# graspnetAPI

***Release 1.2.10***

**graspnet**

**Mar 26, 2022**

# CONTENTS:

CHAPTER

ONE

# ABOUT GRASPNETAPI


GraspNet is an open project for general object grasping that is continuously enriched. Currently we release GraspNet-1Billion, a large-scale benchmark for general object grasping, as well as other related areas (e.g. 6D pose estimation, unseen object segmentation, etc.). graspnetAPI is a Python API that assists in loading, parsing and visualizing the annotations in GraspNet. Please visit graspnet website for more information on GraspNet, including for the data, paper, and tutorials. The exact format of the annotations is also described on the GraspNet website. In addition to this API, please download both the GraspNet images and annotations in order to run the demo.

## 1.1 Resources

- Documentations
- PDF_Documentations
- Website
- Code

## 1.2 License

graspnetAPI is licensed under the none commercial CC4.0 license [see https://graspnet.net/about]

CHAPTER

TWO

# INSTALLATION

**Note:** Only Python 3 on Linux is supported.

## 2.1 Prerequisites

Python version under 3.6 is not tested.

## 2.2 Dataset

### 2.2.1 Download

Download the dataset at https://graspnet.net/datasets.html

### 2.2.2 Unzip

Unzip the files as shown in https://graspnet.net/datasets.html.

### 2.2.3 Rectangle Grasp Labels

Rectangle grasp labels are optional if you need labels in this format. You can both generate the labels or download the file.

If you want to generate the labels by yourself, you may refer to *Generating Rectangle Grasp Labels*.

**Note:** Generating rectangle grasp labels may take a long time.

After generating the labels or unzipping the labels, you need to run copy_rect_labels.py to copy rectangle grasp labels to corresponding folders.

### 2.2.4 Dexnet Model Cache

Dexnet model cache is optional without which the evaluation will be much slower(about 10x time slower). You can both download the file or generate it by yourself by running gen_pickle_dexmodel.py_(recommended).

## 2.3 Install API

You may install using pip:

```
pip install graspnetAPI
```

You can also install from source:

```
git clone https://github.com/graspnet/graspnetAPI.git
cd graspnetAPI/
pip install .
```

CHAPTER

THREE

# GRASP LABEL FORMAT

## 3.1 Raw Label Format

The raw label is composed of two parts, i.e. labels for all grasp candidates on each object and collision masks for each scene.

### 3.1.1 Labels on Objects

The raw label on each object is a list of numpy arrays.

```
>>> import numpy as np
>>> l = np.load('000_labels.npz') # GRASPNET_ROOT/grasp_label/000_labels.npz
>>> l.files
['points', 'offsets', 'collision', 'scores']
>>> l['points'].shape
(3459, 3)
>>> l['offsets'].shape
(3459, 300, 12, 4, 3)
>>> l['collision'].shape
(3459, 300, 12, 4)
>>> l['collision'].dtype
dtype('bool')
>>> l['scores'].shape
(3459, 300, 12, 4)
>>> l['scores'][0][0][0][0]
-1.0
```

- 'points' records the grasp center point coordinates in model frame.
- 'offsets' records the in-plane rotation, depth and width of the gripper respectively in the last dimension.
- 'collision' records the bool mask for if the grasp pose collides with the model.
- 'scores' records the minimum coefficient of friction between the gripper and object to achieve a stable grasp.

**Note:** In the raw label, the **lower** score the grasp has, the **better** it is. However, -1.0 score means the grasp pose is totally not acceptable.

300, 12, 4 denote view id, in-plane rotation id and depth id respectively. The views are defined here in graspnetAPI/utils/utils.py.

```
        cloud = cloud.reshape([-1, 3])
    return cloud

def generate_views(N, phi=(np.sqrt(5)-1)/2, center=np.zeros(3, dtype=np.float32),␣
↪R=1):
    ''' Author: chenxi-wang
    View sampling on a sphere using Febonacci lattices.

    **Input:**
```

### 3.1.2 Collision Masks on Each Scene

Collision mask on each scene is a list of numpy arrays.

```
>>> import numpy as np
>>> c = np.load('collision_labels.npz') # GRASPNET_ROOT/collision_label/scene_0000/
↪collision_labels.npz
>>> c.files
['arr_0', 'arr_4', 'arr_5', 'arr_2', 'arr_3', 'arr_7', 'arr_1', 'arr_8', 'arr_6']
>>> c['arr_0'].shape
(487, 300, 12, 4)
>>> c['arr_0'].dtype
dtype('bool')
>>> c['arr_0'][10][20][3]
array([ True,  True,  True,  True])
```

'arr_i' is the collision mask for the *i* th object in the *object_id_list.txt* for each scene whose shape is (num_points, 300, 12, 4). num_points, 300, 12, 4 denote the number of points in the object, view id, in-plane rotation id and depth id respectively.

Users can refer to `graspnetAPI.GraspNet.loadGrasp()` for more details of how to use the labels.

## 3.2 API Loaded Labels

Dealing with the raw labels are time-consuming and need high familiarity with graspnet. So the API also provides an easy access to the labels.

By calling `graspnetAPI.GraspNet.loadGrasp()`, users can get all the positive grasp labels in a scene with their parameters and scores.

There are totally four kinds of data structures for loaded grasp labels: **Grasp**, **GraspGroup**, **RectGrasp** and **RectGraspGroup**. The internal data format of each class is a numpy array which is more efficient than the Python list. Their definitions are given in grasp.py

### 3.2.1 Example Labels

Before looking into the details, an example is given below.

Loading a GraspGroup instance.

```
__author__ = 'mhgou'
__version__ = '1.0'

from graspnetAPI import GraspNet, Grasp, GraspGroup
import open3d as o3d
import cv2
import numpy as np

# GraspNetAPI example for loading grasp for a scene.
# change the graspnet_root path

####################################################################
graspnet_root = '/disk1/graspnet' # ROOT PATH FOR GRASPNET
####################################################################

sceneId = 1
annId = 3

# initialize a GraspNet instance
g = GraspNet(graspnet_root, camera='kinect', split='train')

# load grasps of scene 1 with annotation id = 3, camera = kinect and fric_coef_thresh␣
↪= 0.2
_6d_grasp = g.loadGrasp(sceneId = sceneId, annId = annId, format = '6d', camera =
↪'kinect', fric_coef_thresh = 0.2)
print('6d grasp:\n{}'.format(_6d_grasp))

# _6d_grasp is an GraspGroup instance defined in grasp.py
print('_6d_grasp:\n{}'.format(_6d_grasp))
```

Users can access elements by index or slice.

```
# index
grasp = _6d_grasp[0]
print('_6d_grasp[0](grasp):\n{}'.format(grasp))

# slice
print('_6d_grasp[0:2]:\n{}'.format(_6d_grasp[0:2]))
print('_6d_grasp[[0,1]]:\n{}'.format(_6d_grasp[[0,1]]))
```

Each element of GraspGroup is a Grasp instance. The properties of Grasp can be accessed via provided methods.

```
# grasp is a Grasp instance defined in grasp.py
# access and set properties
print('grasp.translation={}'.format(grasp.translation))
grasp.translation = np.array([1.0, 2.0, 3.0])
print('After modification, grasp.translation={}'.format(grasp.translation))
print('grasp.rotation_matrix={}'.format(grasp.rotation_matrix))
grasp.rotation_matrix = np.eye(3).reshape((9))
print('After modification, grasp.rotation_matrix={}'.format(grasp.rotation_matrix))
print('grasp.width={}, height:{}, depth:{}, score:{}'.format(grasp.width, grasp.
↪height, grasp.depth, grasp.score))
```

(continues on next page)

(continued from previous page)

```
print('More operation on Grasp and GraspGroup can be seen in the API document')
```

RectGrasp is the class for rectangle grasps. The format is different from Grasp. But the provided APIs are similar.

```
# load rectangle grasps of scene 1 with annotation id = 3, camera = realsense and␣
↪fric_coef_thresh = 0.2
rect_grasp_group = g.loadGrasp(sceneId = sceneId, annId = annId, format = 'rect',␣
↪camera = 'realsense', fric_coef_thresh = 0.2)
print('rectangle grasp group:\n{}'.format(rect_grasp_group))

# rect_grasp is an RectGraspGroup instance defined in grasp.py
print('rect_grasp_group:\n{}'.format(rect_grasp_group))

# index
rect_grasp = rect_grasp_group[0]
print('rect_grasp_group[0](rect_grasp):\n{}'.format(rect_grasp))

# slice
print('rect_grasp_group[0:2]:\n{}'.format(rect_grasp_group[0:2]))
print('rect_grasp_group[[0,1]]:\n{}'.format(rect_grasp_group[[0,1]]))

# properties of rect_grasp
print('rect_grasp.center_point:{}, open_point:{}, height:{}, score:{}'.format(rect_
↪grasp.center_point, rect_grasp.open_point, rect_grasp.height, rect_grasp.score))
```

### 3.2.2 6D Grasp

Actually, 17 float numbers are used to define a general 6d grasp. The width, height, depth, score and attached object id are also part of the definition.

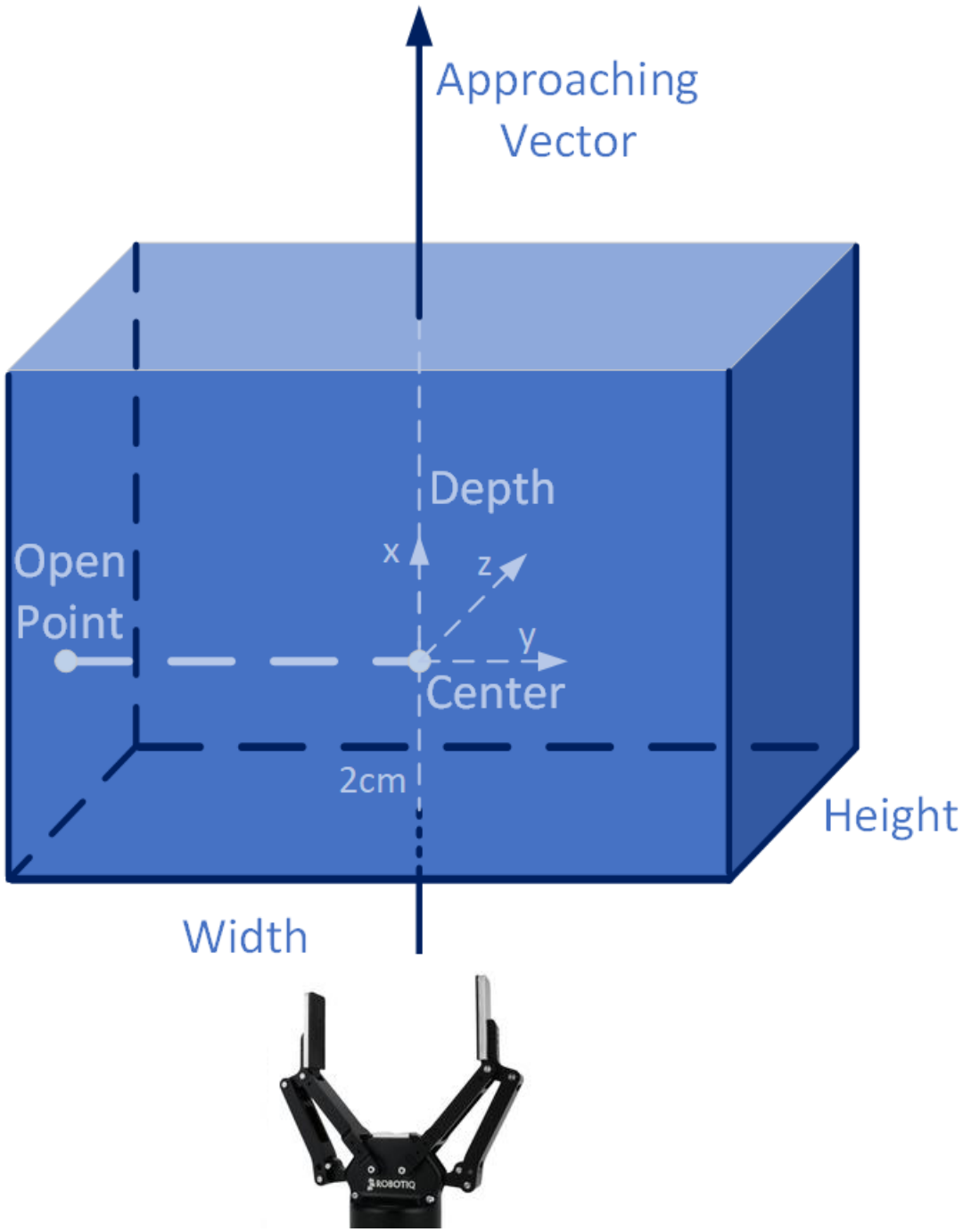
---

**Note:** In the loaded label, the **higher** score the grasp has, the **better** it is which is different from raw labels. Actually, score = 1.1 - raw_score (which is the coefficient of friction)

---

```
                target_points = target_points[mask1]
                target_points = transform_points(target_points, trans)
                target_points = transform_points(target_points, np.linalg.inv(camera_
↪pose))
```

The detailed defition of each parameter is shown in the figure.




```
class Grasp():
    def __init__(self, *args):
        '''
        **Input:**

        - args can be a numpy array or tuple of the score, width, height, depth,␣
↪rotation_matrix, translation, object_id

        - the format of numpy array is [score, width, height, depth, rotation_
↪matrix(9), translation(3), object_id]
```

(continues on next page)

(continued from previous page)

```
        - the length of the numpy array is 17.
        '''
        if len(args) == 0:
            self.grasp_array = np.array([0, 0.02, 0.02, 0.02, 1, 0, 0, 0, 1 ,0 , 0, 0,
↪ 1, 0, 0, 0, -1], dtype = np.float64)
        elif len(args) == 1:
            if type(args[0]) == np.ndarray:
                self.grasp_array = copy.deepcopy(args[0])
            else:
                raise TypeError('if only one arg is given, it must be np.ndarray.')
        elif len(args) == 7:
            score, width, height, depth, rotation_matrix, translation, object_id =␣
↪args
            self.grasp_array = np.concatenate([np.array((score, width, height,␣
↪depth)),rotation_matrix.reshape(-1), translation, np.array((object_id)).reshape(-
↪1)]).astype(np.float64)
        else:
            raise ValueError('only 1 or 7 arguments are accepted')
```

### 3.2.3 6D Grasp Group

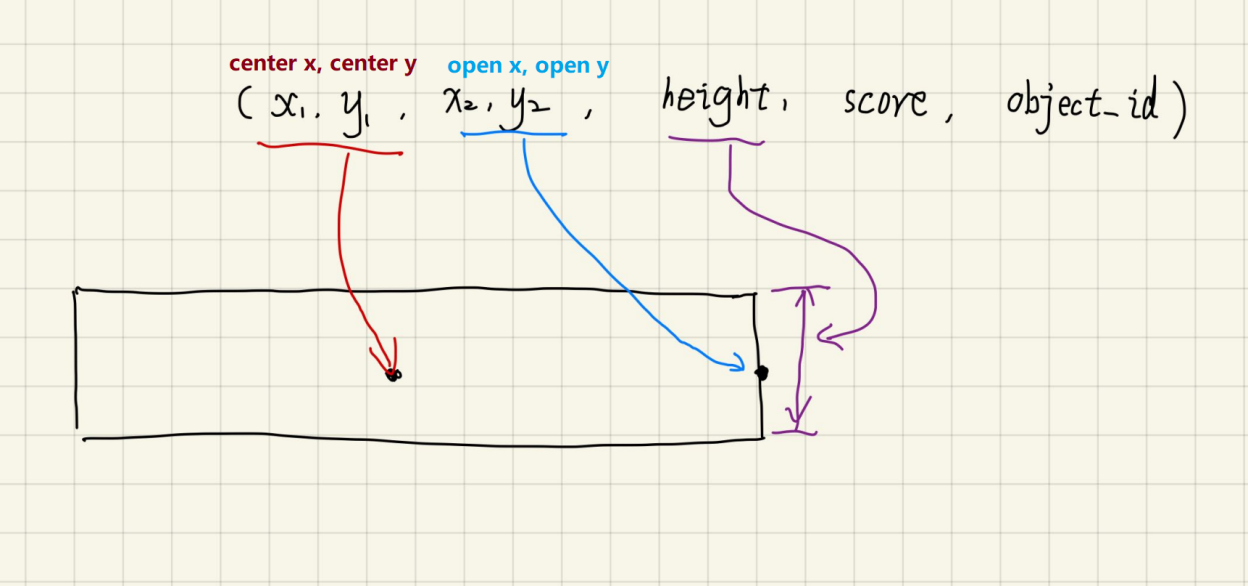
Usually, there are a lot of grasps in a scene, `GraspGroup` is a class for these grasps. Compared with `Grasp`, `GraspGroup` contains a 2D numpy array, the additional dimension is the index for each grasp.

```
        - list of open3d.geometry.Geometry of the gripper.
        '''
        return plot_gripper_pro_max(self.translation, self.rotation_matrix, self.
↪width, self.depth, score = self.score, color = color)

class GraspGroup():
    def __init__(self, *args):
        '''
        **Input:**

        - args can be (1) nothing (2) numpy array of grasp group array (3) str of the␣
↪npy file.
        '''
        if len(args) == 0:
            self.grasp_group_array = np.zeros((0, GRASP_ARRAY_LEN), dtype=np.float64)
        elif len(args) == 1:
            if isinstance(args[0], np.ndarray):
                self.grasp_group_array = args[0]
            elif isinstance(args[0], str):
                self.grasp_group_array = np.load(args[0])
```

Common operations on a list such as indexing, slicing and sorting are implemented. Besides, one important function is that users can **dump** a GraspGroup into a numpy file and **load** it in another program by calling `GraspGroup.save_npy()` and `GraspGroup.from_npy()`.

### 3.2.4 Rectangle Grasp

7 float numbers are used to define a general rectangle grasp, i.e. the center point, the open point, height, score and the attached object id. The detailed definition of each parameter is shown in the figure above and below and the coordinates for center point and open point are in the pixel frame.




```
        '''
        from grasp_nms import nms_grasp
        return GraspGroup(nms_grasp(self.grasp_group_array, translation_thresh,␣
↪rotation_thresh))

class RectGrasp():
    def __init__(self, *args):
        '''
        **Input:**

        - args can be a numpy array or tuple of the center_x, center_y, open_x, open_
↪y, height, score, object_id

        - the format of numpy array is [center_x, center_y, open_x, open_y, height,␣
↪score, object_id]

        - the length of the numpy array is 7.
        '''
        if len(args) == 1:
            if type(args[0]) == np.ndarray:
                self.rect_grasp_array = copy.deepcopy(args[0])
            else:
                raise TypeError('if only one arg is given, it must be np.ndarray.')
```

### 3.2.5 Rectangle Grasp Group

The format for `RectGraspGroup` is similar to that of `RectGrasp` and `GraspGroup`.

```
        if height < EPS:
            return None
        return Grasp(score, width, height, depth, rotation, translation, object_id)

class RectGraspGroup():
    def __init__(self, *args):
        '''
        **Input:**

        - args can be (1) nothing (2) numpy array of rect_grasp_group_array (3) str␣
↪of the numpy file.
        '''
        if len(args) == 0:
            self.rect_grasp_group_array = np.zeros((0, RECT_GRASP_ARRAY_LEN),␣
↪dtype=np.float64)
        elif len(args) == 1:
            if isinstance(args[0], np.ndarray):
                self.rect_grasp_group_array = args[0]
            elif isinstance(args[0], str):
                self.rect_grasp_group_array = np.load(args[0])
```

**Note:** We recommend users to access and modify the labels by provided functions although users can also manipulate the data directly but it is **Not Recommended**. Please refer to the Python API for more details.

