EECS 106B : Lab #2

Spring 2018

Yichi Zhang, Jingjun Liu, Jiarong Li
27005775, 3033483513, 3033483511

March 6, 2018

Section 1

Video

https://youtu.be/STlOD2JoFHM

Section 2
Methods

A grasp is in force closure when nger forces lying in the friction cones span the space of object wrenches

$$G(FC) = \mathbb{R}^p$$

where friction cone is defined as

$$FC_{c_i} = \left\{ f \in \mathbb{R}^4 : \sqrt[2]{f_1^2 + f_2^2} \leq \mu f_3, f_3 > 0, \|f_4\| \leq \gamma f_3 \right\}$$

So we compute the force closure by the following steps.
Firstly, we filter those contact points that

1. z coordinate is smaller than 2 cm, because gripper is not safe too be to close to the table.
2. the angle between the normals of two contact points smaller than 90 degrees.
3. the distance between two contact points are smaller than 2cm or bigger than 6cm, because gripper has a maximum and minimum length.

Secondly, we add those contact point pairs which satisfies the friction cone and force closure equation above.

From the paper Planning Optimal Grasps by Carlo Ferrari and John Canny, we get

$$Q = \min_{\hat{\omega}} LQ_{\hat{\omega}} = \min_{\hat{\omega}} \{\|\omega\| \ \ |\omega \in Bd(BG)\}$$

From the above equation, we compute the best grasp by the following steps Firstly, we set two impact parameters to define the quality of a grasp

1. grasp will be more stable the angle between surface normal and the line of two contact points are smaller. So the factor 1 is:
$$F_1 = 1 - \frac{\theta_1^2 + \theta_2^2}{2 \times arctan(\mu)}$$
2. vertical grasp is better because it will not need two think about the gravity, mass and friction factor of the object, at the same time, it will more stable than parallel grasp.
So the factor 2 is :
$$F_2 = 1 - \frac{\theta}{\frac{\pi}{2}}$$

From the transformations between different frame

$$g_{AC} = g_{AB} \times g_{BC}$$

and the Rigid Body Motion

$$g_{AB}(\theta_1) = g_{AB} \times g_{AB}(0)$$

1. We measure the distance between artag to the center of object by hand
2. We get the transformation between armarker frame to base frame by lookuptag function
3. We calculate the contacts positions in the object frame and returns the hand pose Transformation from object to gripper.

At last, we use open_gripper, close_gripper, go_to_pose function to execute it.

Section 3

Difficulties

1. We can not extract the contact points and normal vectors at first for gearbox and nozzle. Thanks to the help from Chris and Valmik, we use the new object file and output the accurate contacts and normals.

2. The numbers of contacts and normals are not the name for gearbox.obj and nozzle.obj. We have no idea at first, but we try to use the former pairs and delete the extra normals. Surprisingly, we can calculate out the best contact points.

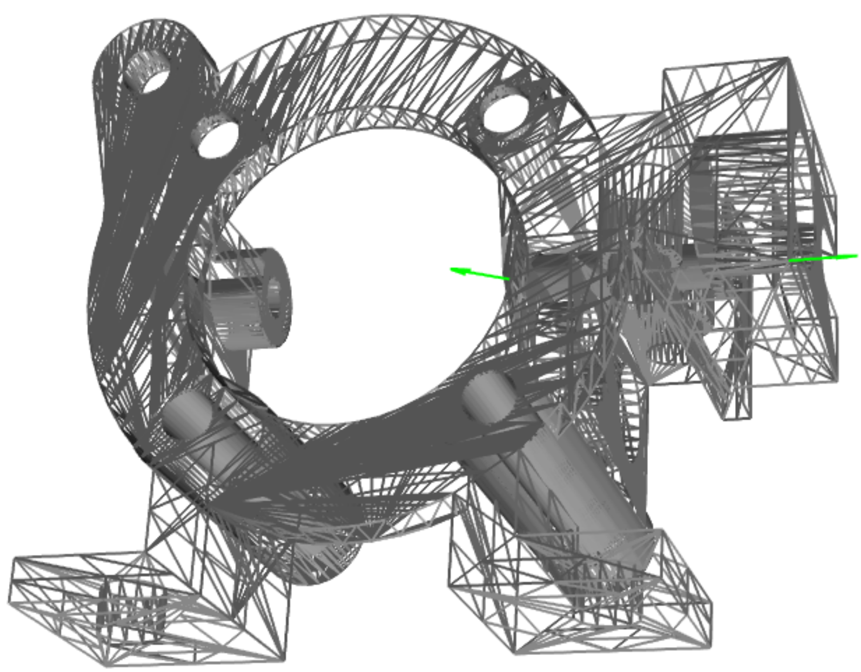
3. The distance between the object center to armarker is hard to measure exactly. We do some fine tuning when trying to grasp the object accurately.

