1 — Project Intro and Goals

For the second project in this course, your task will be to design an algorithm for predicting good grasping approach configurations for a robotic end effector using machine learning techniques and working directly from pointcloud data.

For this project, we will give you pointcloud data of 3 cluttered scenes containing 5 individual objects on a tabletop. The objects correspond to several of the target objects from project 1. For these 3 scenes, we will also give you corresponding *launch files* which will produce replications of the real world scenes in robotic simulation software. For this we make use of PRACSYS, a software package developed and used by Dr. Kostas Bekris and his students in their research. The simulation will give you some limited feedback about collisions and trajectory endpoints (discussed later), and your job will be to use machine learning techniques to predict good grasping approach configurations based on this feedback from the simulator.

The main goals of this project are:

1. Learn how to use a robotic simulator
2. Learn to evaluate grasps based on feedback from the simulator
3. Leverage this feedback and employ existing image processing and machine learning methods to predict good grasps in order to bypass the computationally expensive process of grasp simulation

Once you have developed your solution, you should run your code on scene4 and scene5, where you do not have access to a simulated scene, and should store the predicted good grasps provided by your solution. You will then be asked to schedule a brief demo with the TA where he will have access to the corresponding simulated scenes and will run your produced grasps.

Note: As opposed to the first project, this project will require you to work with and base your approach heavily on the depth channel of the pointcloud data

2 — Preparation and Installation

A large portion of this project of course deals with the processing of pointcloud data from a kinect2 camera. As such, it is recommended that you utilize one of the existing available and popular libraries such as pcl or opencv.

`http://opencv.org/`
`http://pointclouds.org/`

Additionally, in order to make use of the robotic simulator code we've made available, you will need access to a machine with Ubuntu 14.04 or newer installed.

If you have not previously installed/worked with ROS (robotic operating system), it is also necessary that you install ROS-indigo in order to work with the simulator. There are also a good variety of introductory tutorials available on this website, which you are welcome to make use of.

`http://wiki.ros.org/indigo/Installation/Ubuntu`

Once ROS has been successfully installed and before installing PRACSYS, it is necessary to first install a few dependency packages:

```
sudo apt-get install ros-indigo-cmake-modules
     ros-indigo-fcl libopenscenegraph-dev
```

Once ROS and the dependencies have been installed, please follow the installation instructions for installing PRACSYS here:

`https://bitbucket.org/colinmatthew/rl_pracsys/overview`

You should make sure and follow the installation instructions at this location for installing (1) the pracsys simulation code located in this repository, and (2) the rl_models repository which hosts the meshes required by the simulation code. You can also find instructions here on how to use PRACSYS and how to interact with the simulator.

Once downloaded, can find the pointcloud data as well as the list of target grasping objects for each scene in the "rl_data" folder.

3 — PRACSYS Robotic Simulator

Once installed, the grasping simulator may be launched by running any of the following commands:

```
roslaunch manipulation robotlearn_scene1.launch
roslaunch manipulation robotlearn_scene2.launch
roslaunch manipulation robotlearn_scene3.launch
```

In each of these files you will see a line that specifies the *grasp_file* location for the grasping visualization. You should update this line to point to a file containing your generated grasps throughout the project. The structure of your generated grasping configuration files should mimic the structure of the existing data provided in the *dummy_data.yaml* file.

Running the simulation you will see the scene as it appears in the pointcloud image along with a 3-finger Reflex end effector which will simulate the grasps fed into the visualizer.