## 3.3 Grasp and GraspGroup Transformation

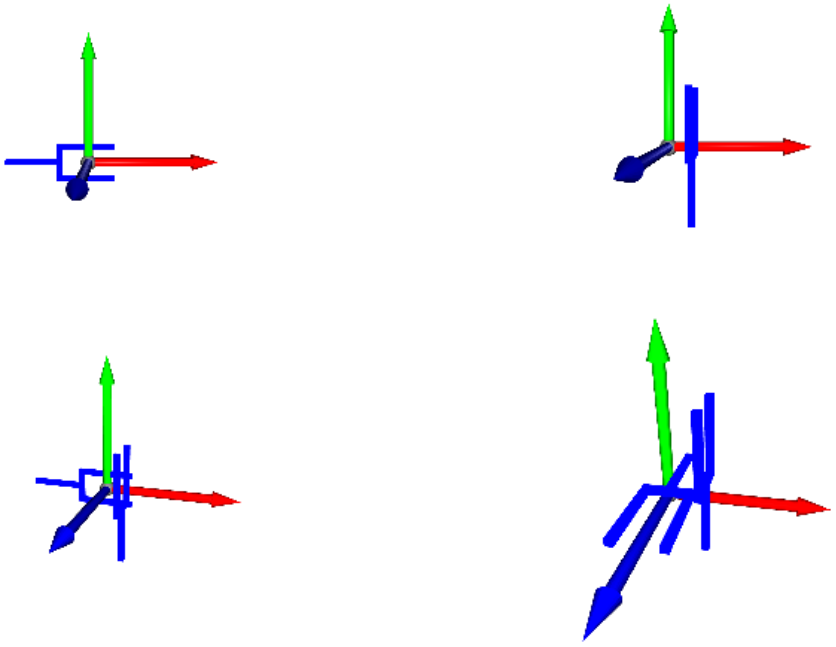
Users can transform a Grasp or GraspGroup giving a 4x4 matrix.

```
# transform grasp
g = Grasp() # simple Grasp
frame = o3d.geometry.TriangleMesh.create_coordinate_frame(0.1)

# Grasp before transformation
o3d.visualization.draw_geometries([g.to_open3d_geometry(), frame])
g.translation = np.array((0,0,0.01))

# setup a transformation matrix
T = np.eye(4)
T[:3,3] = np.array((0.01, 0.02, 0.03))
T[:3,:3] = np.array([[0,0,1.0],[1,0,0],[0,1,0]])
g.transform(T)

# Grasp after transformation
o3d.visualization.draw_geometries([g.to_open3d_geometry(), frame])

g1 = Grasp()
gg = GraspGroup()
gg.add(g)
gg.add(g1)
```

(continues on next page)

(continued from previous page)

```
# GraspGroup before transformation
o3d.visualization.draw_geometries([*gg.to_open3d_geometry_list(), frame])

gg.transform(T)

# GraspGroup after transformation
o3d.visualization.draw_geometries([*gg.to_open3d_geometry_list(), frame])
```



CHAPTER

FOUR

# EXAMPLES

## 4.1 Check Dataset Files

You can check if there is any missing file in the dataset by the following code.

```
__author__ = 'mhgou'
__version__ = '1.0'

from graspnetAPI import GraspNet

# GraspNetAPI example for checking the data completeness.
# change the graspnet_root path

if __name__ == '__main__':

    ################################################################
    graspnet_root = '/home/gmh/graspnet'  ### ROOT PATH FOR GRASPNET ###
    ################################################################

    g = GraspNet(graspnet_root, 'kinect', 'all')
    if g.checkDataCompleteness():
        print('Check for kinect passed')


    g = GraspNet(graspnet_root, 'realsense', 'all')
    if g.checkDataCompleteness():
        print('Check for realsense passed')
```

## 4.2 Generating Rectangle Grasp Labels

You can generate the rectangle grasp labels by yourself.

Import necessary libs:

```
# GraspNetAPI example for generating rectangle grasp from 6d grasp.
# change the graspnet_root path and NUM_PROCESS

from graspnetAPI import GraspNet
from graspnetAPI.graspnet import TOTAL_SCENE_NUM
import os
import numpy as np
from tqdm import tqdm
```

Setup how many processes to use in generating the labels.

```
########################################################################
NUM_PROCESS = 24 # change NUM_PROCESS to the number of cores to use. #
########################################################################
```

The function to generate labels.

```
def generate_scene_rectangle_grasp(sceneId, dump_folder, camera):
    g = GraspNet(graspnet_root, camera=camera, split='all')
    objIds = g.getObjIds(sceneIds = sceneId)
    grasp_labels = g.loadGraspLabels(objIds)
    collision_labels = g.loadCollisionLabels(sceneIds = sceneId)
    scene_dir = os.path.join(dump_folder,'scene_%04d' % sceneId)
    if not os.path.exists(scene_dir):
        os.mkdir(scene_dir)
    camera_dir = os.path.join(scene_dir, camera)
    if not os.path.exists(camera_dir):
        os.mkdir(camera_dir)
    for annId in tqdm(range(256), 'Scene:{}, Camera:{}'.format(sceneId, camera)):
        _6d_grasp = g.loadGrasp(sceneId = sceneId, annId = annId, format = '6d',␣
↪camera = camera, grasp_labels = grasp_labels, collision_labels = collision_labels,␣
↪fric_coef_thresh = 1.0)
        rect_grasp_group = _6d_grasp.to_rect_grasp_group(camera)
        rect_grasp_group.save_npy(os.path.join(camera_dir, '%04d.npy' % annId))
```

Run the function for each scene and camera.

```
if __name__ == '__main__':
    ########################################################################
    graspnet_root = '/home/minghao/graspnet' # ROOT PATH FOR GRASPNET ##
    ########################################################################

    dump_folder = 'rect_labels'
    if not os.path.exists(dump_folder):
        os.mkdir(dump_folder)

    if NUM_PROCESS > 1:
        from multiprocessing import Pool
        pool = Pool(24)
        for camera in ['realsense', 'kinect']:
            for sceneId in range(120):
                pool.apply_async(func = generate_scene_rectangle_grasp, args =␣
↪(sceneId, dump_folder, camera))
        pool.close()
        pool.join()

    else:
        generate_scene_rectangle_grasp(sceneId, dump_folder, camera)
```

## 4.3 Loading Grasp Labels

Both *6d* and *rect* format labels can be loaded.

First, import relative libs.

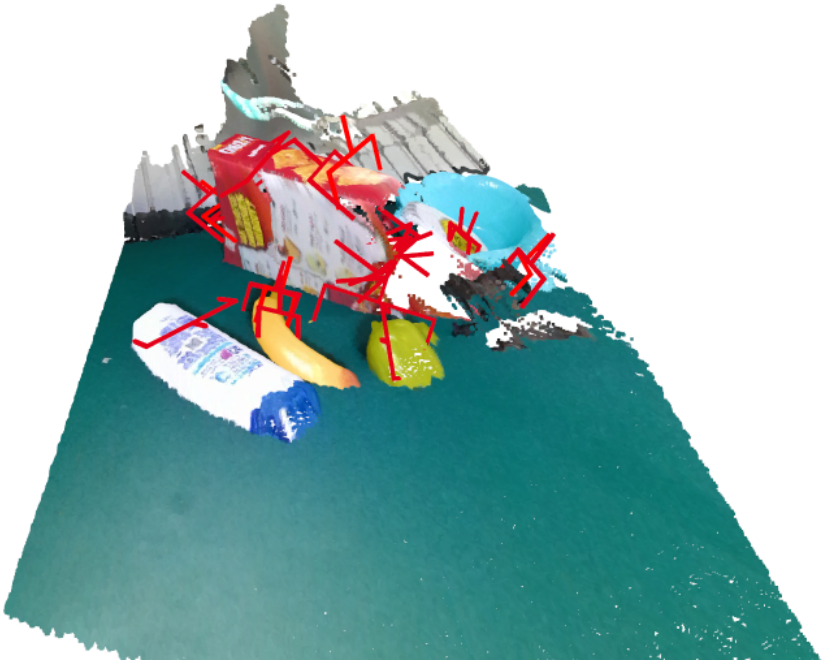
```
from graspnetAPI import GraspNet
import open3d as o3d
import cv2

```

Then, get a GraspNet instance and setup parameters.

```
############################################################################
graspnet_root = '/home/gmh/graspnet' # ROOT PATH FOR GRASPNET
############################################################################

sceneId = 1
annId = 3

# initialize a GraspNet instance
g = GraspNet(graspnet_root, camera='kinect', split='train')
```

Load GraspLabel in *6d* format and visulize the result.

```
# load grasps of scene 1 with annotation id = 3, camera = kinect and fric_coef_thresh␣
↪= 0.2
_6d_grasp = g.loadGrasp(sceneId = sceneId, annId = annId, format = '6d', camera =
↪'kinect', fric_coef_thresh = 0.2)
print('6d grasp:\n{}'.format(_6d_grasp))

# visualize the grasps using open3d
geometries = []
geometries.append(g.loadScenePointCloud(sceneId = sceneId, annId = annId, camera =
↪'kinect'))
geometries += _6d_grasp.random_sample(numGrasp = 20).to_open3d_geometry_list()
o3d.visualization.draw_geometries(geometries)
```



Load GraspLabel in *rect* format and visulize the result.

```
# load rectangle grasps of scene 1 with annotation id = 3, camera = realsense and␣
↪fric_coef_thresh = 0.2
rect_grasp = g.loadGrasp(sceneId = sceneId, annId = annId, format = 'rect', camera =
↪'realsense', fric_coef_thresh = 0.2)
print('rectangle grasp:\n{}'.format(rect_grasp))

# visualize the rectanglegrasps using opencv
bgr = g.loadBGR(sceneId = sceneId, annId = annId, camera = 'realsense')
img = rect_grasp.to_opencv_image(bgr, numGrasp = 20)
cv2.imshow('rectangle grasps', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```


## 4.4 Visualization of Dataset

Get a GraspNet instance.

```
##################################################################
graspnet_root = '/home/gmh/graspnet' # ROOT PATH FOR GRASPNET
##################################################################

from graspnetAPI import GraspNet

# initialize a GraspNet instance
g = GraspNet(graspnet_root, camera='kinect', split='train')
```

Show grasp labels on a object.

```
# show object grasps
g.showObjGrasp(objIds = 0, show=True)
```


Show 6D poses of objects in a scene.

```
# show 6d poses
g.show6DPose(sceneIds = 0, show = True)
```

Show Rectangle grasp labels in a scene.

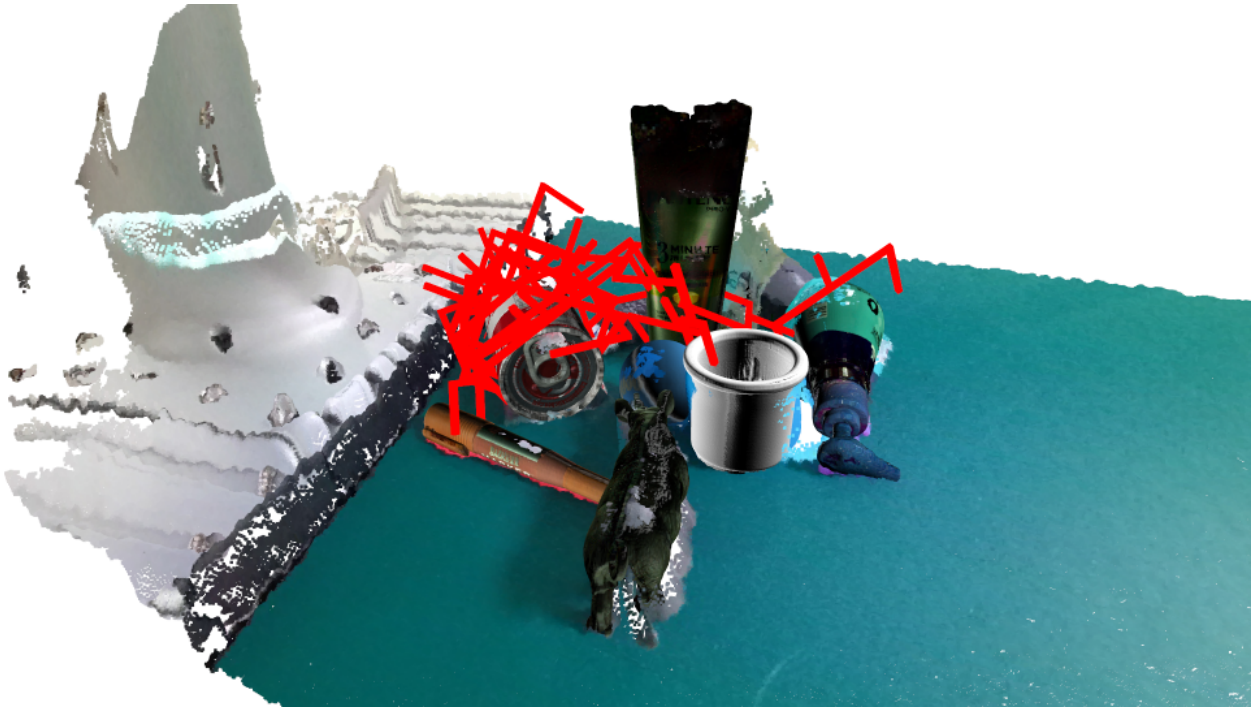
```
# show scene rectangle grasps
g.showSceneGrasp(sceneId = 0, camera = 'realsense', annId = 0, format = 'rect',␣
↪numGrasp = 20)
```

(continues on next page)

(continued from previous page)


Show 6D grasp labels in a scene.

```
# show scene 6d grasps(You may need to wait several minutes)
g.showSceneGrasp(sceneId = 4, camera = 'kinect', annId = 2, format = '6d')
```



## 4.5 Apply NMS on Grasps

Get a GraspNet instance.

```
# GraspNetAPI example for grasp nms.
# change the graspnet_root path

####################################################################
graspnet_root = '/home/gmh/graspnet' # ROOT PATH FOR GRASPNET
####################################################################

sceneId = 1
annId = 3

from graspnetAPI import GraspNet
import open3d as o3d
import cv2

# initialize a GraspNet instance
g = GraspNet(graspnet_root, camera='kinect', split='train')
```

Loading and visualizing grasp lables before NMS.

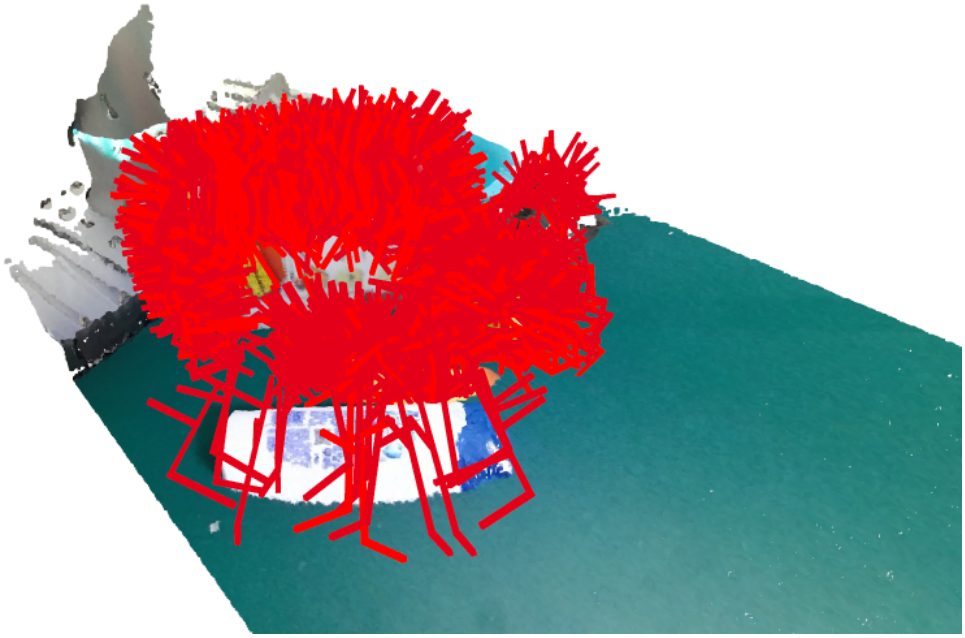
```
# load grasps of scene 1 with annotation id = 3, camera = kinect and fric_coef_thresh␣
↪= 0.2
_6d_grasp = g.loadGrasp(sceneId = sceneId, annId = annId, format = '6d', camera =
↪'kinect', fric_coef_thresh = 0.2)
print('6d grasp:\n{}'.format(_6d_grasp))

# visualize the grasps using open3d
geometries = []
geometries.append(g.loadScenePointCloud(sceneId = sceneId, annId = annId, camera =
↪'kinect'))
geometries += _6d_grasp.random_sample(numGrasp = 1000).to_open3d_geometry_list()
o3d.visualization.draw_geometries(geometries)
```

```
6d grasp:
----------
Grasp Group, Number=90332:
Grasp: score:0.9000000357627869, width:0.11247877031564713, height:0.
↪019999999552965164, depth:0.029999999329447746, translation:[-0.09166837 -0.
↪16910084  0.39480919]
rotation:
[[-0.81045675 -0.57493848  0.11227506]
 [ 0.49874267 -0.77775514 -0.38256073]
 [ 0.30727136 -0.25405255  0.91708326]]
object id:66
Grasp: score:0.9000000357627869, width:0.10030215978622437, height:0.
↪019999999552965164, depth:0.019999999552965164, translation:[-0.09166837 -0.
↪16910084  0.39480919]
rotation:
[[-0.73440629 -0.67870212  0.0033038 ]
 [ 0.64608938 -0.70059127 -0.3028869 ]
 [ 0.20788456 -0.22030747  0.95302087]]
object id:66
Grasp: score:0.9000000357627869, width:0.08487851172685623, height:0.
↪019999999552965164, depth:0.019999999552965164, translation:[-0.10412319 -0.
↪13797761  0.38312319]
```

(continues on next page)

(continued from previous page)