Section 4

Grasp Result

The best contact points results for three objects are as follows:

Figure 1: gearbox



Section 4 continued on next page...

Figure 2: nozzle

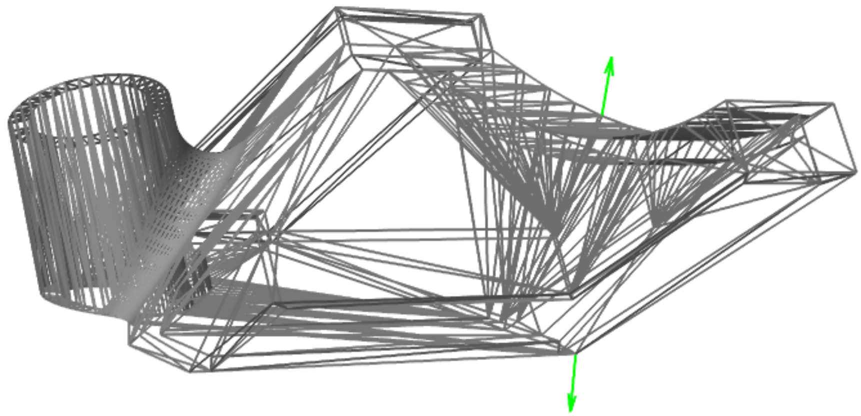
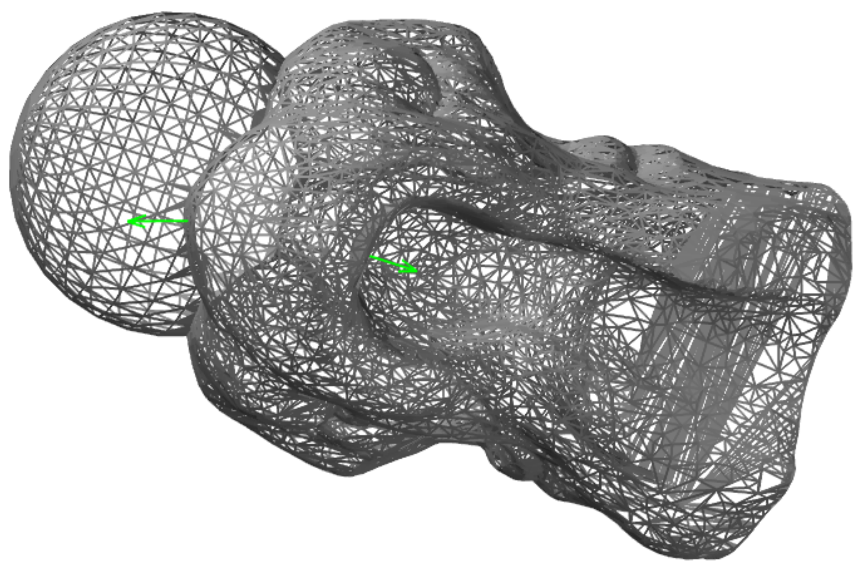


Figure 3: pawn



Gripping Result

We use the baxter and sawyer to grasp and for each object, we attempted 5 grasps. In the video above we chose a success implementation of each object.

Table 1: Robot: Baxter

Object	Gearbox	Nozzle	Pawn
Attempt	5	5	5
Success	4	2	1
Success rate	80%	40%	20%

Table 2: Robot: Sawyer

Object	Gearbox	Nozzle	Pawn
Attempt	5	5	5
Success	2	1	1
Success Rate	40%	20%	20%

Analysis

In general, Baxter performs better than Sawyer.

