of selected grasps (sorted by score)
num_selected = 5

# Visualization
#   plot_normals: plot the surface normals
#   plot_samples: plot the samples
#   plot_candidates: plot the grasp candidates
#   plot_filtered_candidates: plot the grasp candidates which remain after filtering
#   plot_valid_grasps: plot the candidates that are identified as valid grasps
#   plot_clustered_grasps: plot the grasps that after clustering
#   plot_selected_grasps: plot the selected grasps (final output)
plot_normals = 1
plot_samples = 0
plot_candidates = 1
plot_filtered_candidates = 0
plot_valid_grasps = 0
plot_clustered_grasps = 0
plot_selected_grasps = 1
```

Text Editor

After selecting these parameters and setting up the cmake file, I was able to run the model on converted .ply and .pcd files that I had converted through Open3D from MIT's textured model database.

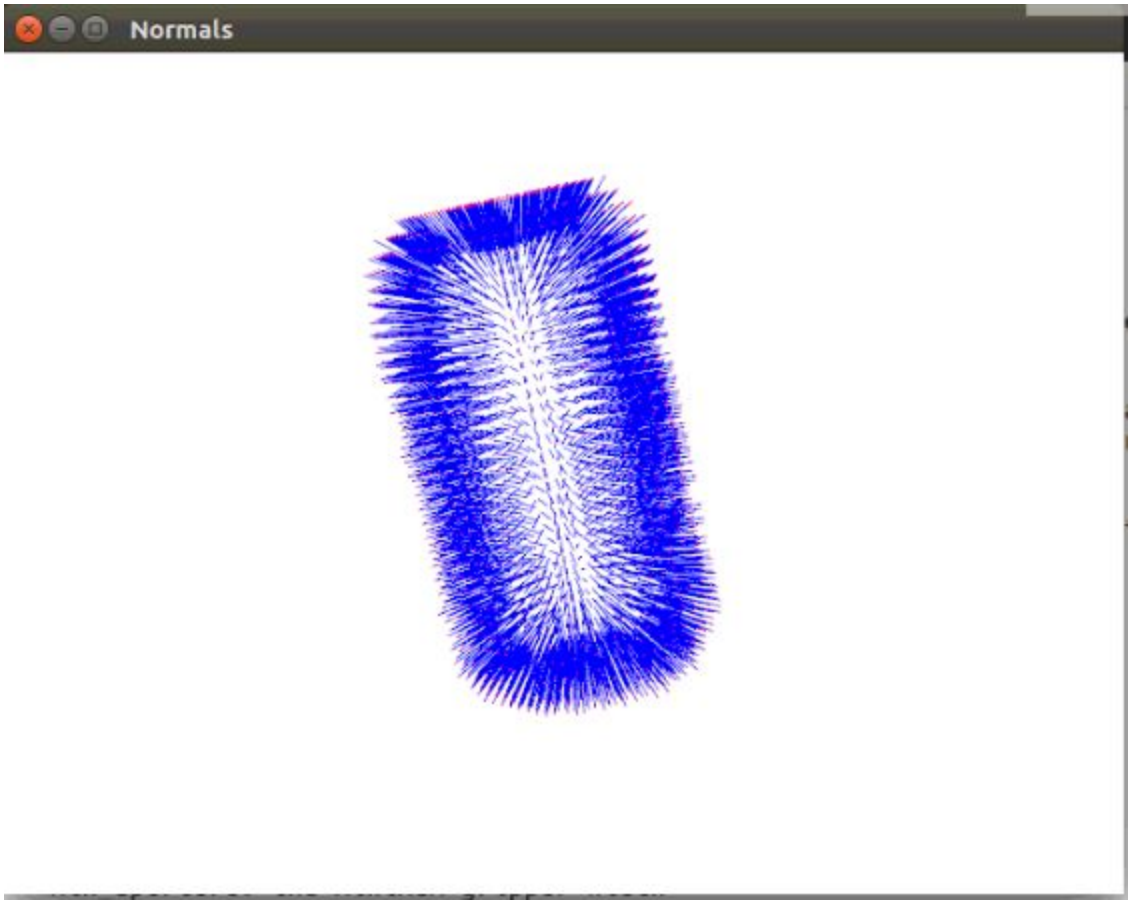
C. Grasp Generation

Below are the results of the grasp generation run on items from MIT's textured items database.

