2

September 30, 2020

[1]:
```python
import numpy as np
import cvxpy as cp
np.set_printoptions(precision=3, suppress=True)
```

[2]:
```python
def get_contacts():
    """
        Return contact normals and locations as a matrix
        :return: <np.array>, <np.array> locations and normal matrices
    """
    # ------------------------------------------------
    # FILL WITH YOUR CODE
    l = 1

    R = np.zeros((2,3))
    N = np.zeros((2,3))
    # print(R,"\t",N)
    R[:,0] = np.array([(2+np.sqrt(2))/4/(1+np.sqrt(2))*l,(2+np.sqrt(2))/4/(1+np.
 ↪sqrt(2))*l]).reshape(1,2)
    R[:,1] = np.array([-.5,0]).reshape(1,2)
    R[:,2] = np.array([0,-.5]).reshape(1,2)

    N[:,2] = np.array([0,1]).reshape(1,2)
    N[:,1] = np.array([1,0]).reshape(1,2)
    N[:,0] = np.array([-np.cos(np.pi/4),-np.sin(np.pi/4)]).reshape(1,2)
    # ------------------------------------------------
    return R, N

# test
R,N = get_contacts()
print(R,"\n",N)
```

```
[[ 0.354 -0.5    0.   ]
 [ 0.354  0.    -0.5  ]]
 [[-0.707  1.     0.   ]
 [-0.707  0.     1.   ]]
```

[3]:
```
def calculate_grasp(R, N):
    """
        Return the grasp matrix as a function of contact locations and normals
        :param R: <np.array> locations of contact
        :param N: <np.array> contact normals
        :return: <np.array> Grasp matrix for Fig. 1
    """
    # ------------------------------------------------
    # FILL WITH YOUR CODE
    G = np.zeros((3,6))
    J = np.zeros((3,2))

    for idx in range(3):
        rx = R[0][idx]
        ry = R[1][idx]
        # print(rx,ry)
        nx = N[0][idx]
        ny = N[1][idx]
        # print(nx,ny)
        J = np.array([[ny, nx],
                      [-nx, ny],
                      [rx*nx+ry*ny, ry*nx-rx*ny]])
        # print(J)
        G[:,[idx*2,idx*2+1]] = J

    # ------------------------------------------------
    return G

# test
G = calculate_grasp(R, N)
print(G)
```

```
[[-0.707 -0.707  0.     1.     1.     0.   ]
 [ 0.707 -0.707 -1.     0.    -0.     1.   ]
 [-0.5   -0.    -0.5    0.    -0.5   -0.   ]]
```

[4]:
```
def calculate_facet(mu):
    """
        Return friction cone representation in terms of facet normals
        :param mu: <float> coefficient of friction
        :return: <np.array> Facet normal matrix
    """
    # ------------------------------------------------
    # FILL WITH YOUR CODE
    mu_mat = np.array([[1,mu],
                      [-1,mu]])
    # print(mu_mat)
```

```python
    f = 1/np.sqrt(1+mu**2)*mu_mat
    # print(f)
    F = np.zeros((6, 6))
    # print(F)
    for idx in range(3):
        F[2*idx:2*idx+2,2*idx:2*idx+2] = f
    # ------------------------------------------------
    return F

# test
mu = 0.3
F = calculate_facet(mu)
print(F)
```

```
[[ 0.958  0.287  0.     0.     0.     0.   ]
 [-0.958  0.287  0.     0.     0.     0.   ]
 [ 0.     0.     0.958  0.287  0.     0.   ]
 [ 0.     0.    -0.958  0.287  0.     0.   ]
 [ 0.     0.     0.     0.     0.958  0.287]
 [ 0.     0.     0.     0.    -0.958  0.287]]
```

[5]:

```python
def compute_grasp_rank(G):
    """
        Return boolean of if grasp has rank 3 or not
        :param G: <np.array> grasp matrix as a numpy array
        :return: <bool> boolean flag for if rank is 3 or not
    """
    # ------------------------------------------------
    # FILL WITH YOUR CODE
    r = np.linalg.matrix_rank(G)
    flag = (r == 3)
    # ------------------------------------------------
    return flag

# test
flag = compute_grasp_rank(G)
print(flag)
```

```
True
```