For each object, success rate of Gearbox is the highest and Pawn is lowest. We think it is related to the orientation of gripper and the shape of object.

For gearbox, as shown in the picture below, the two grasp points are parallel to the ground. Therefore, it is easier and more accurate for the robot to find and execute path.

However, for nozzle and pawn, the two points chosen are perpendicular to the ground, making it more difficult for moveit package to get to the desired configuration. Also, the gripper often hits the object when approaching, especially in grasping pawn, the two points of pawn have only small space for gripper to come in.

In our method, we make sure that the grasp forces are in force closure and resist gravity. After that, we calculate the quality of each pair of grasp points. Then we chose the best points and execute grasp. Theoretically, our method is better than the other two. This is the same in practice. For example, when we only take gravity resistance into consideration, the points chosen are even not able to grasp (e.g. points at bottom). Therefore, we only rely on one metric.

Section 5

Improvement

First, using moveit to move the robot arm is inaccurate and sometimes it may crash. Therefore, we think we can use motion planning in Lab1 to let the arm move to desired configuration. It will be a better way to control the robot arm.
Second, we can use some visualization method except for AR tags. For example, OpenCV can be used to detect the object and analysis its shape, then get the best points of grasping. Moreover, this method will be good in trying regrasp.

Section 6

Application

First, using moveit to move the robot arm is inaccurate and sometimes it may crash. Therefore, we think we can use motion planning in Lab1 to let the arm move to desired configuration. It will be a better way to control the robot arm.
Second, we can use some visualization method except for AR tags. For example, OpenCV can be used to detect the object and analysis its shape, then get the best points of grasping. Moreover, this method will be good in trying regrasp.

Section 7

Bonus

The lab document is great! It is clear and easy to understand. Thanks for all your efforts in it. Here is just some small advice.

1. The document is a little out of touch with the code.
   (a) The two object files gearbox and nozzle have different numbers of normals and contact points respectively, but we do not know that from the lab document, which might be a little confused.
   (b) The visualization package was changed slightly since last year but we do not know it in the lab document, which we should checkout an older version of visualization. We may not need package cvxpy.
   (c) It would be better if you would put some explanation or comment the use of some main function in the code package and point out some of the necessary functions in the code that we must use.
2. We do not know if Baxter can reach those point because we only have the information of the object points, so it might be something that need to improve in this part.