```
rotation:
[[ 0.03316294  0.78667939 -0.61647028]
[-0.47164679  0.55612743  0.68430364]
[ 0.88116372  0.26806271  0.38947764]]
object id:66
......
Grasp: score:0.9000000357627869, width:0.11909123510122299, height:0.
↪019999999552965164, depth:0.019999999552965164, translation:[-0.05140382  0.
↪11790846  0.48782501]
rotation:
[[-0.71453273  0.63476181 -0.2941435 ]
[-0.07400083  0.3495101   0.93400562]
[ 0.69567728  0.68914449 -0.20276351]]
object id:14
Grasp: score:0.9000000357627869, width:0.10943549126386642, height:0.
↪019999999552965164, depth:0.019999999552965164, translation:[-0.05140382  0.
↪11790846  0.48782501]
rotation:
[[ 0.08162415  0.4604325  -0.88393396]
[-0.52200603  0.77526748  0.3556262 ]
[ 0.84902728  0.4323912   0.30362913]]
object id:14
Grasp: score:0.9000000357627869, width:0.11654743552207947, height:0.
↪019999999552965164, depth:0.009999999776482582, translation:[-0.05140382  0.
↪11790846  0.48782501]
rotation:
[[-0.18380146  0.39686993 -0.89928377]
[-0.61254776  0.66926688  0.42055583]
[ 0.76876676  0.62815309  0.12008961]]
object id:14
------------
```



Apply nms to GraspGroup and visualizing the result.

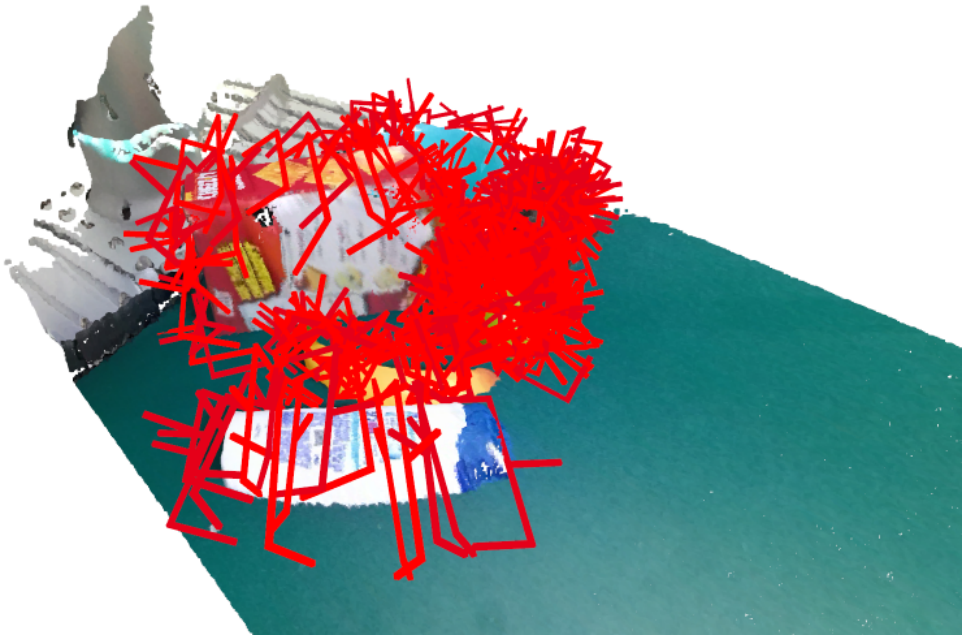
```
nms_grasp = _6d_grasp.nms(translation_thresh = 0.1, rotation_thresh = 30 / 180.0 * 3.
↪1416)
print('grasp after nms:\n{}'.format(nms_grasp))

# visualize the grasps using open3d
geometries = []
geometries.append(g.loadScenePointCloud(sceneId = sceneId, annId = annId, camera =
↪'kinect'))
geometries += nms_grasp.to_open3d_geometry_list()
o3d.visualization.draw_geometries(geometries)
```

```
grasp after nms:
----------
Grasp Group, Number=358:
Grasp: score:1.0, width:0.11948642134666443, height:0.019999999552965164, depth:0.
↪03999999910593033, translation:[-0.00363996  0.03692623  0.3311775 ]
rotation:
[[ 0.32641056 -0.8457799   0.42203382]
[-0.68102902 -0.52005678 -0.51550031]
[ 0.65548128 -0.11915252 -0.74575269]]
object id:0
Grasp: score:1.0, width:0.12185929715633392, height:0.019999999552965164, depth:0.
↪009999999776482582, translation:[-0.03486454  0.08384828  0.35117128]
rotation:
[[-0.00487804 -0.8475557   0.53068405]
[-0.27290785 -0.50941664 -0.81609803]
[ 0.96202785 -0.14880882 -0.22881967]]
object id:0
Grasp: score:1.0, width:0.04842342436313629, height:0.019999999552965164, depth:0.
↪019999999552965164, translation:[0.10816982 0.10254505 0.50272578]
rotation:
[[-0.98109186 -0.01696888 -0.19279723]
[-0.1817532   0.42313483  0.88765001]
[ 0.06651681  0.90590769 -0.41821831]]
object id:20
......
Grasp: score:0.9000000357627869, width:0.006192661356180906, height:0.
↪019999999552965164, depth:0.009999999776482582, translation:[0.0122985  0.29616502
↪0.53319722]
rotation:
[[-0.26423979  0.39734706  0.87880182]
[-0.95826042 -0.00504095 -0.28585231]
[-0.10915259 -0.91765451  0.38209397]]
object id:46
Grasp: score:0.9000000357627869, width:0.024634981527924538, height:0.
↪019999999552965164, depth:0.009999999776482582, translation:[0.11430283 0.18761221
↪0.51991153]
rotation:
[[-0.17379239 -0.96953499  0.17262182]
[-0.9434278   0.11365268 -0.31149188]
[ 0.28238329 -0.2169912  -0.93443805]]
object id:70
Grasp: score:0.9000000357627869, width:0.03459500893950462, height:0.
↪019999999552965164, depth:0.009999999776482582, translation:[0.02079188 0.11184558
↪0.50796509]
rotation:
[[ 0.38108557 -0.27480939  0.88275337]
```

(continues on next page)

(continued from previous page)

```
[-0.92043257 -0.20266907  0.33425891]
[ 0.08704928 -0.93989623 -0.33017775]]
object id:20
----------
```



## 4.6 Convert Labels between rectangle format and 6d format

Get a GraspNet instance.

```
from graspnetAPI import GraspNet
import cv2
import open3d as o3d

# GraspNetAPI example for checking the data completeness.
# change the graspnet_root path

camera = 'kinect'
sceneId = 5
annId = 3

################################################################
graspnet_root = '/home/gmh/graspnet' # ROOT PATH FOR GRASPNET
################################################################

g = GraspNet(graspnet_root, camera = camera, split = 'all')

bgr = g.loadBGR(sceneId = sceneId, camera = camera, annId = annId)
depth = g.loadDepth(sceneId = sceneId, camera = camera, annId = annId)
```

### 4.6.1 Convert rectangle format to 6d format

First, load rectangle labels from dataset.

```
# Rect to 6d
rect_grasp_group = g.loadGrasp(sceneId = sceneId, camera = camera, annId = annId,␣
↪fric_coef_thresh = 0.2, format = 'rect')
```

**Convert a single RectGrasp to Grasp.**

**Note:** This conversion may fail due to invalid depth information.

```
# RectGrasp to Grasp
rect_grasp = rect_grasp_group.random_sample(1)[0]
img = rect_grasp.to_opencv_image(bgr)

cv2.imshow('rect grasp', img)
cv2.waitKey(0)
cv2.destroyAllWindows()

grasp = rect_grasp.to_grasp(camera, depth)
if grasp is not None:
    geometry = []
    geometry.append(g.loadScenePointCloud(sceneId, camera, annId))
    geometry.append(grasp.to_open3d_geometry())
    o3d.visualization.draw_geometries(geometry)
else:
    print('No result because the depth is invalid, please try again!')
```

Before Conversion:

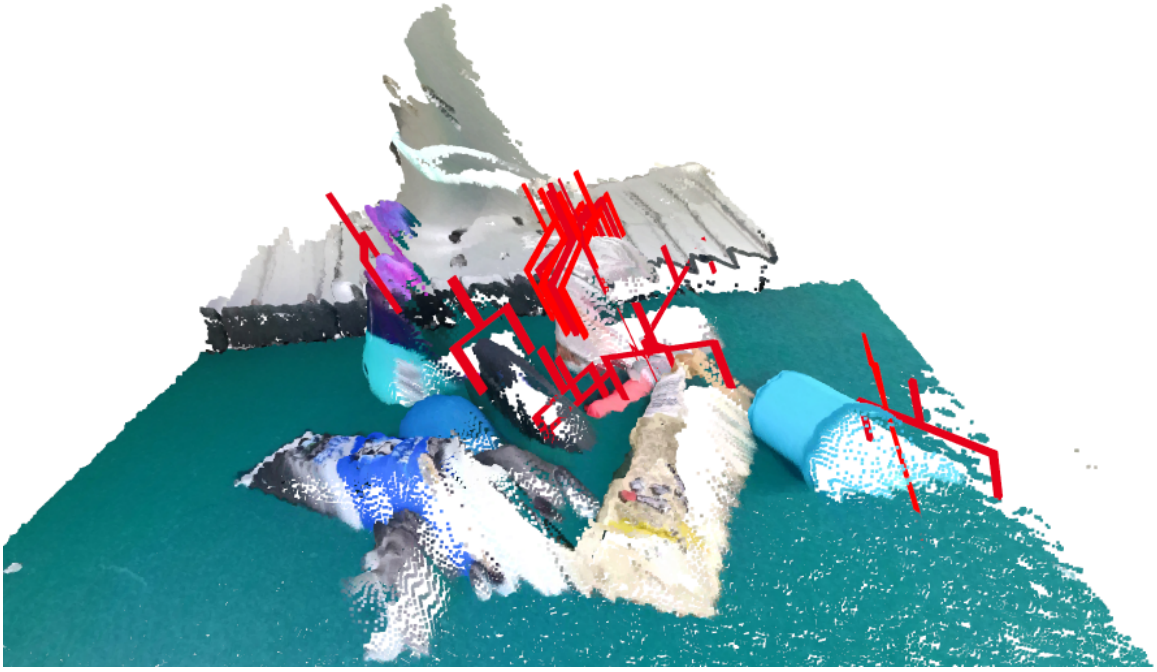
After Conversion:

**Convert RectGraspGroup to GraspGroup.**

```
# RectGraspGroup to GraspGroup
sample_rect_grasp_group = rect_grasp_group.random_sample(20)
img = sample_rect_grasp_group.to_opencv_image(bgr)
cv2.imshow('rect grasp', img)
cv2.waitKey(0)
cv2.destroyAllWindows()

grasp_group = sample_rect_grasp_group.to_grasp_group(camera, depth)
if grasp_group is not None:
    geometry = []
    geometry.append(g.loadScenePointCloud(sceneId, camera, annId))
    geometry += grasp_group.to_open3d_geometry_list()
    o3d.visualization.draw_geometries(geometry)
```

Before Conversion:

After Conversion:

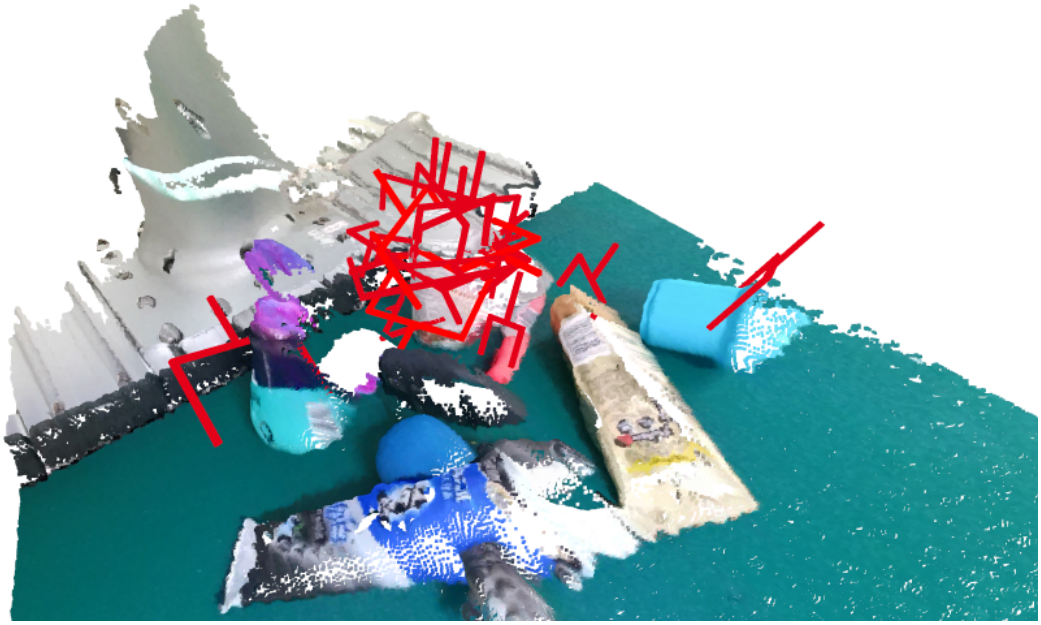


### 4.6.2 Convert 6d format to rectangle format

**Note:** Grasp to RectGrasp conversion is not applicable as only very few 6d grasp can be converted to rectangle grasp.

```
# 6d to Rect
_6d_grasp_group = g.loadGrasp(sceneId = sceneId, camera = camera, annId = annId, fric_
↪coef_thresh = 0.2, format = '6d')

# Grasp to RectGrasp conversion is not applicable as only very few 6d grasp can be␣
↪converted to rectangle grasp.

# GraspGroup to RectGraspGroup
sample_6d_grasp_group = _6d_grasp_group.random_sample(20)
geometry = []
geometry.append(g.loadScenePointCloud(sceneId, camera, annId))
geometry += sample_6d_grasp_group.to_open3d_geometry_list()
o3d.visualization.draw_geometries(geometry)

rect_grasp_group = _6d_grasp_group.to_rect_grasp_group(camera)
img = rect_grasp_group.to_opencv_image(bgr)

cv2.imshow('rect grasps', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Before Conversion:



After Conversion:

## 4.7 Evaluation

### 4.7.1 Data Preparation

The first step of evaluation is to prepare your own results. You need to run your code and generate a *GraspGroup* for each image in each scene. Then call the *save_npy* function of *GraspGroup* to dump the results.

To generate a *GraspGroup* and save it, you can directly input a 2D numpy array for the *GraspGroup* class:

```
gg=GraspGroup(np.array([[score_1, width_1, height_1, depth_1, rotation_matrix_1(9),␣
↪translation_1(3), object_id_1],
                        [score_2, width_2, height_2, depth_2, rotation_matrix_2(9),␣
↪translation_2(3), object_id_2],
                        ...,
                        [score_N, width_N, height_N, depth_N, rotation_matrix_N(9),␣
↪translation_N(3), object_id_N]]
              ))
gg.save_npy(save_path)
```

where your algorithm predicts N grasp poses for an image. For the *object_id*, you can simply input *0*. For the meaning of other entries, you should refer to the doc for Grasp Label Format-API Loaded Labels

The file structure of dump folder should be as follows:

```
|-- dump_folder
    |-- scene_0100
    |   |-- kinect
    |   |   |
    |   |   --- 0000.npy to 0255.npy
    |   |
    |   --- realsense
```

(continues on next page)

(continued from previous page)

```
    |       |
    |       --- 0000.npy to 0255.npy
    |
    |-- scene_0101
    |
    ...
    |
    --- scene_0189
```

You can choose to generate dump files for only one camera, there will be no error for doing that.

### 4.7.2 Evaluation API

Get GraspNetEval instances.

```
# GraspNetAPI example for evaluate grasps for a scene.
# change the graspnet_root path
import numpy as np
from graspnetAPI import GraspNetEval

##################################################################
graspnet_root = '/home/gmh/graspnet' # ROOT PATH FOR GRASPNET
dump_folder = '/home/gmh/git/rgbd_graspnet/dump_affordance_iounan/' # ROOT PATH FOR␣
↪DUMP
##################################################################

sceneId = 121
camera = 'kinect'
ge_k = GraspNetEval(root = graspnet_root, camera = 'kinect', split = 'test')
ge_r = GraspNetEval(root = graspnet_root, camera = 'realsense', split = 'test')
```

#### Evaluate A Single Scene

```
# eval a single scene
print('Evaluating scene:{}, camera:{}'.format(sceneId, camera))
acc = ge_k.eval_scene(scene_id = sceneId, dump_folder = dump_folder)
np_acc = np.array(acc)
print('mean accuracy:{}'.format(np.mean(np_acc)))
```

#### Evaluate All Scenes

```
# # eval all data for kinect
# print('Evaluating kinect')
# res, ap = ge_k.eval_all(dump_folder, proc = 24)
```

### Evaluate 'Seen' Split

```
# # eval 'seen' split for realsense
# print('Evaluating realsense')
# res, ap = ge_r.eval_seen(dump_folder, proc = 24)
```

CHAPTER

FIVE

# PYTHON API

## 5.1 graspnetAPI package

### 5.1.1 Subpackages

#### graspnetAPI.utils package

#### Subpackages

#### graspnetAPI.utils.dexnet package

#### Subpackages

#### graspnetAPI.utils.dexnet.grasping package

#### Subpackages

#### graspnetAPI.utils.dexnet.grasping.meshpy package

#### Submodules

#### graspnetAPI.utils.dexnet.grasping.meshpy.mesh module

Encapsulates mesh for grasping operations Authors: Jeff Mahler and Matt Matl

**class** graspnetAPI.utils.dexnet.grasping.meshpy.mesh.**Mesh3D**(*vertices*, *triangles*, *normals=None*, *density=1.0*, *center_of_mass=None*, *trimesh=None*, *T_obj_world=RigidTransform(rotation=np.arr 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]), translation=np.array([0.0, 0.0, 0.0]), from_frame='obj', to_frame='world'*)

Bases: object

A triangular mesh for a three-dimensional shape representation.

**vertices** [numpy.ndarray of float] A #verts by 3 array, where each row contains an ordered [x,y,z] set that describes one vertex.

**triangles** [numpy.ndarray of int] A #tris by 3 array, where each row contains indices of vertices in the *vertices* array that are part of the triangle.

**normals** [numpy.ndarray of float] A #normals by 3 array, where each row contains a normalized vector. This list should contain one norm per vertex.

**density** [float] The density of the mesh.

**center_of_mass** [numpy.ndarray of float] The 3D location of the mesh's center of mass.

**mass** [float] The mass of the mesh (read-only).

**inertia** [numpy.ndarray of float] The 3x3 inertial matrix of the mesh (read-only).

**bb_center** [numpy.ndarray of float] The 3D location of the center of the mesh's minimal bounding box (read-only).

**centroid** [numpy.ndarray of float] The 3D location of the mesh's vertex mean (read-only).

**C_canonical = array([[0.01666667, 0.00833333, 0.00833333], [0.00833333, 0.01666667, 0.**

**OBJ_EXT = '.obj'**

**PROC_TAG = '_proc'**

**ScalingTypeDiag = 4**

**ScalingTypeMax = 2**

**ScalingTypeMed = 1**

**ScalingTypeMin = 0**

**ScalingTypeRelative = 3**

**property T_obj_world**
Return pose.

**property bb_center**
numpy.ndarray of float : The 3D location of the center of the mesh's minimal bounding box (read-only).

**bounding_box**()
Returns the mesh's bounding box corners.

**tuple of numpy.ndarray of float** A 2-tuple of 3-ndarrays of floats. The first 3-array contains the vertex of the smallest corner of the bounding box, and the second 3-array contains the largest corner of the bounding box.

**bounding_box_mesh**()
Returns the mesh bounding box as a mesh.

***Mesh3D*** A Mesh3D representation of the mesh's bounding box.

**property center_of_mass**
numpy.ndarray of float : The 3D location of the mesh's center of mass.

**center_vertices**()
Center the mesh's vertices on the mesh center of mass.

This shifts the mesh without rotating it so that the center of its bounding box is at the origin.

**center_vertices_avg**()
Center the mesh's vertices at the centroid.

This shifts the mesh without rotating it so that the centroid (mean) of all vertices is at the origin.

**center_vertices_bb**()
Center the mesh's vertices at the center of its bounding box.

This shifts the mesh without rotating it so that the center of its bounding box is at the origin.

**property centroid**
`numpy.ndarray` of float : The 3D location of the mesh's vertex mean (read-only).

**compute_vertex_normals**()
Get normals from triangles

**convex_hull**()
Return a 3D mesh that represents the convex hull of the mesh.

**copy**()
Return a copy of the mesh.

This method only copies the vertices and triangles of the mesh.

**covariance**()
Return the total covariance of the mesh's triangles.

**float** The total covariance of the mesh's triangles.

**property density**
float : The density of the mesh.

**find_contact**(*origin*, *direction*)
Finds the contact location with the mesh, if it exists.

**flip_normals**()
Flips the mesh normals.

**flip_tri_orientation**()
Flips the orientation of all triangles.

**get_T_surface_obj**(*T_obj_surface*, *delta=0.0*)
Gets the transformation that puts the object resting exactly on the z=delta plane

**T_obj_surface** [`RigidTransform`] The RigidTransform by which the mesh is transformed.

**delta** [float] Z-coordinate to rest the mesh on

This method copies the vertices and triangles of the mesh.