[6]:

```python
def compute_constraints(G, F):
    """
        Return grasp constraints as numpy arrays
        :param G: <np.array> grasp matrix as a numpy array
        :param F: <np.array> friction cone facet matrix as a numpy array
        :return: <np.array>x5 contact constraints
    """
```

```
    # ------------------------------------------------
    # FILL WITH YOUR CODE

    A = np.zeros((3,7))   # TODO: Replace None with your result
    b = np.zeros((3,1))   # TODO: Replace None with your result
    P = np.zeros((8,7))   # TODO: Replace None with your result
    q = np.zeros((8,1))   # TODO: Replace None with your result
    c = np.zeros((7,1))   # TODO: Replace None with your result

    A[:,0:-1] = G

    P[0:-2,0:-1] = 0-F
    P[-2,0:-1] = [0,1,0,1,0,1]
    P[0:-2,-1] = 1
    P[-1,-1] = -1


    q[-2] = 3
    q[-1] = 0

    c[-1] = 1

    # ------------------------------------------------
    return A, b.reshape(3,), P, q.reshape(8,), c.reshape(7,)

# test
A, b, P, q, c = compute_constraints(G, F)
print(A,"\n", b,"\n", P,"\n", q,"\n", c)
```

```
[[-0.707 -0.707  0.     1.     1.     0.     0.   ]
 [ 0.707 -0.707 -1.     0.    -0.     1.     0.   ]
 [-0.5   -0.    -0.5    0.    -0.5   -0.     0.   ]]
 [0. 0. 0.]
 [[-0.958 -0.287  0.     0.     0.     0.     1.   ]
 [ 0.958 -0.287  0.     0.     0.     0.     1.   ]
 [ 0.     0.    -0.958 -0.287  0.     0.     1.   ]
 [ 0.     0.     0.958 -0.287  0.     0.     1.   ]
 [ 0.     0.     0.     0.    -0.958 -0.287  1.   ]
 [ 0.     0.     0.     0.     0.958 -0.287  1.   ]
 [ 0.     1.     0.     1.     0.     1.     0.   ]
 [ 0.     0.     0.     0.     0.     0.    -1.   ]]
 [0. 0. 0. 0. 0. 0. 3. 0.]
 [0. 0. 0. 0. 0. 0. 1.]
```

[7]:
```
def check_force_closure(A, b, P, q, c):
    """
        Solves Linear program given grasp constraints - DO NOT EDIT
```

```
        :return: d_star
    """
    # ------------------------------------------------
    # DO NOT EDIT THE CODE IN THIS FUNCTION
    x = cp.Variable(A.shape[1])

    prob = cp.Problem(cp.Maximize(c.T@x),
                      [P @ x <= q, A @ x == b])
    prob.solve()
    d = prob.value
    print('Optimal value of d (d^*): {:3.2f}'.format(d))
    return d

# test
d = check_force_closure(A, b, P, q, c)
print(d)
```

```
Optimal value of d (d^*): 0.25
0.2524867413859962
```

[8]:
```
if __name__ == "__main__":
    mu = 0.3
    R,N = get_contacts()
    print("Part 1 - Contact Locations and Normals")
    print(R,"\n\n",N)
    G = calculate_grasp(R, N)
    print("Part 2 - Contact Jacobians and the Grasp Matrix")
    print(G)
    F = calculate_facet(mu)
    print("Part 3 - Friction Cone Facet Normals")
    print(F)
    flag = compute_grasp_rank(G)
    print("Part 4 - Grasp Rank")
    print(flag)
    A, b, P, q, c = compute_constraints(G, F)
    print("Part 5 - Grasp Constraints and Force Closure Test")
    print(A,"\n\n", b,"\n\n", P,"\n\n", q,"\n\n", c)
    print(A.shape,b.shape,P.shape,q.shape,c.shape)
    d = check_force_closure(A, b, P, q, c)
    print("Part 6 - Definitely Something")
    print(d)
    if(d!=0):
        print("Yup")
    else:
        print("Nah")
```

```
Part 1 - Contact Locations and Normals
[[ 0.354 -0.5    0.   ]
```

```
 [ 0.354  0.    -0.5  ]]

 [[-0.707  1.     0.   ]
 [-0.707  0.     1.   ]]
Part 2 - Contact Jacobians and the Grasp Matrix
[[-0.707 -0.707  0.     1.     1.     0.   ]
 [ 0.707 -0.707 -1.     0.    -0.     1.   ]
 [-0.5   -0.    -0.5    0.    -0.5   -0.   ]]
Part 3 - Friction Cone Facet Normals
[[ 0.958  0.287  0.     0.     0.     0.   ]
 [-0.958  0.287  0.     0.     0.     0.   ]
 [ 0.     0.     0.958  0.287  0.     0.   ]
 [ 0.     0.    -0.958  0.287  0.     0.   ]
 [ 0.     0.     0.     0.     0.958  0.287]
 [ 0.     0.     0.     0.    -0.958  0.287]]
Part 4 - Grasp Rank
True
Part 5 - Grasp Constraints and Force Closure Test
[[-0.707 -0.707  0.     1.     1.     0.     0.   ]
 [ 0.707 -0.707 -1.     0.    -0.     1.     0.   ]
 [-0.5   -0.    -0.5    0.    -0.5   -0.     0.   ]]

 [0. 0. 0.]

 [[-0.958 -0.287  0.     0.     0.     0.     1.   ]
 [ 0.958 -0.287  0.     0.     0.     0.     1.   ]
 [ 0.     0.    -0.958 -0.287  0.     0.     1.   ]
 [ 0.     0.     0.958 -0.287  0.     0.     1.   ]
 [ 0.     0.     0.     0.    -0.958 -0.287  1.   ]
 [ 0.     0.     0.     0.     0.958 -0.287  1.   ]
 [ 0.     1.     0.     1.     0.     1.     0.   ]
 [ 0.     0.     0.     0.     0.     0.    -1.   ]]

 [0. 0. 0. 0. 0. 0. 3. 0.]

 [0. 0. 0. 0. 0. 0. 1.]
(3, 7) (3,) (8, 7) (8,) (7,)
Optimal value of d (d^*): 0.25
Part 6 - Definitely Something
0.2524867413859962
Yup
```