Section 8

Main function file:

```python
#!/usr/bin/env python -W ignore::DeprecationWarning
"""
Starter script for EE106B grasp planning lab
Author: Chris Correa
"""
import numpy as np
import math
import sys

import rospy
import tf
import time
from geometry_msgs.msg import Pose, PoseStamped
import tf.transformations as tfs
import moveit_commander
from moveit_msgs.msg import OrientationConstraint, Constraints
from autolab_core import RigidTransform, Point, NormalCloud, PointCloud
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
from meshpy import ObjFile
warnings.filterwarnings("ignore", category=DeprecationWarning)
from visualization import Visualizer3D as vis
warnings.filterwarnings("ignore", category=DeprecationWarning)
from baxter_interface import gripper as baxter_gripper
from utils import vec, adj
import scipy
import copy
import sys
import trimesh
# import cvxpy as cvx
import Queue
from grasp_metrics import compute_force_closure, compute_gravity_resistance,
    compute_custom_metric

# probably don't need to change these (but confirm that they're correct)
MAX_HAND_DISTANCE = .04
MIN_HAND_DISTANCE = .01
CONTACT_MU = 0.5
CONTACT_GAMMA = 0.1

# will need to change these
OBJECT_MASS = 0.25 # kg
# approximate the friction cone as the linear combination of `NUM_FACETS` vectors
NUM_FACETS = 32
# set this to false while debugging your grasp analysis
BAXTER_CONNECTED = True
# how many to execute
NUM_GRASPS = 6
OBJECT = "nozzle"

# objects are different this year so you'll have to change this
# also you can use nodes/object_pose_publisher.py instead of finding the ar tag and then
    computing T_ar_object in this script.
if OBJECT == "gearbox":
    MESH_FILENAME = '../objects/gearbox.obj'
    # ar tag on the paper
    TAG = 3
    # transform between the object and the AR tag on the paper
```

```
    T_ar_object = tfs.translation_matrix([-.07, -.11, 0.056])
    # how many times to subdivide the mesh
    SUBDIVIDE_STEPS = 0
elif OBJECT == 'nozzle':
    MESH_FILENAME = '../objects/nozzle.obj'
    TAG = 8
    T_ar_object = tfs.translation_matrix([-.065, .09, 0.032])
    SUBDIVIDE_STEPS = 0
elif OBJECT == "pawn":
    MESH_FILENAME = '../objects/pawn.obj'
    TAG = 14
    T_ar_object = tfs.translation_matrix([-.06, -.08, 0.091])#original [-.06, .11, 0.091]
    SUBDIVIDE_STEPS = 0

if BAXTER_CONNECTED:
    rospy.init_node('moveit_node')
    right_gripper = baxter_gripper.Gripper('right')

listener = tf.TransformListener()
from_frame = 'base'
time.sleep(1)

def rigid_transform(tag_pos, tag_rot):
    return tfs.translation_matrix(tag_pos).dot(tfs.quaternion_matrix(tag_rot))

def g_base_tag(tag_number):
    """ Returns the AR tag position in world coordinates

    Parameters
    ----------
    tag_number : int
        AR tag number

    Returns
    -------
    :obj:`autolab_core.RigidTransform` AR tag position in world coordinates
    """
    to_frame = 'ar_marker_{}'.format(tag_number)
    if not listener.frameExists(from_frame) or not listener.frameExists(to_frame):
        print 'Frames not found'
        print 'Did you place AR marker {} within view of the baxter left hand camera?'.
    format(tag_number)
        exit(0)
    t = listener.getLatestCommonTime(from_frame, to_frame)
    tag_pos, tag_rot = listener.lookupTransform(from_frame, to_frame, t)
    return rigid_transform(tag_pos, tag_rot)

# def lookup_tag(tag_number):

#     listener = tf.TransformListener()
#     from_frame = 'base'
#     to_frame = 'ar_marker_{}'.format(tag_number)
#     # if not listener.frameExists(from_frame) or not listener.frameExists(to_frame):
#     #     print 'Frames not found'
#     #     print 'Did you place AR marker {} within view of the baxter left hand camera?'.
    format(tag_number)