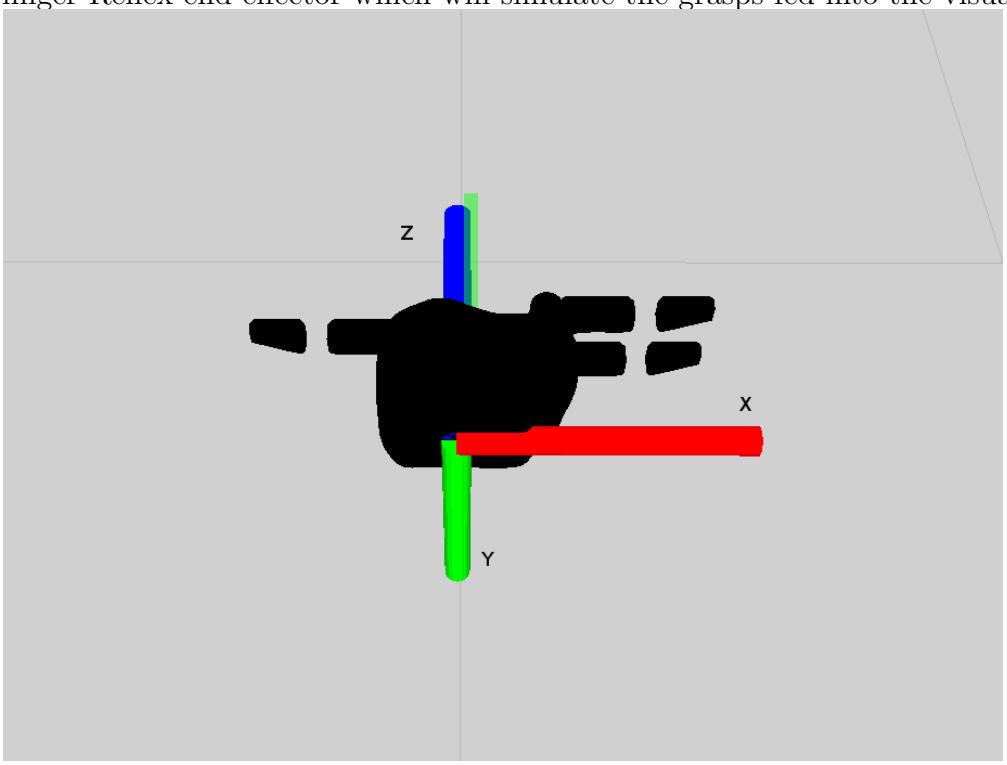

Figure 1: The Reflex end effector for which you will be learning a mechanism to produce configurations. The principal axes are shown in the image.

The simulation we have set up takes an initial configuration as a 7-dimensional <position(x,y,z) quaternion(x,y,z,w)> configuration, where the position should be 10-30cm away from the object initially. It then simulates a trajectory toward the object in 0.1cm increments and gives feedback in the console when the individual steps result in collision (which is undesirable and should be avoided). Pressing "Tab" will run a single configuration/trajectory. Pressing "Enter" will run all input configurations in a loop. At the end of each trajectory, the simulator will output the furthest collision-free configuration of the end effector along the trajectory.

As an important additional note, when defining configurations for this end effector in "camera frame", you should apply the following transformation between the global "world frame" and "camera frame" in order to visualize and simulate these configurations correctly:

Translation<xyz>: 0.0, 0.0, 0.8
Rotation<xyzw>: -0.5, 0.5, -0.5, 0.5

A minimal Python example for applying this transformation is as follows:

```
import tf
import numpy as np

def camera_to_world( quat_obj, trans_obj ):
    quat_cam = tf.transformations.quaternion_from_euler(1.57, 3.14, 1.57)
    trans_cam = np.array([0, 0, 0.8])
    trans_obj = np.array(trans_obj)

    quat_res = tf.transformations.quaternion_multiply(quat_cam, quat_obj)
    rot_obj = tf.transformations.quaternion_matrix(quat_cam)
    rot_obj = rot_obj[:3,:3]
    rot_obj_t = rot_obj.T

    trans_res = trans_obj.T.dot(rot_obj_t)
    trans_res = trans_res + trans_cam
    result = np.append(trans_res, quat_res)

    return result
```

Which yields the following, where *quat_obj* is the quaternion of the object in camera frame and *trans_obj* is the translation in the same frame. The result is the 7-dimensional <translation, rotation> configuration of the object in world frame:

```
In [15]: quat_obj = (-0.5,-0.5,0.5,0.5)
In [16]: trans_obj = (0.5,0.5,0.5)
In [17]: x = camera_to_world(quat_obj, trans_obj)

In [18]: print x
[ 0.49920399 -0.49960009 0.2988063 -0.49920367 0.49999937 0.500398
  0.500398 ]
```

The reason that this transformation is necessary is because the frame of the camera is not located at the origin of the world frame for simulation, as is often the case in real robotic

systems. And in the case of a stationary camera, we can represent this as a fixed offset which must then be applied to configurations in the camera frame in order to transform these configurations to the global coordinate system.

4 — Project Steps

Step 1: Generate a very large number of random initial hand configurations from each image, and run them through the simulator. You can choose any number you think is adequate for your learning algorithm. Save the final hand configurations obtained from the simulator's feedback.

Step 2: Come up with a scoring function that takes as input an RGB-D image and a final hand configuration and returns a number that indicates how likely this grasp will succeed if the robot closes its fingers in the final hand configuration. It's up to you to devise a descriptive scoring function and you may choose any metric you like, but *it should make sense for this task and goal*. For example, the grasps that seem the most robust should have the highest score, and a grasp that certainly fails should have the lowest score.

You can find an example of a scoring function in a recent paper here (pages 1 & 2):

`http://www.cs.rutgers.edu/~ab1544/papers/AbdeslamAAAI2014.pdf?attredirects=0`

Step 3: Use the scoring function to find the best grasp among those generated in Step 1. For each image, you should visualize the best grasp and it should make sense.

Step 4: Use the grasps generated in Step 1 and labeled by their scores in Steps 2 & 3 as training data. The training data is used to learn a function that takes as input an RGB-D image and an initial grasp configuration and returns a score without having to simulate the full trajectory and check for collisions. This learned function should run much faster than the simulator and should be used to replace the simulator.

Step 5: On a new scene, described by an RGB-D image, find the best possible grasp (initial hand configuration) using the simulator. Find the best grasp using the learned function. Compare the two answers.

5 — Deliverables and Grading

Each team will be graded on three criteria according to the outline below.

1. **50%** - Submission of a written report – suggested 3-4 pages – describing (1) your approach to processing the data, (2) which learning methods you tried, (3) which worked best, (4) your chosen scoring function, (5) statistics showing your learned

grasping strategy vs. random selection using your scoring function also **due April 24th at 11:59pm**.

2. **25%** - Submission of your code to the Sakai assignment, **due April 24th at 11:59pm**.
3. **25%** - After the due date, you will be requested to make a short 5-minute appointment with the TA in order to demo your learned grasping strategy on the simulations for scenes 4 and 5.