**property inertia**
`numpy.ndarray` of float : The 3x3 inertial matrix of the mesh (read-only).

**property is_watertight**

**static load**(*filename*, *cache_dir*, *preproc_script=None*)
Load a mesh from a file.

If the mesh is not already in .obj format, this requires the installation of meshlab. Meshlab has a command called meshlabserver that is used to convert the file into a .obj format.

**filename** [`str`] Path to mesh file.

**cache_dir** [`str`] A directory to store a converted .obj file in, if the file isn't already in .obj format.

**preproc_script** [str] The path to an optional script to run before converting the mesh file to .obj if necessary.

***Mesh3D*** A 3D mesh object read from the file.

**property mass**
float : The mass of the mesh (read-only).

**max_coords**()
Returns the maximum coordinates of the mesh.

**numpy.ndarray of float** A 3-ndarray of floats that represents the minimal x, y, and z coordinates represented in the mesh.

**merge**(*other_mesh*)
Combines this mesh with another mesh.

**other_mesh** [*Mesh3D*] the mesh to combine with

***Mesh3D*** merged mesh

**min_coords**()
Returns the minimum coordinates of the mesh.

**numpy.ndarray of float** A 3-ndarray of floats that represents the minimal x, y, and z coordinates represented in the mesh.

**normalize_vertices**()
Normalize the mesh's orientation along its principal axes.

Transforms the vertices and normals of the mesh such that the origin of the resulting mesh's coordinate frame is at the center of the bounding box and the principal axes (as determined from PCA) are aligned with the vertical Z, Y, and X axes in that order.

**property normals**
numpy.ndarray of float : A #normals by 3 array, where each row contains a normalized vector. This list should contain one norm per vertex.

**property num_triangles**
int: The number of total triangles

**property num_vertices**
int: The number of total vertices

**principal_dims**()
Returns the maximal span of the mesh's coordinates.

The maximal span is the maximum coordinate value minus the minimal coordinate value in each principal axis.

**numpy.ndarray of float** A 3-ndarray of floats that represents the maximal x, y, and z spans of the mesh.

**random_points**(*n_points*)
Generate uniformly random points on the surface of the mesh.

**n_points** [int] The number of random points to generate.

**numpy.ndarray of float** A n_points by 3 ndarray that contains the sampled 3D points.

**ray_intersections**(*ray*, *point*, *distance*)
Returns a list containing the indices of the triangles that are intersected by the given ray emanating from the given point within some distance.

**remove_bad_tris**()
Remove triangles with out-of-bounds vertices from the mesh.

**remove_unreferenced_vertices**()
Remove any vertices that are not part of a triangular face.

This method will fail if any bad triangles are present, so run remove_bad_tris() first if you're unsure if bad triangles are present.

**bool** Returns True if vertices were removed, False otherwise.

**rescale**(*scale_factor*)
Rescales the vertex coordinates by scale_factor.

**scale_factor** [float] The desired scale factor for the mesh's vertices.

**rescale_dimension**(*scale*, *scaling_type=0*)
Rescales the vertex coordinates to scale using the given scaling_type.

**scale** [float] The desired scaling factor of the selected dimension, if scaling_type is ScalingTypeMin, ScalingTypeMed, ScalingTypeMax, or ScalingTypeDiag. Otherwise, the overall scaling factor.

**scaling_type** [int] One of ScalingTypeMin, ScalingTypeMed, ScalingTypeMax, ScalingTypeRelative, or ScalingTypeDiag. ScalingTypeMin scales the smallest vertex extent (X, Y, or Z) by scale, ScalingTypeMed scales the median vertex extent, and ScalingTypeMax scales the maximum vertex extent. ScalingTypeDiag scales the bounding box diagonal (divided by three), and ScalingTypeRelative provides absolute scaling.

**resting_pose**(*T_obj_world*, *eps=1e-10*)
Returns the stable pose that the mesh will rest on if it lands on an infinite planar worksurface quasi-statically in the given transformation (only the rotation is used).

**T_obj_world** [`autolab_core.RigidTransform`] transformation from object to table basis (z-axis upward) specifying the orientation of the mesh

**eps** [float] numeric tolerance in cone projection solver

**StablePose** stable pose specifying the face that the mesh will land on

**scale_principal_eigenvalues**(*new_evals*)

**stable_poses**(*min_prob=0.0*)
Computes all valid StablePose objects for the mesh.

**min_prob** [float] stable poses that are less likely than this threshold will be discarded

**list of StablePose** A list of StablePose objects for the mesh.

**subdivide**(*min_tri_length=inf*)
Return a copy of the mesh that has been subdivided by one iteration.

This method only copies the vertices and triangles of the mesh.

**support**(*direction*)
Returns the support function in the given direction

**direction** [`numpy.ndarray` of float] A 3-ndarray of floats that is a unit vector in the direction of the desired support.

**numpy.ndarray of float** A 3-ndarray of floats that represents the support.

**surface_area**()
Return the surface area of the mesh.

**float** The surface area of the mesh.

**total_volume**()
Return the total volume of the mesh.

**float** The total volume of the mesh.

**transform**(*T*)
Return a copy of the mesh that has been transformed by T.

**T** [RigidTransform] The RigidTransform by which the mesh is transformed.

This method only copies the vertices and triangles of the mesh.

**tri_centers**()
Returns an array of the triangle centers as 3D points.

**numpy.ndarray of numpy.ndarray of float** An ndarray of 3-ndarrays of floats, where each 3-ndarray represents the 3D point at the center of the corresponding mesh triangle.

**tri_normals**(*align_to_hull=False*)
Returns a list of the triangle normals.

**align_to_hull** [bool] If true, we re-orient the normals to point outward from the mesh by using the convex hull.

**numpy.ndarray of float** A #triangles by 3 array of floats, where each 3-ndarray represents the 3D normal vector of the corresponding triangle.

**property triangles**
numpy.ndarray of int : A #tris by 3 array, where each row contains indices of vertices in the *vertices* array that are part of the triangle.

**property trimesh**
Convert to trimesh.

**update_tf**(*delta_T*)
Updates the mesh transformation.

**property vertices**
numpy.ndarray of float : A #verts by 3 array, where each row contains an ordered [x,y,z] set that describes one vertex.

**visualize**(*color=0.5, 0.5, 0.5*, *style='surface'*, *opacity=1.0*)
Plots visualization of mesh using MayaVI.

**color** [tuple of float] 3-tuple of floats in [0,1] to give the mesh's color

**style** [str] Either 'surface', which produces an opaque surface, or 'wireframe', which produces a wireframe.

**opacity** [float] A value in [0,1] indicating the opacity of the mesh. Zero is transparent, one is opaque.

**mayavi.modules.surface.Surface** The displayed surface.

### graspnetAPI.utils.dexnet.grasping.meshpy.obj_file module

File for loading and saving meshes from .OBJ files Author: Jeff Mahler

**class** graspnetAPI.utils.dexnet.grasping.meshpy.obj_file.**ObjFile**(*filepath*)
Bases: object

A Wavefront .obj file reader and writer.

**filepath** [str] The full path to the .obj file associated with this reader/writer.

**property filepath**
Returns the full path to the .obj file associated with this reader/writer.

**str** The full path to the .obj file associated with this reader/writer.

**read**()
Reads in the .obj file and returns a Mesh3D representation of that mesh.

**Mesh3D** A Mesh3D created from the data in the .obj file.

**write**(*mesh*)
Writes a Mesh3D object out to a .obj file format

**mesh** [Mesh3D] The Mesh3D object to write to the .obj file.

Does not support material files or texture coordinates.

### graspnetAPI.utils.dexnet.grasping.meshpy.sdf module

Definition of SDF Class Author: Sahaana Suri, Jeff Mahler, and Matt Matl

**Currently assumes clean input**

**class** graspnetAPI.utils.dexnet.grasping.meshpy.sdf.**Sdf**
Bases: object

Abstract class for signed distance fields.

**property center**
Center of grid.

This basically transforms the world frame to grid center.

numpy.ndarray

**center_world**()
Center of grid (basically transforms world frame to grid center)

**property data**
The SDF data.

**numpy.ndarray of float** The 2- or 3-dimensional ndarray that holds the grid of signed distances.

**property dimensions**
SDF dimension information.

**numpy.ndarray of int** The ndarray that contains the dimensions of the sdf.

**property gradients**
Gradients of the SDF.

**list of numpy.ndarray of float** A list of ndarrays of the same dimension as the SDF. The arrays are in axis order and specify the gradients for that axis at each point.

**is_out_of_bounds**(*coords*)
Returns True if coords is an out of bounds access.

**coords** [numpy.ndarray of int] A 2- or 3-dimensional ndarray that indicates the desired coordinates in the grid.

**bool** Are the coordinates in coords out of bounds?

**on_surface**(*coords*)
Determines whether or not a point is on the object surface.

**coords** [numpy.ndarray of int] A 2- or 3-dimensional ndarray that indicates the desired coordinates in the grid.

**tuple of bool, float** Is the point on the object's surface, and what is the signed distance at that point?

**property origin**
The location of the origin in the SDF grid.

**numpy.ndarray of float** The 2- or 3-ndarray that contains the location of the origin of the mesh grid in real space.

**property resolution**
The grid resolution (how wide each grid cell is).

**float** The width of each grid cell.

**abstract surface_points**()
Returns the points on the surface.

**tuple of numpy.ndarray of int, numpy.ndarray of float** The points on the surface and the signed distances at those points.

**abstract transform**(*tf*)
Returns a new SDF transformed by similarity tf.

**abstract transform_pt_grid_to_obj**(*x_grid*, *direction=False*)
Transforms points from grid frame to world frame

**abstract transform_pt_obj_to_grid**(*x_world*, *direction=False*)
Transforms points from world frame to grid frame

**transform_to_world**()
Returns an sdf object with center in the world frame of reference.

**class** graspnetAPI.utils.dexnet.grasping.meshpy.sdf.**Sdf3D**(*sdf_data*, *origin*, *resolution*, *use_abs=False*, *T_sdf_world=RigidTransform(rotation=np.array(*
*0.0, 0.0], [0.0, 1.0, 0.0],*
*[0.0, 0.0, 1.0]]), translation=np.array([0.0, 0.0,*
*0.0]), from_frame='sdf',*
*to_frame='world')*)
Bases: *graspnetAPI.utils.dexnet.grasping.meshpy.sdf.Sdf*

**curvature**(*coords*, *delta=0.001*)
Returns an approximation to the local SDF curvature (Hessian) at the given coordinate in grid basis.

**coords** [numpy 3-vector] the grid coordinates at which to get the curvature

delta : Returns ——- curvature : 3x3 ndarray of the curvature at the surface points

**static find_zero_crossing_linear**(*x1*, *y1*, *x2*, *y2*)
Find zero crossing using linear approximation

**static find_zero_crossing_quadratic**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *eps=1.0*)
Find zero crossing using quadratic approximation along 1d line

**gradient**(*coords*)
Returns the SDF gradient at the given coordinates, interpolating if necessary

**coords** [numpy.ndarray of int] A 3-dimensional ndarray that indicates the desired coordinates in the grid.

**float** The gradient at the given coords (interpolated).

**IndexError** If the coords vector does not have three entries.

**max_coords_x = [1, 4, 6, 7]**

**max_coords_y = [2, 4, 5, 7]**

**max_coords_z = [3, 5, 6, 7]**

**min_coords_x = [0, 2, 3, 5]**

**min_coords_y = [0, 1, 3, 6]**

**min_coords_z = [0, 1, 2, 4]**

**num_interpolants = 8**

**rescale**(*scale*)
Rescale an SDF by a given scale factor.

**scale** [float] the amount to scale the SDF

***Sdf3D*** new sdf with given scale

**surface_normal**(*coords*, *delta=1.5*)
Returns the sdf surface normal at the given coordinates by computing the tangent plane using SDF interpolation.

**coords** [numpy.ndarray of int] A 3-dimensional ndarray that indicates the desired coordinates in the grid.

**delta** [float] A radius for collecting surface points near the target coords for calculating the surface normal.

**numpy.ndarray of float** The 3-dimensional ndarray that represents the surface normal.

**IndexError** If the coords vector does not have three entries.

**surface_points**(*grid_basis=True*)
Returns the points on the surface.

**grid_basis** [bool] If False, the surface points are transformed to the world frame. If True (default), the surface points are left in grid coordinates.

**tuple of numpy.ndarray of int, numpy.ndarray of float** The points on the surface and the signed distances at those points.

**transform**(*delta_T*)
Creates a new SDF with a given pose with respect to world coordinates.

**delta_T** [autolab_core.RigidTransform] transform from cur sdf to transformed sdf coords

**transform_dense**(*delta_T*, *detailed=False*)
Transform the grid by pose T and scale with canonical reference frame at the SDF center with axis alignment.

**delta_T** [SimilarityTransform] the transformation from the current frame of reference to the new frame of reference

**detailed** [bool] whether or not to use interpolation

***Sdf3D*** new sdf with grid warped by T

**transform_pt_grid_to_obj**(*x_grid*, *direction=False*)
Converts a point in grid coords to the world basis. If direction then don't translate.

**x_grid** [numpy 3xN ndarray or numeric scalar] points to transform from grid basis to sdf basis in meters

direction : bool Returns ——- x_sdf : numpy 3xN ndarray

points in sdf basis (meters)

**transform_pt_obj_to_grid**(*x_sdf*, *direction=False*)
Converts a point in sdf coords to the grid basis. If direction then don't translate.

**x_sdf** [numpy 3xN ndarray or numeric scalar] points to transform from sdf basis in meters to grid basis

direction : bool Returns ——- x_grid : numpy 3xN ndarray or scalar

points in grid basis

### graspnetAPI.utils.dexnet.grasping.meshpy.sdf_file module

Reads and writes sdfs to file Author: Jeff Mahler

**class** graspnetAPI.utils.dexnet.grasping.meshpy.sdf_file.**SdfFile**(*filepath*)
Bases: object

A Signed Distance Field .sdf file reader and writer.

**filepath** [str] The full path to the .sdf or .csv file associated with this reader/writer.

**property filepath**
Returns the full path to the file associated with this reader/writer.

**str** The full path to the file associated with this reader/writer.

**read**()
Reads in the SDF file and returns a Sdf object.

**Sdf** A Sdf created from the data in the file.

**write**(*sdf*)
Writes an SDF to a file.

**sdf** [Sdf] An Sdf object to write out.

This is not currently implemented or supported.

### graspnetAPI.utils.dexnet.grasping.meshpy.stable_pose module

A basic struct-like Stable Pose class to make accessing pose probability and rotation matrix easier

Author: Matt Matl and Nikhil Sharma

**class** graspnetAPI.utils.dexnet.grasping.meshpy.stable_pose.**StablePose**(*p*, *r*, *x0*, *face=None*, *stp_id=-1*)

Bases: object

A representation of a mesh's stable pose.

**p** [float] Probability associated with this stable pose.

**r** [numpy.ndarray of :obj`numpy.ndarray` of float] 3x3 rotation matrix that rotates the mesh into the stable pose from standardized coordinates.

**x0** [numpy.ndarray of float] 3D point in the mesh that is resting on the table.

**face** [numpy.ndarray] 3D vector of indices corresponding to vertices forming the resting face

**stp_id** [str] A string identifier for the stable pose

**T_obj_table** [RigidTransform] A RigidTransform representation of the pose's rotation matrix.

**property T_obj_table**

**property T_obj_world**

### Module contents

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**Mesh3D**(*vertices*, *triangles*, *normals=None*, *density=1.0*, *center_of_mass=None*, *trimesh=None*, *T_obj_world=RigidTransform(rotation=np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]), translation=np.array([0.0, 0.0, 0.0]), from_frame='obj', to_frame='world')*)

Bases: object

A triangular mesh for a three-dimensional shape representation.

**vertices** [numpy.ndarray of float] A #verts by 3 array, where each row contains an ordered [x,y,z] set that describes one vertex.

**triangles** [numpy.ndarray of int] A #tris by 3 array, where each row contains indices of vertices in the *vertices* array that are part of the triangle.

**normals** [numpy.ndarray of float] A #normals by 3 array, where each row contains a normalized vector. This list should contain one norm per vertex.

**density** [float] The density of the mesh.

**center_of_mass** [numpy.ndarray of float] The 3D location of the mesh's center of mass.

**mass** [float] The mass of the mesh (read-only).

**inertia** [numpy.ndarray of float] The 3x3 inertial matrix of the mesh (read-only).

**bb_center** [numpy.ndarray of float] The 3D location of the center of the mesh's minimal bounding box (read-only).

**centroid** [numpy.ndarray of float] The 3D location of the mesh's vertex mean (read-only).

**C_canonical = array([[0.01666667, 0.00833333, 0.00833333], [0.00833333, 0.01666667, 0.**

**OBJ_EXT = '.obj'**

**PROC_TAG = '_proc'**

**ScalingTypeDiag = 4**

**ScalingTypeMax = 2**

**ScalingTypeMed = 1**

**ScalingTypeMin = 0**

**ScalingTypeRelative = 3**

**property T_obj_world**
Return pose.

**property bb_center**
numpy.ndarray of float : The 3D location of the center of the mesh's minimal bounding box (read-only).

**bounding_box**()
Returns the mesh's bounding box corners.

**tuple of numpy.ndarray of float** A 2-tuple of 3-ndarrays of floats. The first 3-array contains the vertex of the smallest corner of the bounding box, and the second 3-array contains the largest corner of the bounding box.

**bounding_box_mesh**()
Returns the mesh bounding box as a mesh.

*Mesh3D* A Mesh3D representation of the mesh's bounding box.

**property center_of_mass**
numpy.ndarray of float : The 3D location of the mesh's center of mass.

**center_vertices**()
Center the mesh's vertices on the mesh center of mass.

This shifts the mesh without rotating it so that the center of its bounding box is at the origin.

**center_vertices_avg**()
Center the mesh's vertices at the centroid.

This shifts the mesh without rotating it so that the centroid (mean) of all vertices is at the origin.

**center_vertices_bb**()
Center the mesh's vertices at the center of its bounding box.

This shifts the mesh without rotating it so that the center of its bounding box is at the origin.

**property centroid**
numpy.ndarray of float : The 3D location of the mesh's vertex mean (read-only).

**compute_vertex_normals**()
Get normals from triangles

**convex_hull**()
Return a 3D mesh that represents the convex hull of the mesh.

**copy**()
Return a copy of the mesh.

This method only copies the vertices and triangles of the mesh.

**covariance**()
Return the total covariance of the mesh's triangles.

**float** The total covariance of the mesh's triangles.

**property density**
float : The density of the mesh.

**find_contact**(*origin*, *direction*)
Finds the contact location with the mesh, if it exists.

**flip_normals**()
Flips the mesh normals.

**flip_tri_orientation**()
Flips the orientation of all triangles.

**get_T_surface_obj**(*T_obj_surface*, *delta=0.0*)
Gets the transformation that puts the object resting exactly on the z=delta plane

**T_obj_surface** [`RigidTransform`] The RigidTransform by which the mesh is transformed.

**delta** [float] Z-coordinate to rest the mesh on

This method copies the vertices and triangles of the mesh.

**property inertia**
`numpy.ndarray` of float : The 3x3 inertial matrix of the mesh (read-only).

**property is_watertight**

**static load**(*filename*, *cache_dir*, *preproc_script=None*)
Load a mesh from a file.

If the mesh is not already in .obj format, this requires the installation of meshlab. Meshlab has a command called meshlabserver that is used to convert the file into a .obj format.