#     #     exit(0)
#     # t = rospy.Time(0)*
#     # if listener.canTransform(from_frame, to_frame, t):
#     listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(4.0))
#     t = listener.getLatestCommonTime(from_frame, to_frame)
#     tag_pos, tag_rot = listener.lookupTransform(from_frame, to_frame, t)
```

```python
#     return (tag_pos + tag_rot)    # Return value is a 'list'

def close_gripper():
    """closes the gripper"""
    right_gripper.close(block=True)
    rospy.sleep(1.0)

def open_gripper():
    """opens the gripper"""
    right_gripper.open(block=True)
    rospy.sleep(1.0)

def go_to_pose(pose):
    """Uses Moveit to go to the pose specified
    Parameters
    ----------
    pose : :obj:`geometry_msgs.msg.Pose`
        The pose to move to
    """

    right_arm.set_start_state_to_current_state()
    right_arm.set_pose_target(pose)
    right_arm.plan()
    right_arm.go()

def execute_grasp(T_object_gripper):
    """takes in the desired hand position relative to the object, finds the desired hand position in world coordinates.
       Then moves the gripper from its starting orientation to some distance behind the object, then move to the
       hand pose in world coordinates, closes the gripper, then moves up.

    Parameters
    ----------
    T_object_gripper : :obj:`autolab_core.RigidTransform`
        desired position of gripper relative to the objects coordinate frame
    """
    inp = raw_input('Press <Enter> to move, or \'exit\' to exit')
    if inp == "exit":
        return
    # YOUR CODE HERE

def contacts_to_baxter_hand_pose(contacts, normals, approach_direction=None):
    """ takes the contacts positions in the object frame and returns the hand pose T_obj_gripper

    Parameters
    ----------
    contact1 : :obj:`numpy.ndarray`
        position of finger1 in object frame
    contact2 : :obj:`numpy.ndarray`
        position of finger2 in object frame
    approach_direction : :obj:`numpy.ndarray`
        there are multiple grasps that go through contact1 and contact2.  This describes which
        orientation the hand should be in

    Returns
    -------
    :obj:`autolab_core:RigidTransform` Hand pose in the object frame
    """
    # YOUR CODE HERE
```

```
    # T_obj_gripper = ????
    target_pos = (contacts[:3] + contacts[3:]) / 2
    target_normal = - np.cross(normals[:3], normals[3:])
    target_parallel = contacts[:3] - contacts[3:]
    # target_rot = np.array( [-0.603, 0.402, -0.456, 0.516]) #gearbox
    target_rot = np.array( [-0.559, 0.829, -0.019, -0.024]) #nozzle
    # target_rot = np.array( [0.398, 0.519, 0.609, 0.449]) #pawn

    return np.append(target_pos , target_rot)

def sorted_contacts(vertices, normals, T_ar_object):
    """ takes mesh and returns pairs of contacts and the quality of grasp between the
    contacts, sorted by quality

    Parameters
    ----------
    vertices : :obj:`numpy.ndarray`
        nx3 mesh vertices
    normals : :obj:`numpy.ndarray`
        nx3 mesh normals
    T_ar_object : :obj:`autolab_core.RigidTransform`
        transform from the AR tag on the paper to the object

    Returns
    -------
    :obj:`list` of :obj:`numpy.ndarray`
        grasp_indices[i][0] and grasp_indices[i][1] are the indices of a pair of vertices.
    These are randomly
        sampled and their quality is saved in best_metric_indices
    :obj:`list` of int
        best_metric_indices is the indices of grasp_indices in order of grasp quality
    """

    # prune vertices that are too close to the table so you dont smack into the table
    # you may want to change this line, to be how you see fit
    possible_indices = np.r_[:len(vertices)][vertices[:,2] + T_ar_object[2,3] >= 0.03]

    # Finding grasp via vertex sampling.  make sure to not consider grasps where the
    # vertices are too big for the gripper
    all_metrics = list()
    metric = compute_custom_metric
    grasp_indices = list()
    # for i in range(?????):
    #     candidate_indices = np.random.choice(possible_indices, 2, replace=False)