```
(./detect_grasps ../cfg/eigen_params.cfg ./krylon.pcd)
```

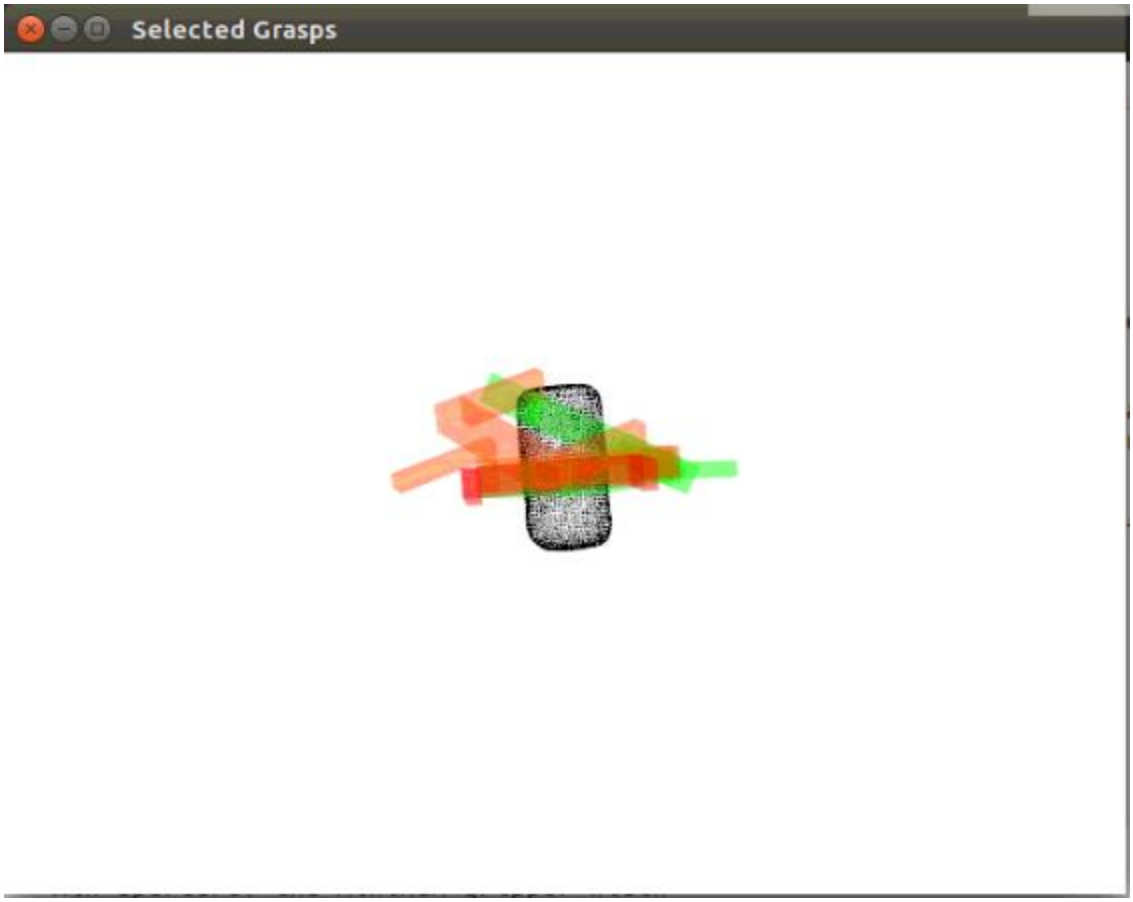
```
Loaded point cloud with 4467 points
============ HAND GEOMETRY =====================
finger_width: 0.01
hand_outer_diameter: 0.12
hand_depth: 0.06
hand_height: 0.02
init_bite: 0.01
================================================
============ PLOTTING ========================
plot_normals: true
plot_samples false
plot_candidates: true
plot_filtered_candidates: false
plot_valid_grasps: false
plot_clustered_grasps: false
plot_selected_grasps: true
================================================
============ CLOUD PREPROCESSING =============
voxelize: true
voxel_size: 0.003
remove_outliers: false
workspace: -1.00 1.00 -1.00 1.00 -1.00 1.00
sample_above_plane: false
normals_radius: 0.030
refine_normals_k: 0
================================================
============ CANDIDATE GENERATION ============
num_samples: 30
num_threads: 4
nn_radius: 0.01
hand axes: 2
num_orientations: 8
num_finger_placements: 10
deepen_hand: true
friction_coeff: 20.00
min_viable: 6
================================================
============ GRASP IMAGE GEOMETRY ===============
volume width: 0.1
volume depth: 0.06
volume height: 0.02
image_size: 60
image_num_channels: 15
================================================
NET SETUP runtime: 0.0695345
============ CLASSIFIER ======================
model_file:
weights_file: ../models/lenet/15channels/params/
batch_size: 1
================================================
============ CANDIDATE FILTERING =============
candidate_workspace: -1.00 1.00 -1.00 1.00 -1.00 1.00
min_aperture: 0.0000
max_aperture: 0.0850
================================================
============ CLUSTERING ======================
min_inliers: 0
================================================

Processing cloud with 4467 points.
Voxelized cloud: 3366
Calculating surface normals ...
```

The model then produces three pcd viewers with different viewpoints. The first of the three displays the surface normals of the object, which will be in turn used to calculate the grasp.



Then, the model displays all possible generated grasps. These are grasps that aren't necessarily viable, but are generally possible given the surface normals of the item.

The model then classifies the viable grasps and displays them in another window, and ranks them by viability, differentiating grasps of lower viability with different colors. In the below picture, the top five most viable grasps are shown:



The grasps are then stored in a text file as numerical values, and can be passed through the pipeline to interface with the motion API to physically move the robot.

IV. Klampt Simulation

The next steps in the pipeline process was to test the grasp generation on point clouds generated by a Klampt simulation to determine if the model would be viable in the lab. The simulation isn't an end-all be-all metric to determine if the model will work physically, but usually hints towards models that will work better than others. The following code was used to simulate the robot in Klampt and continuously receive point cloud readings. I was able to convert these points to point cloud through the klampt function "camera_to_points", which provided a compatible file

with the generation. Through this, I was able to sample some point clouds from the klampt simulation with the model, which was hopefully a sign of viability physically in the lab.