**filename** [`str`] Path to mesh file.

**cache_dir** [`str`] A directory to store a converted .obj file in, if the file isn't already in .obj format.

**preproc_script** [`str`] The path to an optional script to run before converting the mesh file to .obj if necessary.

***Mesh3D*** A 3D mesh object read from the file.

**property mass**
float : The mass of the mesh (read-only).

**max_coords**()
Returns the maximum coordinates of the mesh.

**numpy.ndarray of float** A 3-ndarray of floats that represents the minimal x, y, and z coordinates represented in the mesh.

**merge**(*other_mesh*)
Combines this mesh with another mesh.

**other_mesh** [*Mesh3D*] the mesh to combine with

***Mesh3D*** merged mesh

**min_coords**()
Returns the minimum coordinates of the mesh.

**numpy.ndarray of float** A 3-ndarray of floats that represents the minimal x, y, and z coordinates represented in the mesh.

**normalize_vertices**()
Normalize the mesh's orientation along its principal axes.

Transforms the vertices and normals of the mesh such that the origin of the resulting mesh's coordinate frame is at the center of the bounding box and the principal axes (as determined from PCA) are aligned with the vertical Z, Y, and X axes in that order.

**property normals**
`numpy.ndarray` of float : A #normals by 3 array, where each row contains a normalized vector. This list should contain one norm per vertex.

**property num_triangles**
`int`: The number of total triangles

**property num_vertices**
`int`: The number of total vertices

**principal_dims**()
Returns the maximal span of the mesh's coordinates.

The maximal span is the maximum coordinate value minus the minimal coordinate value in each principal axis.

**numpy.ndarray of float** A 3-ndarray of floats that represents the maximal x, y, and z spans of the mesh.

**random_points**(*n_points*)
Generate uniformly random points on the surface of the mesh.

**n_points** [int] The number of random points to generate.

**numpy.ndarray of float** A n_points by 3 ndarray that contains the sampled 3D points.

**ray_intersections**(*ray*, *point*, *distance*)
Returns a list containing the indices of the triangles that are intersected by the given ray emanating from the given point within some distance.

**remove_bad_tris**()
Remove triangles with out-of-bounds vertices from the mesh.

**remove_unreferenced_vertices**()
Remove any vertices that are not part of a triangular face.

This method will fail if any bad triangles are present, so run remove_bad_tris() first if you're unsure if bad triangles are present.

**bool** Returns True if vertices were removed, False otherwise.

**rescale**(*scale_factor*)
Rescales the vertex coordinates by scale_factor.

**scale_factor** [float] The desired scale factor for the mesh's vertices.

**rescale_dimension**(*scale*, *scaling_type=0*)
Rescales the vertex coordinates to scale using the given scaling_type.

**scale** [float] The desired scaling factor of the selected dimension, if scaling_type is ScalingTypeMin, ScalingTypeMed, ScalingTypeMax, or ScalingTypeDiag. Otherwise, the overall scaling factor.

**scaling_type** [int] One of ScalingTypeMin, ScalingTypeMed, ScalingTypeMax, ScalingTypeRelative, or ScalingTypeDiag. ScalingTypeMin scales the smallest vertex extent (X, Y, or Z) by scale, ScalingTypeMed scales the median vertex extent, and ScalingTypeMax scales the maximum vertex extent. ScalingTypeDiag scales the bounding box diagonal (divided by three), and ScalingTypeRelative provides absolute scaling.

**resting_pose**(*T_obj_world*, *eps=1e-10*)
Returns the stable pose that the mesh will rest on if it lands on an infinite planar worksurface quasi-statically in the given transformation (only the rotation is used).

**T_obj_world** [`autolab_core.RigidTransform`] transformation from object to table basis (z-axis upward) specifying the orientation of the mesh

**eps** [float] numeric tolerance in cone projection solver

***StablePose*** stable pose specifying the face that the mesh will land on

**scale_principal_eigenvalues**(*new_evals*)

**stable_poses**(*min_prob=0.0*)
Computes all valid StablePose objects for the mesh.

**min_prob** [float] stable poses that are less likely than this threshold will be discarded

**list of *StablePose*** A list of StablePose objects for the mesh.

**subdivide**(*min_tri_length=inf*)
Return a copy of the mesh that has been subdivided by one iteration.

This method only copies the vertices and triangles of the mesh.

**support**(*direction*)
Returns the support function in the given direction

**direction** [`numpy.ndarray` of float] A 3-ndarray of floats that is a unit vector in the direction of the desired support.

**numpy.ndarray of float** A 3-ndarray of floats that represents the support.

**surface_area**()
Return the surface area of the mesh.

**float** The surface area of the mesh.

**total_volume**()
Return the total volume of the mesh.

**float** The total volume of the mesh.

**transform**(*T*)
Return a copy of the mesh that has been transformed by T.

**T** [`RigidTransform`] The RigidTransform by which the mesh is transformed.

This method only copies the vertices and triangles of the mesh.

**tri_centers**()
Returns an array of the triangle centers as 3D points.

**numpy.ndarray of numpy.ndarray of float** An ndarray of 3-ndarrays of floats, where each 3-ndarray represents the 3D point at the center of the corresponding mesh triangle.

**tri_normals**(*align_to_hull=False*)
Returns a list of the triangle normals.

**align_to_hull** [bool] If true, we re-orient the normals to point outward from the mesh by using the convex hull.

**numpy.ndarray of float** A #triangles by 3 array of floats, where each 3-ndarray represents the 3D normal vector of the corresponding triangle.

**property triangles**
numpy.ndarray of int : A #tris by 3 array, where each row contains indices of vertices in the *vertices* array that are part of the triangle.

**property trimesh**
Convert to trimesh.

**update_tf**(*delta_T*)
Updates the mesh transformation.

**property vertices**
numpy.ndarray of float : A #verts by 3 array, where each row contains an ordered [x,y,z] set that describes one vertex.

**visualize**(*color=0.5, 0.5, 0.5*, *style='surface'*, *opacity=1.0*)
Plots visualization of mesh using MayaVI.

**color** [tuple of float] 3-tuple of floats in [0,1] to give the mesh's color

**style** [str] Either 'surface', which produces an opaque surface, or 'wireframe', which produces a wireframe.

**opacity** [float] A value in [0,1] indicating the opacity of the mesh. Zero is transparent, one is opaque.

**mayavi.modules.surface.Surface** The displayed surface.

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**ObjFile**(*filepath*)
Bases: object

A Wavefront .obj file reader and writer.

**filepath** [str] The full path to the .obj file associated with this reader/writer.

**property filepath**
Returns the full path to the .obj file associated with this reader/writer.

**str** The full path to the .obj file associated with this reader/writer.

**read**()
Reads in the .obj file and returns a Mesh3D representation of that mesh.

*Mesh3D* A Mesh3D created from the data in the .obj file.

**write**(*mesh*)
Writes a Mesh3D object out to a .obj file format

**mesh** [*Mesh3D*] The Mesh3D object to write to the .obj file.

Does not support material files or texture coordinates.

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**Sdf**
Bases: object

Abstract class for signed distance fields.

**property center**
Center of grid.

This basically transforms the world frame to grid center.

numpy.ndarray

**center_world**()
Center of grid (basically transforms world frame to grid center)

**property data**
The SDF data.

**numpy.ndarray of float** The 2- or 3-dimensional ndarray that holds the grid of signed distances.

**property dimensions**
SDF dimension information.

**numpy.ndarray of int** The ndarray that contains the dimensions of the sdf.

**property gradients**
Gradients of the SDF.

**list of numpy.ndarray of float** A list of ndarrays of the same dimension as the SDF. The arrays are in axis order and specify the gradients for that axis at each point.

**is_out_of_bounds**(*coords*)
Returns True if coords is an out of bounds access.

**coords** [numpy.ndarray of int] A 2- or 3-dimensional ndarray that indicates the desired coordinates in the grid.

**bool** Are the coordinates in coords out of bounds?

**on_surface**(*coords*)
Determines whether or not a point is on the object surface.

**coords** [numpy.ndarray of int] A 2- or 3-dimensional ndarray that indicates the desired coordinates in the grid.

**tuple of bool, float** Is the point on the object's surface, and what is the signed distance at that point?

**property origin**
The location of the origin in the SDF grid.

**numpy.ndarray of float** The 2- or 3-ndarray that contains the location of the origin of the mesh grid in real space.

**property resolution**
The grid resolution (how wide each grid cell is).

**float** The width of each grid cell.

**abstract surface_points**()
Returns the points on the surface.

**tuple of numpy.ndarray of int, numpy.ndarray of float** The points on the surface and the signed distances at those points.

**abstract transform**(*tf*)
Returns a new SDF transformed by similarity tf.

**abstract transform_pt_grid_to_obj**(*x_grid*, *direction=False*)
Transforms points from grid frame to world frame

**abstract transform_pt_obj_to_grid**(*x_world*, *direction=False*)
Transforms points from world frame to grid frame

**transform_to_world**()
Returns an sdf object with center in the world frame of reference.

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**Sdf3D**(*sdf_data*, *origin*, *resolution*, *use_abs=False*, *T_sdf_world=RigidTransform(rotation=np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]), translation=np.array([0.0, 0.0, 0.0]), from_frame='sdf', to_frame='world')*)

Bases: *graspnetAPI.utils.dexnet.grasping.meshpy.sdf.Sdf*

**curvature**(*coords*, *delta=0.001*)
Returns an approximation to the local SDF curvature (Hessian) at the given coordinate in grid basis.

**coords** [numpy 3-vector] the grid coordinates at which to get the curvature

delta : Returns ——- curvature : 3x3 ndarray of the curvature at the surface points

**static find_zero_crossing_linear**(*x1*, *y1*, *x2*, *y2*)
Find zero crossing using linear approximation

**static find_zero_crossing_quadratic**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *eps=1.0*)
Find zero crossing using quadratic approximation along 1d line

**gradient**(*coords*)
Returns the SDF gradient at the given coordinates, interpolating if necessary

**coords** [numpy.ndarray of int] A 3-dimensional ndarray that indicates the desired coordinates in the grid.

**float** The gradient at the given coords (interpolated).

**IndexError** If the coords vector does not have three entries.

**max_coords_x = [1, 4, 6, 7]**

**max_coords_y = [2, 4, 5, 7]**

**max_coords_z = [3, 5, 6, 7]**

**min_coords_x = [0, 2, 3, 5]**

**min_coords_y = [0, 1, 3, 6]**

**min_coords_z = [0, 1, 2, 4]**

**num_interpolants = 8**

**rescale**(*scale*)
Rescale an SDF by a given scale factor.

**scale** [float] the amount to scale the SDF

***Sdf3D*** new sdf with given scale

**surface_normal**(*coords*, *delta=1.5*)
Returns the sdf surface normal at the given coordinates by computing the tangent plane using SDF interpolation.

**coords** [`numpy.ndarray` of int] A 3-dimensional ndarray that indicates the desired coordinates in the grid.

**delta** [float] A radius for collecting surface points near the target coords for calculating the surface normal.

**`numpy.ndarray` of float** The 3-dimensional ndarray that represents the surface normal.

**IndexError** If the coords vector does not have three entries.

**surface_points**(*grid_basis=True*)
Returns the points on the surface.

**grid_basis** [bool] If False, the surface points are transformed to the world frame. If True (default), the surface points are left in grid coordinates.

**`tuple` of `numpy.ndarray` of int, `numpy.ndarray` of float** The points on the surface and the signed distances at those points.

**transform**(*delta_T*)
Creates a new SDF with a given pose with respect to world coordinates.

**delta_T** [`autolab_core.RigidTransform`] transform from cur sdf to transformed sdf coords

**transform_dense**(*delta_T*, *detailed=False*)
Transform the grid by pose T and scale with canonical reference frame at the SDF center with axis alignment.

**delta_T** [SimilarityTransform] the transformation from the current frame of reference to the new frame of reference

**detailed** [bool] whether or not to use interpolation

***Sdf3D*** new sdf with grid warped by T

**transform_pt_grid_to_obj**(*x_grid*, *direction=False*)
Converts a point in grid coords to the world basis. If direction then don't translate.

**x_grid** [numpy 3xN ndarray or numeric scalar] points to transform from grid basis to sdf basis in meters

direction : bool Returns ——- x_sdf : numpy 3xN ndarray

points in sdf basis (meters)

**transform_pt_obj_to_grid**(*x_sdf*, *direction=False*)
Converts a point in sdf coords to the grid basis. If direction then don't translate.

**x_sdf** [numpy 3xN ndarray or numeric scalar] points to transform from sdf basis in meters to grid basis

direction : bool Returns ——- x_grid : numpy 3xN ndarray or scalar

points in grid basis

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**SdfFile**(*filepath*)
Bases: object

A Signed Distance Field .sdf file reader and writer.

**filepath** [str] The full path to the .sdf or .csv file associated with this reader/writer.

**property filepath**
Returns the full path to the file associated with this reader/writer.

**str** The full path to the file associated with this reader/writer.

**read**()
Reads in the SDF file and returns a Sdf object.

***Sdf*** A Sdf created from the data in the file.

**write**(*sdf*)
Writes an SDF to a file.

**sdf** [*Sdf*] An Sdf object to write out.

This is not currently implemented or supported.

**class** graspnetAPI.utils.dexnet.grasping.meshpy.**StablePose**(*p*, *r*, *x0*, *face=None*, *stp_id=- 1*)
Bases: object

A representation of a mesh's stable pose.

**p** [float] Probability associated with this stable pose.

**r** [numpy.ndarray of :obj`numpy.ndarray` of float] 3x3 rotation matrix that rotates the mesh into the stable pose from standardized coordinates.

**x0** [numpy.ndarray of float] 3D point in the mesh that is resting on the table.

**face** [numpy.ndarray] 3D vector of indices corresponding to vertices forming the resting face

**stp_id** [str] A string identifier for the stable pose

**T_obj_table** [RigidTransform] A RigidTransform representation of the pose's rotation matrix.

**property T_obj_table**

**property T_obj_world**

## Submodules

### graspnetAPI.utils.dexnet.grasping.contacts module

REGENTS SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, PROVIDED HEREUNDER IS PROVIDED "AS IS". REGENTS HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

**class** graspnetAPI.utils.dexnet.grasping.contacts.**Contact**
Bases: object

Abstract class for contact models.

**class** graspnetAPI.utils.dexnet.grasping.contacts.**Contact3D**(*graspable*, *contact_point*, *in_direction=None*)
Bases: *graspnetAPI.utils.dexnet.grasping.contacts.Contact*

3D contact points.

**graspable** [GraspableObject3D] object to use to get contact information

**contact_point** [3x1 numpy.ndarray] point of contact on the object

**in_direction** [3x1 numpy.ndarray] direction along which contact was made

**normal** [normalized 3x1 numpy.ndarray] surface normal at the contact point

**friction_cone**(*num_cone_faces=8*, *friction_coef=0.5*)
Computes the friction cone and normal for a contact point.

**num_cone_faces** [int] number of cone faces to use in discretization

**friction_coef** [float ] coefficient of friction at contact point

**success** [bool] False when cone can't be computed

**cone_support** [numpy.ndarray] array where each column is a vector on the boundary of the cone

**normal** [normalized 3x1 numpy.ndarray] outward facing surface normal

**property graspable**

**property in_direction**

**property normal**

**normal_force_magnitude**()
Returns the component of the force that the contact would apply along the normal direction.

**float** magnitude of force along object surface normal

**plot_friction_cone**(*color='y'*, *scale=1.0*)

**property point**

**reference_frame**(*align_axes=True*)
Returns the local reference frame of the contact. Z axis in the in direction (or surface normal if not specified) X and Y axes in the tangent plane to the direction

**align_axes** [bool] whether or not to align to the object axes

**RigidTransform** rigid transformation from contact frame to object frame

**surface_information**(*width*, *num_steps*, *sigma_range=0.1*, *sigma_spatial=1*, *back_up=0.0*, *max_projection=0.1*, *direction=None*, *debug_objs=None*, *samples_per_grid=2*)

Returns the local surface window, gradient, and curvature for a single contact.

**width** [float] width of surface window in object frame

**num_steps** [int ] number of steps to use along the in direction

**sigma_range** [float] bandwidth of bilateral range filter on window

**sigma_spatial** [float] bandwidth of gaussian spatial filter of bilateral filter

**back_up** [float] amount to back up before finding a contact in meters

**max_projection** [float] maximum amount to search forward for a contact (meters)

**direction** [3x1 `numpy.ndarray`] direction along width to render the window

**debug_objs** [`list`] list to put debugging info into

**samples_per_grid** [float] number of samples per grid when finding contacts

**surface_window** [*SurfaceWindow*] window information for local surface patch of contact on the given object

**surface_window_projection**(*width=0.01*, *num_steps=21*, *max_projection=0.1*, *back_up=0.0*, *samples_per_grid=2.0*, *sigma_range=0.1*, *sigma_spatial=1*, *direction=None*, *compute_pca=False*, *vis=False*, *debug_objs=None*)

Projects the local surface onto the tangent plane at a contact point.

**width** [float] width of the window in obj frame

**num_steps** [int ] number of steps to use along the in direction

**max_projection** [float] maximum amount to search forward for a contact (meters)

**back_up** [float] amount to back up before finding a contact in meters

**samples_per_grid** [float] number of samples per grid when finding contacts

**sigma_range** [float] bandwidth of bilateral range filter on window

**sigma_spatial** [float] bandwidth of gaussian spatial filter of bilateral filter

**direction** [3x1 `numpy.ndarray`] dir to do the projection along

**window** [NUM_STEPSxNUM_STEPS `numpy.ndarray`] array of distances from tangent plane to obj, False if surface window can't be computed

**surface_window_projection_unaligned**(*width=0.01*, *num_steps=21*, *max_projection=0.1*, *back_up=0.0*, *samples_per_grid=2.0*, *sigma=1.5*, *direction=None*, *vis=False*)

Projects the local surface onto the tangent plane at a contact point. Deprecated.

**surface_window_sdf**(*width=0.01*, *num_steps=21*)

Returns a window of SDF values on the tangent plane at a contact point. Used for patch computation.

**width** [float] width of the window in obj frame

**num_steps** [int] number of steps to use along the contact in direction

**window** [NUM_STEPSxNUM_STEPS numpy.ndarray] array of distances from tangent plane to obj along in direction, False if surface window can't be computed

**tangents**(*direction=None*, *align_axes=True*, *max_samples=1000*)

Returns the direction vector and tangent vectors at a contact point. The direction vector defaults to the *inward-facing* normal vector at this contact. The direction and tangent vectors for a right handed coordinate frame.

**direction** [3x1 numpy.ndarray] direction to find orthogonal plane for

**align_axes** [bool] whether or not to align the tangent plane to the object reference frame

**max_samples** [int] number of samples to use in discrete optimization for alignment of reference frame

**direction** [normalized 3x1 numpy.ndarray] direction to find orthogonal plane for

**t1** [normalized 3x1 numpy.ndarray] first tangent vector, x axis

**t2** [normalized 3x1 numpy.ndarray] second tangent vector, y axis

**torques**(*forces*)

Get the torques that can be applied by a set of force vectors at the contact point.

**forces** [3xN numpy.ndarray] the forces applied at the contact

**success** [bool] whether or not computation was successful

**torques** [3xN numpy.ndarray] the torques that can be applied by given forces at the contact

**class** graspnetAPI.utils.dexnet.grasping.contacts.**SurfaceWindow**(*proj_win*, *grad*, *hess_x*, *hess_y*, *gauss_curvature*)

Bases: object

Struct for encapsulating local surface window features.