    #     grasp_indices.append(candidate_indices)
    #     contacts = vertices[candidate_indices]
    #     contact_normals = normals[candidate_indices]

    #     # YOUR CODE HERE
    #     all_metrics.append(????)

    # YOUR CODE HERE.  sort metrics and return the sorted order

    return grasp_indices, best_metric_indices


if __name__ == '__main__':
    if BAXTER_CONNECTED:
        moveit_commander.roscpp_initialize(sys.argv)
        # rospy.init_node('moveit_node')
        # print('init\n\n\n\n\n')
        robot = moveit_commander.RobotCommander()
```

```
        scene = moveit_commander.PlanningSceneInterface()
        right_arm = moveit_commander.MoveGroupCommander('right_arm')
        right_arm.set_planner_id('RRTConnectkConfigDefault')
        right_arm.set_planning_time(5)

        # rospy.Subscriber("tf",tfMessage, callback)

    # Main Code
    br = tf.TransformBroadcaster()

    # SETUP
    # of = ObjFile(MESH_FILENAME)
    # mesh = of.read()
    mesh = trimesh.load(MESH_FILENAME)

    g_base_ar = g_base_tag(TAG)
    g_base_obj = g_base_ar.dot(T_ar_object)
    print('g base->ar',g_base_ar)
    print('g ar->obj',T_ar_object)
    print('g base->obj' ,g_base_obj)


    # We found this helped.  You may not.  I believe there was a problem with setting the
    surface normals.
    # I remember fixing that....but I didn't save that code, so you may have to redo it.
    # You may need to fix that if you call this function.
    for i in range(SUBDIVIDE_STEPS):
        mesh = mesh.subdivide(min_tri_length=.02)

    vertices = mesh.vertices
    triangles = mesh.triangles
    # normals = mesh.normals
    normals = mesh.vertex_normals
    force_closure = compute_force_closure(vertices, normals, CONTACT_MU)
    best_contacts_objframe, best_normals_objframe = compute_custom_metric(force_closure[0],
    force_closure[1], CONTACT_MU)
    c1, c2 = np.append(best_contacts_objframe[:3], 1),np.append(best_contacts_objframe[3:],
    1)
    best_contacts_baseframe = np.append(g_base_obj.dot(c1.reshape((4, 1)))[0:3] , g_base_obj
    .dot(c2.reshape((4, 1)))[0:3])
    print('best_contacts_baseframe',best_contacts_baseframe)
    best_normals_baseframe = best_normals_objframe
    hand_pos = contacts_to_baxter_hand_pose(best_contacts_baseframe, best_normals_baseframe)
    print(hand_pos)

    # open_gripper()
    # hand_pos1 = hand_pos.copy()
    # hand_pos1[0] -= .10
    # hand_pos2 = hand_pos.copy()
    # hand_pos2[2] += .1
    # go_to_pose(list(hand_pos1))
    # rospy.sleep(1)
    # hand_pos[0] += 0.015
    # go_to_pose(list(hand_pos))
    # close_gripper()
    # go_to_pose(list(hand_pos2))
    # rospy.sleep(0.5)
    # go_to_pose(list(hand_pos))
    # open_gripper()

    open_gripper()
    hand_pos1 = hand_pos.copy()
```

```
    hand_pos1[2] += .10
    hand_pos2 = hand_pos.copy()
    hand_pos2[2] += .1
    go_to_pose(list(hand_pos1))
    rospy.sleep(1)
    # hand_pos[2] -= 0.022
    hand_pos[2] += 0.01
    go_to_pose(list(hand_pos))
    close_gripper()
    go_to_pose(list(hand_pos2))
    rospy.sleep(0.5)
    go_to_pose(list(hand_pos))
    open_gripper()



    # ??? = sorted_contacts(???)

    # YOUR CODE HERE
    # for current_metric in ?????:
    #     # YOUR CODE HERE

    #     # visualize the mesh and contacts
    #     vis.figure()
    #     vis.mesh(mesh)
    #     vis.normals(NormalCloud(np.hstack((normal1.reshape(-1, 1), normal2.reshape(-1, 1))
    ), frame='test'),
    #         PointCloud(np.hstack((contact1.reshape(-1, 1), contact2.reshape(-1, 1))),
    frame='test'))
    #     # vis.pose(T_obj_gripper, alpha=0.05)
    #     vis.show()
    #     if BAXTER_CONNECTED:
    #         repeat = True
    #         while repeat:
    #             execute_grasp(T_obj_gripper)
    #             repeat = bool(raw_input("repeat?"))


    # # 500, 1200
    exit()
```