```
import klampt
import time
import math
from klampt import vis
from klampt import io
from klampt.math import so2
from klampt.model import sensing
import matplotlib.pyplot as plt

from motion import *

def interpolate(a, b, u):
    return (b-a)*u + a

def lidar_to_pc(robot, sensor, scan):
    x, y, theta = robot.base_state.measuredPos

    angle_range = float(sensor.getSetting("xSweepMagnitude"))

    rv = klampt.PointCloud()

    for i in range(len(scan)):
        u = float(i)/float(len(scan))
        # If period = 0, measurement sweeps over range of [-magnitude,magnitude]
        angle = interpolate(-angle_range, angle_range, u)
        pt = [math.cos(angle), math.sin(angle)]
        pt[0] *= scan[i]
        pt[1] *= scan[i]
        pt[0] += 0.2    # lidar is 0.2 offset (in x direction) from robot
        pt = so2.apply(theta, pt)
        pt[0] += x
        pt[1] += y
        pt.append(0.2)  # height is 0.2

        rv.addPoint(pt)

    return rv

robot = Motion(mode = "Kinematic", codename="anthrax")

leftTuckedConfig = [0.7934980392456055, -2.541288038293356, -2.7833543555, 4.664876623744629, -0.049166981373, 0.0973691940307617 2]
leftUntuckedConfig = [-0.2028,-2.1063,-1.610,3.7165,-0.9622,0.0974] #motionAPI format
rightTuckedConfig = robot.mirror_arm_config(leftTuckedConfig)
rightUntuckedConfig = robot.mirror_arm_config(leftUntuckedConfig)

robot.startup()

# reset arms
robot.setLeftLimbPositionLinear(leftUntuckedConfig,5)
robot.setRightLimbPositionLinear(rightUntuckedConfig,5)

world = robot.getWorld()

vis.add("world", world)
vis.show()

sim = klampt.Simulator(world)

lidar = sim.controller(0).sensor("lidar")
left_cam = sim.controller(0).sensor("left_hand_camera")
right_cam = sim.controller(0).sensor("right_hand_camera")
vis.add("lidar", lidar)
vis.add("left_cam", left_cam)
right_cam = sim.controller(0).sensor("right_hand_camera")
vis.add("lidar", lidar)
vis.add("left_cam", left_cam)
#vis.add("right_cam", right_cam)

#time.sleep(3)

start_time = time.time()
while True:
    lidar.kinematicSimulate(world, 0.01)
    left_cam.kinematicSimulate(world, 0.01)
    #right_cam.kinematicSimulate(world, 0.01)

    measurements = lidar.getMeasurements()
    lidar_pc = lidar_to_pc(robot, lidar, measurements)
    vis.add("pc", lidar_pc)

    # rgb image, depth_image
    lc_rgb, lc_depth = sensing.camera_to_images(left_cam)

    # Get point cloud. For some reason triggers an assertion for me in Python2
    # with the latest version on Klampt Master
    left_cam_pc = sensing.camera_to_points(left_cam, points_format='Geometry3D', all_points=True, color_format='rgb')
    left_cam_pc.saveFile("/home/avatrina-gpu/rohan/gpd/gpd/build/temp.pcd")

    if time.time() - start_time < 5:
        robot.setBaseVelocity([0.3, 0.15])
    else:
        robot.setBaseVelocity([0.0,0.5])
    time.sleep(0.01)
```

The Klampt simulation at times provided an empty point cloud, and I'm not sure the reason for this. It might be variation in the simulation, or it might be an inaccurate conversion of points. Regardless of the outcome, it's something I'd like to investigate when back in the lab physically. Due to COVID-19, this was the extent of the progress I was able to make this semester. Hopefully, I'll be able to make more progress in person post-pandemic; the future scope of the project is below.

V. Conclusion and future scope

The preliminary infrastructure and interfacing code for the grasp generation is complete. However, a script still must be written to incorporate the Klampt simulation with the model; currently I'm running both independently. This project provided me with valuable insight into writing code to interface with stand-alone modules, and introduced me to working with computer vision in a lab environment (and remotely!), and has been a very valuable experience. The next steps for this project and the pipeline is to test the model on physical objects in the lab using either the zed camera or the realsense camera, and generating grasps that can be used to move the robot to the specified positions. I hope to accomplish this in the fall, or whenever post-pandemic might be.

VI. Works Cited

Hauser, Kris. *Robust Contact Generation for Robot Simulation with Unstructured Meshes*. motion.pratt.duke.edu/papers/ISRR2013-RobustContact.pdf.

ten Pas, Andreas. *Grasp Pose Detection in Point Clouds*. arxiv.org/pdf/1706.09911.pdf.