**proj_win** [NxN numpy.ndarray] the window of distances to a surface (depth image created by orthographic projection)

**grad** [NxN numpy.ndarray] X and Y gradients of the projection window

**hess_x** [NxN numpy.ndarray] hessian, partial derivatives of the X gradient window

**hess_y** [NxN numpy.ndarray] hessian, partial derivatives of the Y gradient window

**gauss_curvature** [NxN numpy.ndarray] gauss curvature at each point (function of hessian determinant)

**asarray**(*proj_win_weight=0.0*, *grad_x_weight=0.0*, *grad_y_weight=0.0*, *curvature_weight=0.0*)

**property curvature**

**property grad_x**

**property grad_x_2d**

**property grad_y**

**property grad_y_2d**

**property proj_win**

**property proj_win_2d**

### graspnetAPI.utils.dexnet.grasping.grasp module

**class** graspnetAPI.utils.dexnet.grasping.grasp.**Grasp**
Bases: object

Abstract grasp class.

**configuration** [numpy.ndarray] vector specifying the parameters of the grasp (e.g. hand pose, opening width, joint angles, etc)

**frame** [str] string name of grasp reference frame (defaults to obj)

**abstract close_fingers**(*obj*)
Finds the contact points by closing on the given object.

**obj** [GraspableObject3D] object to find contacts on

**abstract configuration**()
Returns the numpy array representing the hand configuration

**abstract static configuration_from_params**(**params*)
Convert param list to a configuration vector for the class

**abstract frame**()
Returns the string name of the grasp reference frame

**abstract static params_from_configuration**(*configuration*)
Convert configuration vector to a set of params for the class

**samples_per_grid = 2**

**class** graspnetAPI.utils.dexnet.grasping.grasp.**ParallelJawPtGrasp3D**(*configuration*, *max_grasp_depth=0*, *frame='object'*, *grasp_id=None*)
Bases: *graspnetAPI.utils.dexnet.grasping.grasp.PointGrasp*

Parallel Jaw point grasps in 3D space.

**property T_grasp_obj**
Rigid transformation from grasp frame to object frame. Rotation matrix is X-axis along approach direction, Y axis pointing between the jaws, and Z-axis orthogonal. Translation vector is the grasp center.

**RigidTransform** transformation from grasp to object coordinates

**property approach_angle**
float : approach angle of the grasp

**property axis**
numpy.ndarray : normalized 3-vector specifying the line between the jaws

**static axis_from_endpoints**(*g1*, *g2*)
Normalized axis of grasp from endpoints as np 3-arrays

**property center**
numpy.ndarray : 3-vector specifying the center of the jaws

**static center_from_endpoints**(*g1*, *g2*)
Grasp center from endpoints as np 3-arrays

**close_fingers**(*obj*, *vis=False*, *check_approach=True*, *approach_dist=1.0*)
Steps along grasp axis to find the locations of contact with an object

**obj** [GraspableObject3D] object to close fingers on

**vis** [bool] whether or not to plot the line of action and contact points

**check_approach** [bool] whether or not to check if the contact points can be reached

**approach_dist** [float] how far back to check the approach distance, only if checking the approach is set

**success** [bool] whether or not contacts were found

**c1** [`Contact3D`] the contact point for jaw 1

**c2** [`Contact3D`] the contact point for jaw 2

**close_fingers_with_contacts**(*obj*, *contacts*, *vis=False*, *check_approach=True*, *approach_dist=0.5*)

Steps along grasp axis to find the locations of contact with an object

**obj** [`GraspableObject3D`] object to close fingers on

**vis** [bool] whether or not to plot the line of action and contact points

**check_approach** [bool] whether or not to check if the contact points can be reached

**approach_dist** [float] how far back to check the approach distance, only if checking the approach is set

**success** [bool] whether or not contacts were found

**c1** [`Contact3D`] the contact point for jaw 1

**c2** [`Contact3D`] the contact point for jaw 2

**property close_width**

float : minimum opening width of the jaws

**property configuration**

`numpy.ndarray` : vector specifying the parameters of the grasp as follows (grasp_center, grasp_axis, grasp_angle, grasp_width, jaw_width)

**static configuration_from_params**(*center*, *axis*, *width*, *angle=0*, *jaw_width=0*, *min_width=0*)

Converts grasp parameters to a configuration vector.

**static create_line_of_action**(*g*, *axis*, *width*, *obj*, *num_samples*, *min_width=0*, *convert_grid=True*)

Creates a straight line of action, or list of grid points, from a given point and direction in world or grid coords

**g** [3x1 `numpy.ndarray`] start point to create the line of action

**axis** [normalized 3x1 `numpy.ndarray`] normalized numpy 3 array of grasp direction

**width** [float] the grasp width

**num_samples** [int] number of discrete points along the line of action

**convert_grid** [bool] whether or not the points are specified in world coords

**line_of_action** [`list` of 3x1 `numpy.ndarrays`] coordinates to pass through in 3D space for contact checking

**static distance**(*g1*, *g2*, *alpha=0.05*)

Evaluates the distance between two grasps.

**g1** [*ParallelJawPtGrasp3D*] the first grasp to use

**g2** [*ParallelJawPtGrasp3D*] the second grasp to use

**alpha** [float] parameter weighting rotational versus spatial distance

**float** distance between grasps g1 and g2

**property endpoints**

**numpy.ndarray** location of jaws in 3D space at max opening width

**static find_contact**(*line_of_action*, *obj*, *vis=False*, *strict=False*)
Find the point at which a point traveling along a given line of action hits a surface.

**line_of_action** [`list` of 3x1 `numpy.ndarray`] the points visited as the fingers close (grid coords)

**obj** [`GraspableObject3D`] to check contacts on

**vis** [bool] whether or not to display the contact check (for debugging)

**contact_found** [bool] whether or not the point contacts the object surface

**contact** [`Contact3D`] found along line of action (None if contact not found)

**property frame**
`str` : name of grasp reference frame

**grasp_angles_from_stp_z**(*stable_pose*)
Get angles of the the grasp from the table plane: 1) the angle between the grasp axis and table normal 2) the angle between the grasp approach axis and the table normal

**stable_pose** [`StablePose` or `RigidTransform`] the stable pose to compute the angles for

**psi** [float] grasp y axis rotation from z axis in stable pose

**phi** [float] grasp x axis rotation from z axis in stable pose

**static grasp_from_contact_and_axis_on_grid**(*obj*, *grasp_c1_world*, *grasp_axis_world*, *grasp_width_world*, *grasp_angle=0*, *jaw_width_world=0*, *min_grasp_width_world=0*, *vis=False*, *backup=0.5*)
Creates a grasp from a single contact point in grid coordinates and direction in grid coordinates.

**obj** [`GraspableObject3D`] object to create grasp for

**grasp_c1_grid** [3x1 `numpy.ndarray`] contact point 1 in world

**grasp_axis** [normalized 3x1 `numpy.ndarray`] normalized direction of the grasp in world

**grasp_width_world** [float] grasp_width in world coords

**jaw_width_world** [float] width of jaws in world coords

**min_grasp_width_world** [float] min closing width of jaws

**vis** [bool] whether or not to visualize the grasp

**g** [`ParallelJawGrasp3D`] grasp created by finding the second contact

**c1** [`Contact3D`] first contact point on the object

**c2** [`Contact3D`] second contact point on the object

**static grasp_from_endpoints**(*g1*, *g2*, *width=None*, *approach_angle=0*, *close_width=0*)
Create a grasp from given endpoints in 3D space, making the axis the line between the points.

**g1** [numpy.ndarray] location of the first jaw

**g2** [numpy.ndarray] location of the second jaw

**width** [float] maximum opening width of jaws

**approach_angle** [float] approach angle of grasp

**close_width** [float] width of gripper when fully closed

**grasp_y_axis_offset**(*theta*)
Return a new grasp with the given approach angle.

**theta** [float] approach angle for the new grasp

***ParallelJawPtGrasp3D*** grasp with the given approach angle

**gripper_pose**(*gripper=None*)
Returns the RigidTransformation from the gripper frame to the object frame when the gripper is executing the given grasp. Differs from the grasp reference frame because different robots use different conventions for the gripper reference frame.

**gripper** [RobotGripper] gripper to get the pose for

**RigidTransform** transformation from gripper frame to object frame

**property id**
int : id of grasp

**property jaw_width**
float : width of the jaws in the tangent plane to the grasp axis

**property open_width**
float : maximum opening width of the jaws

**parallel_table**(*stable_pose*)
Returns a grasp with approach_angle set to be perpendicular to the table normal specified in the given stable pose.

**stable_pose** [StablePose] the pose specifying the table

***ParallelJawPtGrasp3D*** aligned grasp

**static params_from_configuration**(*configuration*)
Converts configuration vector into grasp parameters.

**grasp_center** [numpy.ndarray] center of grasp in 3D space

**grasp_axis** [numpy.ndarray] normalized axis of grasp in 3D space

**max_width** [float] maximum opening width of jaws

**angle** [float] approach angle

**jaw_width** [float] width of jaws

**min_width** [float] minimum closing width of jaws

**perpendicular_table**(*stable_pose*)
Returns a grasp with approach_angle set to be aligned width the table normal specified in the given stable pose.

**stable_pose** [StablePose or RigidTransform] the pose specifying the orientation of the table

***ParallelJawPtGrasp3D*** aligned grasp

**project_camera**(*T_obj_camera*, *camera_intr*)
Project a grasp for a given gripper into the camera specified by a set of intrinsics.

**T_obj_camera** [autolab_core.RigidTransform] rigid transformation from the object frame to the camera frame

**camera_intr** [perception.CameraIntrinsics] intrinsics of the camera to use

**property rotated_full_axis**
Rotation matrix from canonical grasp reference frame to object reference frame. X axis points out of the gripper palm along the grasp approach angle, Y axis points between the jaws, and the Z axs is orthogonal.

**numpy.ndarray** rotation matrix of grasp

**surface_information**(*graspable*, *width=0.02*, *num_steps=21*, *direction=None*)
Return the patch surface information at the contacts that this grasp makes on a graspable.

**graspable** [GraspableObject3D] object to get surface information for

**width** [float] width of the window in obj frame

**num_steps** [int] number of steps

**list of SurfaceWindow** surface patches, one for each contact

**property unrotated_full_axis**
Rotation matrix from canonical grasp reference frame to object reference frame. X axis points out of the gripper palm along the 0-degree approach direction, Y axis points between the jaws, and the Z axs is orthogonal.

**numpy.ndarray** rotation matrix of grasp

**vis_grasp**(*obj*, **args*, ***kwargs*)

**static width_from_endpoints**(*g1*, *g2*)
Width of grasp from endpoints

**class** graspnetAPI.utils.dexnet.grasping.grasp.**PointGrasp**
Bases: *graspnetAPI.utils.dexnet.grasping.grasp.Grasp*

Abstract grasp class for grasps with a point contact model.

**configuration** [numpy.ndarray] vector specifying the parameters of the grasp (e.g. hand pose, opening width, joint angles, etc)

**frame** [str] string name of grasp reference frame (defaults to obj)

**abstract create_line_of_action**(*axis*, *width*, *obj*, *num_samples*)
Creates a line of action, or the points in space that the grasp traces out, from a point g in world coordinates on an object.

**bool** whether or not successful

**list of numpy.ndarray** points in 3D space along the line of action

**class** graspnetAPI.utils.dexnet.grasping.grasp.**VacuumPoint**(*configuration*, *frame='object'*, *grasp_id=None*)

Bases: *graspnetAPI.utils.dexnet.grasping.grasp.Grasp*

Defines a vacuum target point and axis in 3D space (5 DOF)

**property axis**

**property center**

**property configuration**

Returns the numpy array representing the hand configuration

**static configuration_from_params**(*center*, *axis*)

Converts grasp parameters to a configuration vector.

**property frame**

Returns the string name of the grasp reference frame

**static params_from_configuration**(*configuration*)

Converts configuration vector into vacuum grasp parameters.

**center** [numpy.ndarray] center of grasp in 3D space

**axis** [numpy.ndarray] normalized axis of grasp in 3D space

### graspnetAPI.utils.dexnet.grasping.grasp_quality_config module



**class** graspnetAPI.utils.dexnet.grasping.grasp_quality_config.**GraspQualityConfig**(*config*)

Bases: object

Base wrapper class for parameters used in grasp quality computation. Used to elegantly enforce existence and type of required parameters.

**config** [dict] dictionary mapping parameter names to parameter values

**abstract check_valid**(*config*)

Raise an exception if the config is missing required keys

**contains**(*key*)

Checks whether or not the key is supported

**keys**()

**class** graspnetAPI.utils.dexnet.grasping.grasp_quality_config.**GraspQualityConfigFactory**
Bases: object

Helper class to automatically create grasp quality configurations of different types.

**static create_config**(*config*)
Automatically create a quality config from a dictionary.

**config** [dict] dictionary mapping parameter names to parameter values

**class** graspnetAPI.utils.dexnet.grasping.grasp_quality_config.**QuasiStaticGraspQualityConfig**
Bases: *graspnetAPI.utils.dexnet.grasping.grasp_quality_config.GraspQualityConfig*

Parameters for quasi-static grasp quality computation.

**config** [dict] dictionary mapping parameter names to parameter values

Required configuration key-value pairs in Other Parameters.

**quality_method** [str] string name of quasi-static quality metric

**friction_coef** [float] coefficient of friction at contact point

**num_cone_faces** [int] number of faces to use in friction cone approximation

**soft_fingers** [bool] whether to use a soft finger model

**quality_type** [str] string name of grasp quality type (e.g. quasi-static, robust quasi-static)

**check_approach** [bool] whether or not to check the approach direction

**REQUIRED_KEYS = ['quality_method', 'friction_coef', 'num_cone_faces', 'soft_fingers',**

**check_valid**(*config*)
Raise an exception if the config is missing required keys

**class** graspnetAPI.utils.dexnet.grasping.grasp_quality_config.**RobustQuasiStaticGraspQuality**
Bases: *graspnetAPI.utils.dexnet.grasping.grasp_quality_config.GraspQualityConfig*

Parameters for quasi-static grasp quality computation.

**config** [dict] dictionary mapping parameter names to parameter values

Required configuration key-value pairs in Other Parameters.

**quality_method** [str] string name of quasi-static quality metric

**friction_coef** [float] coefficient of friction at contact point

**num_cone_faces** [int] number of faces to use in friction cone approximation

**soft_fingers** [bool] whether to use a soft finger model

**quality_type** [str] string name of grasp quality type (e.g. quasi-static, robust quasi-static)

**check_approach** [bool] whether or not to check the approach direction

**num_quality_samples** [int] number of samples to use

**ROBUST_REQUIRED_KEYS = ['num_quality_samples']**

**check_valid**(*config*)
Raise an exception if the config is missing required keys

### graspnetAPI.utils.dexnet.grasping.grasp_quality_function module

### graspnetAPI.utils.dexnet.grasping.graspable_object module



**class** graspnetAPI.utils.dexnet.grasping.graspable_object.**GraspableObject**(*sdf*, *mesh*, *key=''*, *model_name=''*, *mass=1.0*, *convex_pieces=None*)

Bases: object

Encapsulates geometric structures for computing contact in grasping.

**sdf** [Sdf3D] signed distance field, for quickly computing contact points

**mesh** [Mesh3D] 3D triangular mesh to specify object geometry, should match SDF

**key** [str] object identifier, usually given from the database

**model_name** [str] name of the object mesh as a .obj file, for use in collision checking

**mass** [float] mass of the object

**convex_pieces** [list of Mesh3D] convex decomposition of the object geom for collision checking

**property convex_pieces**

**property key**

**property mass**

**property mesh**

**property model_name**

**property sdf**

**class** graspnetAPI.utils.dexnet.grasping.graspable_object.**GraspableObject3D**(*sdf*, *mesh*, *key=''*, *model_name=''*, *mass=1.0*, *convex_pieces=None*)

Bases: *graspnetAPI.utils.dexnet.grasping.graspable_object.GraspableObject*

3D Graspable object for computing contact in grasping.

**sdf** [Sdf3D] signed distance field, for quickly computing contact points

**mesh** [Mesh3D] 3D triangular mesh to specify object geometry, should match SDF

**key** [str] object identifier, usually given from the database

**model_name** [str] name of the object mesh as a .obj file, for use in collision checking

**mass** [float] mass of the object

**convex_pieces** [list of Mesh3D] convex decomposition of the object geom for collision checking

**moment_arm**(*x*)

Computes the moment arm to a point x.

**x** [3x1 numpy.ndarray] point to get moment arm for

3x1 numpy.ndarray

**rescale**(*scale*)

Rescales uniformly by a given factor.

**scale** [float] the amount to scale the object

***GraspableObject3D*** the graspable object rescaled by the given factor

**surface_information**(*grasp*, *width*, *num_steps*, *plot=False*, *direction1=None*, *direction2=None*)

Returns the patches on this object for a given grasp.

**grasp** [ParallelJawPtGrasp3D] grasp to get the patch information for

**width** [float] width of jaw opening

**num_steps** [int] number of steps

**plot** [bool] whether to plot the intermediate computation, for debugging

**direction1** [normalized 3x1 numpy.ndarray] direction along which to compute the surface information for the first jaw, if None then defaults to grasp axis

**direction2** [normalized 3x1 numpy.ndarray] direction along which to compute the surface information for the second jaw, if None then defaults to grasp axis

**list of SurfaceWindow** surface patches, one for each contact

**transform**(*delta_T*)

Transform by a delta transform.

**delta_T** [RigidTransform] the transformation from the current reference frame to the alternate reference frame

***GraspableObject3D*** graspable object trasnformed by the delta

### graspnetAPI.utils.dexnet.grasping.quality module

**class** graspnetAPI.utils.dexnet.grasping.quality.**PointGraspMetrics3D**
Bases: object

Class to wrap functions for quasistatic point grasp quality metrics.

**static ferrari_canny_L1**(*forces*, *torques*, *normals*, *soft_fingers=False*, *params=None*, *wrench_norm_thresh=0.001*, *wrench_regularizer=1e-10*)
Ferrari & Canny's L1 metric. Also known as the epsilon metric.

**forces** [3xN numpy.ndarray] set of forces on object in object basis

**torques** [3xN numpy.ndarray] set of torques on object in object basis

**normals** [3xN numpy.ndarray] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**params** [GraspQualityConfig] set of parameters for grasp matrix and contact model

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

float : value of metric

**static ferrari_canny_L1_force_only**(*forces*, *torques*, *normals*, *soft_fingers=False*, *params=None*, *wrench_norm_thresh=0.001*, *wrench_regularizer=1e-10*)
Ferrari & Canny's L1 metric with force only. Also known as the epsilon metric.

**forces** [3xN numpy.ndarray] set of forces on object in object basis

**torques** [3xN numpy.ndarray] set of torques on object in object basis

**normals** [3xN numpy.ndarray] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**params** [GraspQualityConfig] set of parameters for grasp matrix and contact model

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

float : value of metric

**static force_closure**(*c1*, *c2*, *friction_coef*, *use_abs_value=True*)
" Checks force closure using the antipodality trick.

**c1** [Contact3D] first contact point

**c2** [Contact3D] second contact point

**friction_coef** [float] coefficient of friction at the contact point

**use_abs_value** [bool] whether or not to use directoinality of the surface normal (useful when mesh is not oriented)

int : 1 if in force closure, 0 otherwise

**static force_closure_qp**(*forces*, *torques*, *normals*, *soft_fingers=False*, *wrench_norm_thresh=0.001*, *wrench_regularizer=1e-10*, *params=None*)
Checks force closure by solving a quadratic program (whether or not zero is in the convex hull)

**forces** [3xN numpy.ndarray] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

int : 1 if in force closure, 0 otherwise

**static grasp_isotropy**(*forces*, *torques*, *normals*, *soft_fingers=False*, *params=None*)

Condition number of grasp matrix - ratio of "weakest" wrench that the grasp can exert to the "strongest" one.