./main.py

Grasp function file:

```
# may need more imports
import numpy as np
from utils import vec, adj

def cal_distance(point1, point2):
    """point1, 2 is np.array 1x3 a
       return the distance scalar"""
    return np.linalg.norm(point1.reshape((3, 1)) - point2.reshape((3, 1)))

def cal_angle(point1, point2):
    """point1, 2 is np.array 1x3 a
       return the angle in rad"""
    lx, ly = np.linalg.norm(point1.reshape((3, 1))), np.linalg.norm(point2.reshape((3, 1)))
    cos_angle = point1.dot(point2) / (lx * ly)
    angle = np.arccos(cos_angle)
    if angle > np.pi / 2:
        return np.pi - angle
    else:
```

```
        return angle

def compute_force_closure(contacts, normals, mu):

#def compute_force_closure(contacts, normals, num_facets, mu, gamma, object_mass):
    """ Compute the force closure of some object at contacts, with normal vectors stored in
    normals
        You can use the line method described in HW2.  if you do you will not need
    num_facets

    Parameters
    ----------
    contacts : :obj:`numpy.ndarray`
        obj mesh vertices on which the fingers will be placed
    normals : :obj:`numpy.ndarray`
        obj mesh normals at the contact points
    num_facets : int
        number of vectors to use to approximate the friction cone.  these vectors will be
    along the friction cone boundary
    mu : float
        coefficient of friction
    gamma : float
        torsional friction coefficient
    object_mass : float
        mass of the object

    Returns
    -------
    float : quality of the grasp
    """
    # YOUR CODE HERE
    grasp_candidates = []
    grasp_candidates_normals = []
    factor = 5
    length = contacts.shape[0]
    for i in range(0, length, factor):
        for j in range(0, length, factor):
            c1, c2 = contacts[i],  contacts[j]
            if c1[2] == c2[2] or normals[i].dot(normals[j]) > 0 or c1[2]<0.02 or c2[2]<0.02:
                continue
            distance = cal_distance(c1, c2)

            if distance < 0.02 or distance > 0.06:
                continue
            else:
                pointVector = (c1 - c2)
                theta1 = cal_angle(pointVector, normals[i])
                theta2 = cal_angle(pointVector, normals[j])
                coneangle = np.arctan(mu)
                if theta1 < coneangle and theta2 < coneangle:
                    grasp_candidates.append(list(c1) + list(c2))
                    grasp_candidates_normals.append(list(normals[i]) + list(normals[j]))

    return np.array(grasp_candidates), np.array(grasp_candidates_normals)



# defined in the book on page 219
def get_grasp_map(contacts, normals, num_facets, mu, gamma):
    """ Compute the grasp map given the contact points and their surface normals

    Parameters
```

```
    ----------
    contacts : :obj:`numpy.ndarray`
        obj mesh vertices on which the fingers will be placed
    normals : :obj:`numpy.ndarray`
        obj mesh normals at the contact points
    num_facets : int
        number of vectors to use to approximate the friction cone.  these vectors will be
    along the friction cone boundary
    mu : float
        coefficient of friction
    gamma : float
        torsional friction coefficient

    Returns
    -------
    :obj:`numpy.ndarray` grasp map
    """
    # YOUR CODE HERE
    pass

def contact_forces_exist(contacts, normals, num_facets, mu, gamma, desired_wrench):
    """ Compute whether the given grasp (at contacts with surface normals) can produce the
    desired_wrench.
        will be used for gravity resistance.

    Parameters
    ----------
    contacts : :obj:`numpy.ndarray`
        obj mesh vertices on which the fingers will be placed
    normals : :obj:`numpy.ndarray`
        obj mesh normals at the contact points
    num_facets : int
        number of vectors to use to approximate the friction cone.  these vectors will be
    along the friction cone boundary
    mu : float
        coefficient of friction
    gamma : float
        torsional friction coefficient
    desired_wrench : :obj:`numpy.ndarray`
        potential wrench to be produced

    Returns
    -------
    bool : whether contact forces can produce the desired_wrench on the object
    """
    # YOUR CODE HERE
    pass

def compute_gravity_resistance(contacts, normals, num_facets, mu, gamma, object_mass):
    """ Gravity produces some wrench on your object.  Computes whether the grasp can produce
     and equal and opposite wrench

    Parameters
    ----------
    contacts : :obj:`numpy.ndarray`
        obj mesh vertices on which the fingers will be placed
    normals : :obj:`numpy.ndarray`
        obj mesh normals at the contact points
    num_facets : int
        number of vectors to use to approximate the friction cone.  these vectors will be
    along the friction cone boundary
    mu : float
```

```
        coefficient of friction
    gamma : float
        torsional friction coefficient
    object_mass : float
        mass of the object

    Returns
    -------
    float : quality of the grasp
    """
    # YOUR CODE HERE (contact forces exist may be useful here)
    pass

# def compute_custom_metric(contacts, normals, num_facets, mu, gamma, object_mass):
def compute_custom_metric(contacts, normals, mu):
    """ I suggest Ferrari Canny, but feel free to do anything other metric you find.

    Parameters
    ----------
    contacts : :obj:`numpy.ndarray`
        obj mesh vertices on which the fingers will be placed
    normals : :obj:`numpy.ndarray`
        obj mesh normals at the contact points
    num_facets : int
        number of vectors to use to approximate the friction cone.  these vectors will be
    along the friction cone boundary
    mu : float
        coefficient of friction
    gamma : float
        torsional friction coefficient
    object_mass : float
        mass of the object

    Returns
    -------
    float : quality of the grasp
    """
    # YOUR CODE HERE :)
    scores = np.zeros((contacts.shape[0], 1))
    for i in range(contacts.shape[0]):
        c1, c2 = contacts[i][0:3], contacts[i][3:]
        n1, n2 = normals[i][0:3], normals[i][3:]
        theta1, theta2 = cal_angle(c1 - c2, n1), cal_angle(c1 - c2, n2)
        theta = cal_angle(c1 - c2, np.array([0, 0, 1]))
        score1 = (1- ( np.power(theta1,2)+np.power(theta2,2) )/2/np.power(np.arctan(mu),2) )
    *100
        score2 = (1- np.power( theta/np.pi*2 ,2) )*100
        score = score1*0.1 + score2 * 0.9
        scores[i] = score
    max_index = np.where(scores==np.max(scores))[0][0]
    # print('max_index',max_index.shape)
    return contacts[max_index], normals[max_index]
```