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

float : value of grasp isotropy metric

**static grasp_matrix**(*forces*, *torques*, *normals*, *soft_fingers=False*, *finger_radius=0.005*, *params=None*)

Computes the grasp map between contact forces and wrenchs on the object in its reference frame.

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**finger_radius** [float] the radius of the fingers to use

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

**G** [6xM `numpy.ndarray`] grasp map

**static grasp_quality**(*grasp*, *obj*, *params*, *contacts=None*, *vis=False*)

Computes the quality of a two-finger point grasps on a given object using a quasi-static model.

**grasp** [`ParallelJawPtGrasp3D`] grasp to evaluate

**obj** [`GraspableObject3D`] object to evaluate quality on

**params** [`GraspQualityConfig`] parameters of grasp quality function

**static min_norm_vector_in_facet**(*facet*, *wrench_regularizer=1e-10*)

Finds the minimum norm point in the convex hull of a given facet (aka simplex) by solving a QP.

**facet** [6xN `numpy.ndarray`] vectors forming the facet

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

**float** minimum norm of any point in the convex hull of the facet

**Nx1 `numpy.ndarray`** vector of coefficients that achieves the minimum

**static min_singular**(*forces*, *torques*, *normals*, *soft_fingers=False*, *params=None*)
Min singular value of grasp matrix - measure of wrench that grasp is "weakest" at resisting.

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

float : value of smallest singular value

**static partial_closure**(*forces*, *torques*, *normals*, *soft_fingers=False*, *wrench_norm_thresh=0.001*, *wrench_regularizer=1e-10*, *params=None*)
Evalutes partial closure: whether or not the forces and torques can resist a specific wrench. Estimates resistance by sollving a quadratic program (whether or not the target wrench is in the convex hull).

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

int : 1 if in partial closure, 0 otherwise

**static wrench_in_positive_span**(*wrench_basis*, *target_wrench*, *force_limit*, *num_fingers=1*, *wrench_norm_thresh=0.0001*, *wrench_regularizer=1e-10*)
Check whether a target can be exerted by positive combinations of wrenches in a given basis with L1 norm fonger force limit limit.

**wrench_basis** [6xN `numpy.ndarray`] basis for the wrench space

**target_wrench** [6x1 `numpy.ndarray`] target wrench to resist

**force_limit** [float] L1 upper bound on the forces per finger (aka contact point)

**num_fingers** [int] number of contacts, used to enforce L1 finger constraint

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

**int** whether or not wrench can be resisted

**float** minimum norm of the finger forces required to resist the wrench

**static wrench_resistance**(*forces*, *torques*, *normals*, *soft_fingers=False*, *wrench_norm_thresh=0.001*, *wrench_regularizer=1e-10*, *finger_force_eps=1e-09*, *params=None*)
Evalutes wrench resistance: the inverse norm of the contact forces required to resist a target wrench Estimates resistance by sollving a quadratic program (min normal contact forces to produce a wrench).

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**wrench_norm_thresh** [float] threshold to use to determine equivalence of target wrenches

**wrench_regularizer** [float] small float to make quadratic program positive semidefinite

**finger_force_eps** [float] small float to prevent numeric issues in wrench resistance metric

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

float : value of wrench resistance metric

**static wrench_volume**(*forces*, *torques*, *normals*, *soft_fingers=False*, *params=None*)
Volume of grasp matrix singular values - score of all wrenches that the grasp can resist.

**forces** [3xN `numpy.ndarray`] set of forces on object in object basis

**torques** [3xN `numpy.ndarray`] set of torques on object in object basis

**normals** [3xN `numpy.ndarray`] surface normals at the contact points

**soft_fingers** [bool] whether or not to use the soft finger contact model

**params** [`GraspQualityConfig`] set of parameters for grasp matrix and contact model

float : value of wrench volume

### Module contents

### Submodules

#### graspnetAPI.utils.dexnet.abstractstatic module



**class** graspnetAPI.utils.dexnet.abstractstatic.**abstractstatic**(*function*)
Bases: staticmethod

#### graspnetAPI.utils.dexnet.constants module

### Module contents

### Submodules

### graspnetAPI.utils.config module

graspnetAPI.utils.config.**get_config**()

- return the config dict

### graspnetAPI.utils.eval_utils module

graspnetAPI.utils.eval_utils.**collision_detection**(*grasp_list*, *model_list*, *dexnet_models*, *poses*, *scene_points*, *outlier=0.05*, *empty_thresh=10*, *return_dexgrasps=False*)

**Input:**

- grasp_list: [(k1, 17), (k2, 17), ..., (kn, 17)] in camera coordinate
- model_list: [(N1, 3), (N2, 3), ..., (Nn, 3)] in camera coordinate
- dexnet_models: [GraspableObject3D,] in object coordinate
- poses: [(4, 4),] from model coordinate to camera coordinate
- scene_points: (Ns, 3) in camera coordinate
- outlier: float, used to compute workspace mask
- empty_thresh: int, 'num_inner_points < empty_thresh' means empty grasp
- return_dexgrasps: bool, return grasps in dex-net format while True

**Output:**

- collsion_mask_list: [(k1,), (k2,), ..., (kn,)]
- empty_mask_list: [(k1,), (k2,), ..., (kn,)]
- dexgrasp_list: [[ParallelJawPtGrasp3D,],] in object coordinate

graspnetAPI.utils.eval_utils.**compute_closest_points**(*A*, *B*)

**Input:**

- A: (N, 3)
- B: (M, 3)

**Output:**

- indices: (N,) closest point index in B for each point in A

graspnetAPI.utils.eval_utils.**compute_point_distance**(*A*, *B*)

**Input:** - A: (N, 3)

- B: (M, 3)

**Output:** - dists: (N, M)

graspnetAPI.utils.eval_utils.**create_table_points**(*lx*, *ly*, *lz*, *dx=0*, *dy=0*, *dz=0*, *grid_size=0.01*)

**Input:** - lx: - ly: - lz: **Output:** - numpy array of the points with shape (-1, 3).

graspnetAPI.utils.eval_utils.**eval_grasp**(*grasp_group*, *models*, *dexnet_models*, *poses*, *config*, *table=None*, *voxel_size=0.008*, *TOP_K=50*)

**Input:**

- grasp_group: GraspGroup instance for evaluation.
- models: in model coordinate
- dexnet_models: models in dexnet format
- poses: from model to camera coordinate
- config: dexnet config.
- table: in camera coordinate
- voxel_size: float of the voxel size.
- TOP_K: int of the number of top grasps to evaluate.

graspnetAPI.utils.eval_utils.**get_grasp_score**(*grasp*, *obj*, *fc_list*, *force_closure_quality_config*)

graspnetAPI.utils.eval_utils.**get_scene_name**(*num*)

**Input:** - num: int of the scene number.

**Output:** - string of the scene name.

graspnetAPI.utils.eval_utils.**load_dexnet_model**(*data_path*)

**Input:**

- data_path: path to load .obj & .sdf files

**Output:** - obj: dexnet model

graspnetAPI.utils.eval_utils.**parse_posevector**(*posevector*)

**Input:** - posevector: list of pose **Output:** - obj_idx: int of the index of object. - mat: numpy array of shape (4, 4) of the 6D pose of object.

graspnetAPI.utils.eval_utils.**topk_grasps**(*grasps*, *k=10*)

**Input:**

- grasps: (N, 17)
- k: int

**Output:**

- topk_grasps: (k, 17)

graspnetAPI.utils.eval_utils.**transform_points**(*points*, *trans*)

**Input:**

- points: (N, 3)
- trans: (4, 4)

**Output:** - points_trans: (N, 3)

graspnetAPI.utils.eval_utils.**voxel_sample_points**(*points*, *voxel_size=0.008*)

**Input:**

- points: (N, 3)

**Output:**

- points: (n, 3)

### graspnetAPI.utils.pose module

**class** graspnetAPI.utils.pose.**Pose**(*id*, *x*, *y*, *z*, *alpha*, *beta*, *gamma*)
Bases: object

**get_id**()
**Output:**
- return the id of this object

**get_mat_4x4**()
**Output:**
- Convert self.x, self.y, self.z, self.alpha, self.beta and self.gamma into mat_4x4 pose

**get_quat**()
**Output:**
- Convert self.alpha, self.beta, self.gamma into self.quat

**get_translation**()
**Output:**
- Convert self.x, self.y, self.z into self.translation

graspnetAPI.utils.pose.**pose_from_pose_vector**(*pose_vector*)
**Input:**
- pose_vector: A list in the format of [id,x,y,z,alpha,beta,gamma]

**Output:**
- A pose class instance

graspnetAPI.utils.pose.**pose_list_from_pose_vector_list**(*pose_vector_list*)
**Input:**
- Pose vector list defined in xmlhandler.py

**Output:**
- list of poses.

### graspnetAPI.utils.rotation module

Author: chenxi-wang Transformation matrices from/to viewpoints and dexnet gripper params.

graspnetAPI.utils.rotation.**batch_viewpoint_params_to_matrix**(*batch_towards*, *batch_angle*)
**Input:**
- towards: numpy array towards vectors with shape (n, 3).
- angle: numpy array of in-plane rotations (n, ).

**Output:**
- numpy array of the rotation matrix with shape (n, 3, 3).

graspnetAPI.utils.rotation.**dexnet_params_to_matrix**(*binormal*, *angle*)
**Input:**
- binormal: numpy array of shape (3,).
- angle: float of the angle.

**Output:**

- numpy array of shape (3, 3) of the rotation matrix.

graspnetAPI.utils.rotation.**matrix_to_dexnet_params**(*matrix*)

**Input:**

- numpy array of shape (3, 3) of the rotation matrix.

**Output:**

- binormal: numpy array of shape (3,).
- angle: float of the angle.

graspnetAPI.utils.rotation.**rotation_matrix**(*alpha*, *beta*, *gamma*)

**Input:**

- alpha: float of alpha angle.
- beta: float of beta angle.
- gamma: float of the gamma angle.

**Output:**

- numpy array of shape (3, 3) of rotation matrix.

graspnetAPI.utils.rotation.**viewpoint_params_to_matrix**(*towards*, *angle*)

**Input:**

- towards: numpy array towards vector with shape (3,).
- angle: float of in-plane rotation.

**Output:**

- numpy array of the rotation matrix with shape (3, 3).

### graspnetAPI.utils.trans3d module

graspnetAPI.utils.trans3d.**get_mat**(*x*, *y*, *z*, *alpha*, *beta*, *gamma*)

Calls get_mat() to get the 4x4 matrix

graspnetAPI.utils.trans3d.**get_pose**(*pose*)

graspnetAPI.utils.trans3d.**pos_quat_to_pose_4x4**(*pos*, *quat*)

pose = pos_quat_to_pose_4x4(pos, quat) Convert pos and quat into pose, 4x4 format

**Args:** pos: length-3 position quat: length-4 quaternion

**Returns:** pose: numpy array, 4x4

graspnetAPI.utils.trans3d.**pose_4x4_to_pos_quat**(*pose*)

Convert pose, 4x4 format into pos and quat

**Args:** pose: numpy array, 4x4

**Returns:**

pos: length-3 position

quat: length-4 quaternion

### graspnetAPI.utils.utils module

**class** graspnetAPI.utils.utils.**CameraInfo**(*width*, *height*, *fx*, *fy*, *cx*, *cy*, *scale*)
Bases: object

Author: chenxi-wang Camera intrinsics for point cloud generation.

graspnetAPI.utils.utils.**batch_center_depth**(*depths*, *centers*, *open_points*, *upper_points*)
**Input:**

- depths: numpy array of the depths.
- centers: numpy array of the center points of shape(-1, 2).
- open_points: numpy array of the open points of shape(-1, 2).
- upper_points: numpy array of the upper points of shape(-1, 2).

**Output:**

- depths: numpy array of the grasp depth of shape (-1).

graspnetAPI.utils.utils.**batch_framexy_depth_2_xyz**(*pixel_x*, *pixel_y*, *depth*, *camera*)
**Input:**

- pixel_x: numpy array of int of the pixel x coordinate. shape: (-1,)
- pixel_y: numpy array of int of the pixle y coordicate. shape: (-1,)
- depth: numpy array of float of depth. The unit is millimeter. shape: (-1,)
- camera: string of type of camera. “realsense” or “kinect”.

**Output:**

x, y, z: numpy array of float of x, y and z coordinates in camera frame. The unit is millimeter.

graspnetAPI.utils.utils.**batch_key_point_2_rotation**(*centers_xyz*, *open_points_xyz*, *upper_points_xyz*)
**Input:**

- centers_xyz: numpy array of the center points of shape (-1, 3).
- open_points_xyz: numpy array of the open points of shape (-1, 3).
- upper_points_xyz: numpy array of the upper points of shape (-1, 3).

**Output:**

- rotations: numpy array of the rotation matrix of shape (-1, 3, 3).

graspnetAPI.utils.utils.**batch_key_points_2_tuple**(*key_points*, *scores*, *object_ids*, *camera*)
**Input:**

- key_points: np.array(-1,4,3) of grasp key points, definition is shown in key_points.png
- scores: numpy array of batch grasp scores.
- camera: string of ‘realsense’ or ‘kinect’.

**Output:**

- np.array([center_x,center_y,open_x,open_y,height])

graspnetAPI.utils.utils.**batch_rgbdxyz_2_rgbxy_depth**(*points*, *camera*)
**Input:**

- points: np.array(-1,3) of the points in camera frame
- camera: string of the camera type

**Output:**

- coords: float of xy in pixel frame [-1, 2]
- depths: float of the depths of pixel frame [-1]

graspnetAPI.utils.utils.**center_depth**(*depths*, *center*, *open_point*, *upper_point*)

**Input:**

- depths: numpy array of the depths.
- center: numpy array of the center point.
- open_point: numpy array of the open point.
- upper_point: numpy array of the upper point.

**Output:**

- depth: float of the grasp depth.

graspnetAPI.utils.utils.**create_axis**(*length*, *grid_size=0.01*)

graspnetAPI.utils.utils.**create_mesh_box**(*width*, *height*, *depth*, *dx=0*, *dy=0*, *dz=0*)

Author: chenxi-wang Create box instance with mesh representation.

graspnetAPI.utils.utils.**create_point_cloud_from_depth_image**(*depth*, *camera*, *organized=True*)

graspnetAPI.utils.utils.**create_table_cloud**(*width*, *height*, *depth*, *dx=0*, *dy=0*, *dz=0*, *grid_size=0.01*)

Author: chenxi-wang

**Input:**

- width/height/depth: float, table width/height/depth along x/z/y-axis in meters
- dx/dy/dz: float, offset along x/y/z-axis in meters
- grid_size: float, point distance along x/y/z-axis in meters

**Output:**

- open3d.geometry.PointCloud

graspnetAPI.utils.utils.**dexnet_params_to_matrix**(*binormal*, *angle*)

Author: chenxi-wang

**Input:**

- binormal: numpy array of shape (3,).
- angle: float of the angle.

**Output:**

- numpy array of shape (3, 3) of the rotation matrix.

graspnetAPI.utils.utils.**find_scene_by_model_id**(*dataset_root*, *model_id_list*)

graspnetAPI.utils.utils.**framexy_depth_2_xyz**(*pixel_x*, *pixel_y*, *depth*, *camera*)

**Input:**

- pixel_x: int of the pixel x coordinate.
- pixel_y: int of the pixle y coordicate.

- depth: float of depth. The unit is millimeter.
- camera: string of type of camera. “realsense” or “kinect”.

**Output:**

- x, y, z: float of x, y and z coordinates in camera frame. The unit is millimeter.

graspnetAPI.utils.utils.**generate_scene**(*scene_idx*, *anno_idx*, *return_poses=False*, *align=False*, *camera='realsense'*)

graspnetAPI.utils.utils.**generate_scene_model**(*dataset_root*, *scene_name*, *anno_idx*, *return_poses=False*, *align=False*, *camera='realsense'*)

Author: chenxi-wang

**Input:**

- dataset_root: str, graspnet dataset root
- scene_name: str, name of scene folder, e.g. scene_0000
- anno_idx: int, frame index from 0-255
- return_poses: bool, return object ids and 6D poses if set to True
- align: bool, transform to table coordinates if set to True
- camera: str, camera name (realsense or kinect)

**Output:**

- list of open3d.geometry.PointCloud.

graspnetAPI.utils.utils.**generate_scene_pointcloud**(*dataset_root*, *scene_name*, *anno_idx*, *align=False*, *camera='kinect'*)

Author: chenxi-wang

**Input:**

- dataset_root: str, graspnet dataset root
- scene_name: str, name of scene folder, e.g. scene_0000
- anno_idx: int, frame index from 0-255
- align: bool, transform to table coordinates if set to True
- camera: str, camera name (realsense or kinect)

**Output:**

- open3d.geometry.PointCloud.

graspnetAPI.utils.utils.**generate_views**(*N*, *phi=0.6180339887498949*, *center=array([0.0, 0.0, 0.0], dtype=float32)*, *R=1*)

Author: chenxi-wang View sampling on a sphere using Febonacci lattices.

**Input:**

- N: int, number of viewpoints.
- phi: float, constant angle to sample views, usually 0.618.
- center: numpy array of (3,), sphere center.
- R: float, sphere radius.

**Output:**

- numpy array of (N, 3), coordinates of viewpoints.

graspnetAPI.utils.utils.**get_batch_key_points**(*centers*, *Rs*, *widths*)

**Input:**

- centers: np.array(-1,3) of the translation
- Rs: np.array(-1,3,3) of the rotation matrix
- widths: np.array(-1) of the grasp width

**Output:**

- key_points: np.array(-1,4,3) of the key point of the grasp

graspnetAPI.utils.utils.**get_camera_intrinsic**(*camera*)

**Input:**

- camera: string of type of camera, "realsense" or "kinect".

**Output:**

- numpy array of shape (3, 3) of the camera intrinsic matrix.

graspnetAPI.utils.utils.**get_model_grasps**(*datapath*)

Author: chenxi-wang Load grasp labels from .npz files.

graspnetAPI.utils.utils.**get_obj_pose_list**(*camera_pose*, *pose_vectors*)

graspnetAPI.utils.utils.**key_point_2_rotation**(*center_xyz*, *open_point_xyz*, *upper_point_xyz*)

**Input:**

- center_xyz: numpy array of the center point.
- open_point_xyz: numpy array of the open point.
- upper_point_xyz: numpy array of the upper point.

**Output:**

- rotation: numpy array of the rotation matrix.

graspnetAPI.utils.utils.**matrix_to_dexnet_params**(*matrix*)

Author: chenxi-wang

**Input:**

- numpy array of shape (3, 3) of the rotation matrix.

**Output:**

- binormal: numpy array of shape (3,).
- angle: float of the angle.

graspnetAPI.utils.utils.**parse_posevector**(*posevector*)

Author: chenxi-wang Decode posevector to object id and transformation matrix.

graspnetAPI.utils.utils.**plot_axis**(*R*, *center*, *length*, *grid_size=0.01*)

graspnetAPI.utils.utils.**plot_gripper_pro_max**(*center*, *R*, *width*, *depth*, *score=1*, *color=None*)

Author: chenxi-wang

**Input:**

- center: numpy array of (3,), target point as gripper center

- R: numpy array of (3,3), rotation matrix of gripper
- width: float, gripper width
- score: float, grasp quality score

**Output:**

- open3d.geometry.TriangleMesh

graspnetAPI.utils.utils.**rotation_matrix**(*rx*, *ry*, *rz*)
Author: chenxi-wang

**Input:**

- rx/ry/rz: float, rotation angle along x/y/z-axis

**Output:**

- numpy array of (3, 3), rotation matrix.

graspnetAPI.utils.utils.**transform_matrix**(*tx*, *ty*, *tz*, *rx*, *ry*, *rz*)
Author: chenxi-wang

**Input:**

- tx/ty/tz: float, translation along x/y/z-axis
- rx/ry/rz: float, rotation angle along x/y/z-axis

**Output:**

- numpy array of (4, 4), transformation matrix.

graspnetAPI.utils.utils.**transform_points**(*points*, *trans*)
Author: chenxi-wang

**Input:**

- points: numpy array of (N,3), point cloud
- trans: numpy array of (4,4), transformation matrix

**Output:**

- numpy array of (N,3), transformed points.

graspnetAPI.utils.utils.**viewpoint_params_to_matrix**(*towards*, *angle*)
Author: chenxi-wang

**Input:**

- towards: numpy array towards vector with shape (3,).
- angle: float of in-plane rotation.