./grasp_metrics.py

Test function file (Output the information of the object file:

```
# #!/usr/bin/env python -W ignore::DeprecationWarning
# """
# Starter script for EE106B grasp planning lab
# Author: Chris Correa
# """
import numpy as np
```

```python
from grasp_metrics import *
# import math
# import sys

# #import rospy
# import tf
# import time
# from geometry_msgs.msg import Pose, PoseStamped
# import tf.transformations as tfs
# import moveit_commander
# from moveit_msgs.msg import OrientationConstraint, Constraints
from autolab_core import RigidTransform, Point, NormalCloud, PointCloud
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
from meshpy import ObjFile
import trimesh
warnings.filterwarnings("ignore", category=DeprecationWarning)
from visualization import Visualizer3D as vis
warnings.filterwarnings("ignore", category=DeprecationWarning)
# from baxter_interface import gripper as baxter_gripper
# from utils import vec, adj
# import scipy
# import copy
# import sys
# import cvxpy as cvx
# import Queue
# from grasp_metrics import compute_force_closure, compute_gravity_resistance,
    compute_custom_metric

# probably don't need to change these (but confirm that they're correct)
MAX_HAND_DISTANCE = .04
MIN_HAND_DISTANCE = .01
CONTACT_MU = 0.5
CONTACT_GAMMA = 0.1

# will need to change these
OBJECT_MASS = 0.25 # kg
# approximate the friction cone as the linear combination of `NUM_FACETS` vectors
NUM_FACETS = 32
# set this to false while debugging your grasp analysis
BAXTER_CONNECTED = False
# how many to execute
NUM_GRASPS = 6
OBJECT = "pawn"
MESH_FILENAME = '../objects/pawn.obj'

mesh = trimesh.load(MESH_FILENAME)
# vertices = mesh.vertices
# triangles = mesh.triangles
# normals = mesh.normals
vertices = mesh.vertices
triangles = mesh.triangles
normals = -mesh.vertex_normals

of = ObjFile(MESH_FILENAME)
mesh = of.read()

# number = min(vertices.shape[0],normals.shape[0])
# vertices = vertices[:number]
# normals = normals[:number]

# print('vertices:',vertices)
```

```
# print('triangles:',triangles)
# print('normals:',normals)
print(vertices.shape)
print(normals.shape)
fc = compute_force_closure(vertices, normals, CONTACT_MU)
# print('fc:',fc)
best = compute_custom_metric(fc[0], fc[1], CONTACT_MU)
# print(best)
# print(fc[0].shape)
#print('fc', fc)

# contact1 = fc[0][100][0:3]
# contact2 = fc[0][100][3:]
# normal1 = fc[1][100][0:3]
# normal2 = fc[1][100][3:]

contact1 = best[0][0:3]
contact2 = best[0][3:]
normal1 = best[1][0:3]
normal2 =  best[1][3:]

# from autolab_core import BagOfPoints
# point = BagOfPoints(fc[0][0:3, :])

vis.figure()
vis.mesh(mesh)
vis.normals(NormalCloud(np.hstack((normal1.reshape(-1, 1), normal2.reshape(-1, 1))), frame='
    test'),
    PointCloud(np.hstack((contact1.reshape(-1, 1), contact2.reshape(-1, 1))), frame='test'))
# vis.pose(T_obj_gripper, alpha=0.05)
vis.show()
```

./test.py