**Output:**

- numpy array of the rotation matrix with shape (3, 3).

### graspnetAPI.utils.vis module

graspnetAPI.utils.vis.**create_table_cloud**(*width*, *height*, *depth*, *dx=0*, *dy=0*, *dz=0*, *grid_size=0.01*)

Author: chenxi-wang

**Input:**

- width/height/depth: float, table width/height/depth along x/z/y-axis in meters
- dx/dy/dz: float, offset along x/y/z-axis in meters
- grid_size: float, point distance along x/y/z-axis in meters

**Output:**

- open3d.geometry.PointCloud

graspnetAPI.utils.vis.**get_camera_parameters**(*camera='kinect'*)

author: Minghao Gou

**Input:**

- camera: string of type of camera: 'kinect' or 'realsense'

**Output:**

- open3d.camera.PinholeCameraParameters

graspnetAPI.utils.vis.**vis6D**(*dataset_root*, *scene_name*, *anno_idx*, *camera*, *align_to_table=True*, *save_folder='save_fig'*, *show=False*, *per_obj=False*)

**Input:**

- dataset_root: str, graspnet dataset root
- scene_name: str, name of scene folder, e.g. scene_0000
- anno_idx: int, frame index from 0-255
- camera: str, camera name (realsense or kinect)
- align_to_table: bool, transform to table coordinates if set to True
- save_folder: str, folder to save screen captures
- show: bool, show visualization in open3d window if set to True
- per_obj: bool, show pose of each object

graspnetAPI.utils.vis.**visAnno**(*dataset_root*, *scene_name*, *anno_idx*, *camera*, *num_grasp=10*, *th=0.3*, *align_to_table=True*, *max_width=0.08*, *save_folder='save_fig'*, *show=False*, *per_obj=False*)

Author: chenxi-wang

**Input:**

- dataset_root: str, graspnet dataset root
- scene_name: str, name of scene folder, e.g. scene_0000
- anno_idx: int, frame index from 0-255
- camera: str, camera name (realsense or kinect)
- num_grasp: int, number of sampled grasps
- th: float, threshold of friction coefficient
- align_to_table: bool, transform to table coordinates if set to True

- max_width: float, only visualize grasps with width<=max_width
- save_folder: str, folder to save screen captures
- show: bool, show visualization in open3d window if set to True
- per_obj: bool, show grasps on each object

graspnetAPI.utils.vis.**visObjGrasp**(*dataset_root*, *obj_idx*, *num_grasp=10*, *th=0.5*, *max_width=0.08*, *save_folder='save_fig'*, *show=False*)

Author: chenxi-wang

**Input:**

- dataset_root: str, graspnet dataset root
- obj_idx: int, index of object model
- num_grasp: int, number of sampled grasps
- th: float, threshold of friction coefficient
- max_width: float, only visualize grasps with width<=max_width
- save_folder: str, folder to save screen captures
- show: bool, show visualization in open3d window if set to True

graspnetAPI.utils.vis.**vis_rec_grasp**(*rec_grasp_tuples*, *numGrasp*, *image_path*, *save_path*, *show=False*)

author: Minghao Gou

**Input:**

- rec_grasp_tuples: np.array of rectangle grasps
- numGrasp: int of total grasps number to show
- image_path: string of path of the image
- image_path: string of the path to save the image
- show: bool of whether to show the image

**Output:**

- no output but display the rectangle grasps in image

### graspnetAPI.utils.xmlhandler module

graspnetAPI.utils.xmlhandler.**empty_pose_vector**(*objectid*)

graspnetAPI.utils.xmlhandler.**empty_pose_vector_list**(*objectidlist*)

graspnetAPI.utils.xmlhandler.**getframeposevectorlist**(*objectidlist*, *is_resume*, *frame_number*, *xml_dir*)

graspnetAPI.utils.xmlhandler.**getposevectorlist**(*objectidlist*, *is_resume*, *num_frame*, *frame_number*, *xml_dir*)

**class** graspnetAPI.utils.xmlhandler.**xmlReader**(*xmlfilename*)

Bases: object

**get_pose_list**()

**getposevectorlist**()

**gettop**()

**showinfo**()

**class** graspnetAPI.utils.xmlhandler.**xmlWriter**(*topfromreader=None*)
Bases: object

**addobject**(*pose*, *objname*, *objpath*, *objid*)

**objectlistfromposevectorlist**(*posevectorlist*, *objdir*, *objnamelist*, *objidlist*)

**writexml**(*xmlfilename='scene.xml'*)

#### Module contents

### 5.1.2 Submodules

### 5.1.3 graspnetAPI.grasp module

**class** graspnetAPI.grasp.**Grasp**(**args*)
Bases: object

**property depth**
**Output:**

- float of the depth.

**property height**
**Output:**

- float of the height.

**property object_id**
**Output:**

- int of the object id that this grasp grasps

**property rotation_matrix**
**Output:**

- np.array of shape (3, 3) of the rotation matrix.

**property score**
**Output:**

- float of the score.

**to_open3d_geometry**(*color=None*)
**Input:**

- color: optional, tuple of shape (3) denotes (r, g, b), e.g., (1,0,0) for red

**Ouput:**

- list of open3d.geometry.Geometry of the gripper.

**transform**(*T*)
**Input:**

- T: np.array of shape (4, 4)

**Output:**

- Grasp instance after transformation, the original Grasp will also be changed.

**property translation**
**Output:**

- np.array of shape (3,) of the translation.

**property width**
**Output:**

- float of the width.

**class** graspnetAPI.grasp.**GraspGroup**(**args*)
Bases: object

**add**(*element*)
**Input:**

- element: Grasp instance or GraspGroup instance.

**property depths**
**Output:**

- numpy array of shape (-1, ) of the depths.

**from_npy**(*npy_file_path*)
**Input:**

- npy_file_path: string of the file path.

**property heights**
**Output:**

- numpy array of shape (-1, ) of the heights.

**nms**(*translation_thresh=0.03*, *rotation_thresh=0.5235987755982988*)
**Input:**

- translation_thresh: float of the translation threshold.
- rotation_thresh: float of the rotation threshold.

**Output:**

- GraspGroup instance after nms.

**property object_ids**
**Output:**

- numpy array of shape (-1, ) of the object ids.

**random_sample**(*numGrasp=20*)
**Input:**

- numGrasp: int of the number of sampled grasps.

**Output:**

- GraspGroup instance of sample grasps.

**remove**(*index*)
**Input:**

- index: list of the index of grasp

**property rotation_matrices**
**Output:**

- np.array of shape (-1, 3, 3) of the rotation matrices.

**save_npy**(*npy_file_path*)
**Input:**

- npy_file_path: string of the file path.

**property scores**
**Output:**

- numpy array of shape (-1, ) of the scores.

**sort_by_score**(*reverse=False*)
**Input:**

- reverse: bool of order, if False, from high to low, if True, from low to high.

**Output:**

- no output but sort the grasp group.

**to_open3d_geometry_list**()
**Output:**

- list of open3d.geometry.Geometry of the grippers.

**to_rect_grasp_group**(*camera*)
**Input:**

- camera: string of type of camera, 'realsense' or 'kinect'.

**Output:**

- RectGraspGroup instance or None.

**transform**(*T*)
**Input:**

- T: np.array of shape (4, 4)

**Output:**

- GraspGroup instance after transformation, the original GraspGroup will also be changed.

**property translations**
**Output:**

- np.array of shape (-1, 3) of the translations.

**property widths**
**Output:**

- numpy array of shape (-1, ) of the widths.

**class** graspnetAPI.grasp.**RectGrasp**(**args*)
Bases: object

**property center_point**
**Output:**

- tuple of x,y of the center point.

**get_key_points**()
**Output:**

- center, open_point, upper_point, each of them is a numpy array of shape (2,)

**property height**
**Output:**

- float of the height.

**property object_id**
**Output:**

- int of the object id that this grasp grasps

**property open_point**
**Output:**

- tuple of x,y of the open point.

**property score**
**Output:**

- float of the score.

**to_grasp**(*camera*, *depths*, *depth_method=<function center_depth>*)
**Input:**

- camera: string of type of camera, 'kinect' or 'realsense'.
- depths: numpy array of the depths image.
- depth_method: function of calculating the depth.

**Output:**

- grasp: Grasp instance of None if the depth is not valid.

**to_opencv_image**(*opencv_rgb*)
**input:**

- opencv_rgb: numpy array of opencv BGR format.

**Output:**

- numpy array of opencv RGB format that shows the rectangle grasp.

**class** graspnetAPI.grasp.**RectGraspGroup**(**args*)
Bases: object

**add**(*rect_grasp*)
**Input:**

- rect_grasp: RectGrasp instance

**batch_get_key_points**()
**Output:**

- center, open_point, upper_point, each of them is a numpy array of shape (2,)

**property center_points**
**Output:**

- numpy array the center points of shape (-1, 2).

**from_npy**(*npy_file_path*)
**Input:**

- npy_file_path: string of the file path.

**property heights**
**Output:**

- numpy array of the heights.

**property object_ids**
**Output:**

- numpy array of the object ids that this grasp grasps.

**property open_points**
**Output:**

- numpy array the open points of shape (-1, 2).

**random_sample**(*numGrasp=20*)
**Input:**

- numGrasp: int of the number of sampled grasps.

**Output:**

- RectGraspGroup instance of sample grasps.

**remove**(*index*)
**Input:**

- index: list of the index of rect_grasp

**save_npy**(*npy_file_path*)
**Input:**

- npy_file_path: string of the file path.

**property scores**
**Output:**

- numpy array of the scores.

**sort_by_score**(*reverse=False*)
**Input:**

- reverse: bool of order, if False, from high to low, if True, from low to high.

**Output:**

- no output but sort the grasp group.

**to_grasp_group**(*camera*, *depths*, *depth_method=<function batch_center_depth>*)
**Input:**

- camera: string of type of camera, 'kinect' or 'realsense'.
- depths: numpy array of the depths image.
- depth_method: function of calculating the depth.

**Output:**

- grasp_group: GraspGroup instance or None.

---

**Note:** The number may not be the same to the input as some depth may be invalid.

---

**to_opencv_image**(*opencv_rgb*, *numGrasp=0*)
**input:**

- opencv_rgb: numpy array of opencv BGR format.
- numGrasp: int of the number of grasp, 0 for all.

**Output:**

- numpy array of opencv RGB format that shows the rectangle grasps.

### 5.1.4 graspnetAPI.graspnet module

**class** graspnetAPI.graspnet.**GraspNet**(*root*, *camera='kinect'*, *split='train'*)

Bases: object

**checkDataCompleteness**()

Check whether the dataset files are complete.

**Output:**

- bool, True for complete, False for not complete.

**getDataIds**(*sceneIds=None*)

**Input:**

- sceneIds:int or list of int of the scenes ids.

**Output:**

- a list of int of the data ids. Data could be accessed by calling self.loadData(ids).

**getObjIds**(*sceneIds=None*)

**Input:**

- sceneIds: int or list of int of the scene ids.

**Output:**

- a list of int of the object ids in the given scenes.

**getSceneIds**(*objIds=None*)

**Input:**

- objIds: int or list of int of the object ids.

**Output:**

- a list of int of the scene ids that contains **all** the objects.

**loadBGR**(*sceneId*, *camera*, *annId*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.

**Output:**

- numpy array of the rgb in BGR order.

**loadCollisionLabels**(*sceneIds=None*)

**Input:**

- sceneIds: int or list of int of the scene ids.

**Output:**

- dict of the collision labels.

**loadData**(*ids=None*, **extargs*)

**Input:**

- ids: int or list of int of the the data ids.

- extargs: extra arguments. This function can also be called with loadData(sceneId, camera, annId)

**Output:**

- if ids is int, returns a tuple of data path
- if ids is not specified or is a list, returns a tuple of data path lists

**loadDepth**(*sceneId*, *camera*, *annId*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.

**Output:**

- numpy array of the depth with dtype = np.uint16

**loadGrasp**(*sceneId*, *annId=0*, *format='6d'*, *camera='kinect'*, *grasp_labels=None*, *collision_labels=None*, *fric_coef_thresh=0.4*)

**Input:**

- sceneId: int of scene id.
- annId: int of annotation id.
- format: string of grasp format, '6d' or 'rect'.
- camera: string of camera type, 'kinect' or 'realsense'.
- grasp_labels: dict of grasp labels. Call self.loadGraspLabels if not given.
- collision_labels: dict of collision labels. Call self.loadCollisionLabels if not given.
- fric_coef_thresh: float of the frcition coefficient threshold of the grasp.

**ATTENTION**

the LOWER the friction coefficient is, the better the grasp is.

**Output:**

- If format == '6d', return a GraspGroup instance.
- If format == 'rect', return a RectGraspGroup instance.

**loadGraspLabels**(*objIds=None*)

**Input:**

- objIds: int or list of int of the object ids.

**Output:**

- a dict of grasplabels of each object.

**loadMask**(*sceneId*, *camera*, *annId*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.

**Output:**

- numpy array of the mask with dtype = np.uint16

**loadObjModels**(*objIds=None*)

**Function:**

- load object 3D models of the given obj ids

**Input:**

- objIDs: int or list of int of the object ids

**Output:**

- a list of open3d.geometry.PointCloud of the models

**loadObjTrimesh**(*objIds=None*)

**Function:**

- load object 3D trimesh of the given obj ids

**Input:**

- objIDs: int or list of int of the object ids

**Output:**

- a list of trimesh.Trimesh of the models

**loadRGB**(*sceneId*, *camera*, *annId*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.

**Output:**

- numpy array of the rgb in RGB order.

**loadSceneModel**(*sceneId*, *camera='kinect'*, *annId=0*, *align=False*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.
- align: bool of whether align to the table frame.

**Output:**

- open3d.geometry.PointCloud list of the scene models.

**loadScenePointCloud**(*sceneId*, *camera*, *annId*, *align=False*, *format='open3d'*, *use_workspace=False*, *use_mask=True*, *use_inpainting=False*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.
- aligh: bool of whether align to the table frame.
- format: string of the returned type. 'open3d' or 'numpy'
- use_workspace: bool of whether crop the point cloud in the work space.

- **use_mask: bool of whether crop the point cloud use mask(z>0), only open3d 0.9.0 is supported for False option.** Only turn to False if you know what you are doing.
- use_inpainting: bool of whether inpaint the depth image for the missing information.

**Output:**

- open3d.geometry.PointCloud instance of the scene point cloud.
- or tuple of numpy array of point locations and colors.

**loadWorkSpace**(*sceneId*, *camera*, *annId*)

**Input:**

- sceneId: int of the scene index.
- camera: string of type of camera, 'realsense' or 'kinect'
- annId: int of the annotation index.

**Output:**

- tuple of the bounding box coordinates (x1, y1, x2, y2).

**show6DPose**(*sceneIds*, *saveFolder='save_fig'*, *show=False*, *perObj=False*)

**Input:**

- sceneIds: int or list of scene ids.
- saveFolder: string of the folder to store the image.
- show: bool of whether to show the image.
- perObj: bool, show grasps on each object

**Output:**

- No output but to save the rendered image and maybe show the result.

**showObjGrasp**(*objIds=[]*, *numGrasp=10*, *th=0.5*, *maxWidth=0.08*, *saveFolder='save_fig'*, *show=False*)

**Input:**

- objIds: int of list of objects ids.
- numGrasp: how many grasps to show in the image.
- th: threshold of the coefficient of friction.
- maxWidth: float, only visualize grasps with width<=maxWidth
- saveFolder: string of the path to save the rendered image.
- show: bool of whether to show the image.

**Output:**

- No output but save the rendered image and maybe show it.

**showSceneGrasp**(*sceneId*, *camera='kinect'*, *annId=0*, *format='6d'*, *numGrasp=20*, *show_object=True*, *coef_fric_thresh=0.1*)

**Input:**

- sceneId: int of the scene index.
- camera: string of the camera type, 'realsense' or 'kinect'.
- annId: int of the annotation index.
- format: int of the annotation type, 'rect' or '6d'.

- numGrasp: int of the displayed grasp number, grasps will be randomly sampled.
- coef_fric_thresh: float of the friction coefficient of grasps.

### 5.1.5 graspnetAPI.graspnet_eval module

**class** graspnetAPI.graspnet_eval.**GraspNetEval**(*root*, *camera*, *split='test'*)

Bases: *graspnetAPI.graspnet.GraspNet*

Class for evaluation on GraspNet dataset.

**Input:**

- root: string of root path for the dataset.
- camera: string of type of the camera.
- split: string of the date split.

**eval_all**(*dump_folder*, *proc=2*)

**Input:**

- dump_folder: string of the folder that saves the npy files.
- proc: int of the number of processes to use to evaluate.

**Output:**

- res: numpy array of the detailed accuracy.
- ap: float of the AP for all split.

**eval_novel**(*dump_folder*, *proc=2*)

**Input:**

- dump_folder: string of the folder that saves the npy files.
- proc: int of the number of processes to use to evaluate.

**Output:**

- res: numpy array of the detailed accuracy.
- ap: float of the AP for novel split.

**eval_scene**(*scene_id*, *dump_folder*, *TOP_K=50*, *return_list=False*, *vis=False*, *max_width=0.1*)

**Input:**

- scene_id: int of the scene index.
- dump_folder: string of the folder that saves the dumped npy files.
- TOP_K: int of the top number of grasp to evaluate
- return_list: bool of whether to return the result list.
- vis: bool of whether to show the result
- max_width: float of the maximum gripper width in evaluation

**Output:**

- scene_accuracy: np.array of shape (256, 50, 6) of the accuracy tensor.

**eval_seen**(*dump_folder*, *proc=2*)

**Input:**

- dump_folder: string of the folder that saves the npy files.

- proc: int of the number of processes to use to evaluate.

**Output:**

- res: numpy array of the detailed accuracy.
- ap: float of the AP for seen split.

**eval_similar**(*dump_folder*, *proc=2*)

**Input:**

- dump_folder: string of the folder that saves the npy files.
- proc: int of the number of processes to use to evaluate.

**Output:**

- res: numpy array of the detailed accuracy.
- ap: float of the AP for similar split.

**get_model_poses**(*scene_id*, *ann_id*)

**Input:**

- scene_id: int of the scen index.
- ann_id: int of the annotation index.

**Output:**

- obj_list: list of int of object index.
- pose_list: list of 4x4 matrices of object poses.
- camera_pose: 4x4 matrix of the camera pose relative to the first frame.
- align mat: 4x4 matrix of camera relative to the table.

**get_scene_models**(*scene_id*, *ann_id*)

return models in model coordinate

**parallel_eval_scenes**(*scene_ids*, *dump_folder*, *proc=2*)

**Input:**

- scene_ids: list of int of scene index.
- dump_folder: string of the folder that saves the npy files.
- proc: int of the number of processes to use to evaluate.

**Output:**

- scene_acc_list: list of the scene accuracy.

### 5.1.6 Module contents

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## g

# INDEX

## D



## E



## F

## G

## M

## N



## O



## P

## Q



## R

## S

## T

## U



## V



## W



## X