```python
import numpy as np, sympy as sy

#define numerical rotation matricies
def rotation_z(theta):
  return np.matrix([
    [ np.cos(theta), -np.sin(theta), 0],
    [ np.sin(theta),  np.cos(theta), 0],
    [0, 0, 1]
    ])
def rotation_y(theta):
  return np.matrix([
    [ np.cos(theta), 0,  np.sin(theta)],
    [0, 1, 0],
    [-np.sin(theta), 0,  np.cos(theta)]
    ])
def rotation_x(theta):
  return np.matrix([
    [1, 0, 0],
    [0, np.cos(theta), -np.sin(theta)],
    [0, np.sin(theta),  np.cos(theta)]
    ])

# define symbolic rotation matricies
def sym_rotation_z(theta):
  return sy.Matrix([
    [ sy.cos(theta), -sy.sin(theta), 0],
    [ sy.sin(theta),  sy.cos(theta), 0],
    [0, 0, 1]
    ])
def sym_rotation_y(theta):
  return sy.Matrix([
    [ sy.cos(theta), 0,  sy.sin(theta)],
    [0, 1, 0],
    [-sy.sin(theta), 0,  sy.cos(theta)]
    ])
def sym_rotation_x(theta):
  return sy.Matrix([
    [1, 0, 0],
    [0, sy.cos(theta), -sy.sin(theta)],
    [0, sy.sin(theta),  sy.cos(theta)]
    ])

# define numerical translation matricies
def translation_z( d):
  return np.matrix([
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,d],
    [0,0,0,1]
    ])
def translation_y( d):
  return np.matrix([
    [1,0,0,0],
    [0,1,0,d],
    [0,0,1,0],
    [0,0,0,1]
    ])
def translation_x( d):
  return np.matrix([
    [1,0,0,d],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1]
    ])

# define symbolic translation matrices
def sym_translation_z( d):
  return sy.Matrix([
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,d],
    [0,0,0,1]
    ])
```

```
def sym_translation_y( d):
  return sy.Matrix([
    [1,0,0,0],
    [0,1,0,d],
    [0,0,1,0],
    [0,0,0,1]
    ])
def sym_translation_x( d):
  return sy.Matrix([
    [1,0,0,d],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1]
    ])
# numerically convert 3x3 rotation to 4x4 rotation
def convert3x3to4x4( matrix):
  # add column of zeroes
  matrix = np.hstack((matrix, np.transpose([np.zeros(3)])))
  # add row of 0,0,0,1
  matrix = np.vstack((matrix, np.array([0,0,0,1])))
  return matrix
# symbolically convert 3x3 rotation to 4x4 rotation
def syms_convert3x3to4x4( matrix):
  # add column of zeroes
  matrix = sy.Matrix.hstack(matrix, sy.Matrix(sy.zeros(3)[:,2]))
  # add row of 0,0,0,1
  matrix = sy.Matrix.vstack((matrix, sy.Matrix([0,0,0,1]).T))
  return matrix

def denavit_hartenberg( link):
  return (
    translation_z(link[2])*convert3x3to4x4(rotation_z(link[3]))*
    translation_x(link[0])*convert3x3to4x4(rotation_x(link[1]))
    )
def sym_denavit_hartenberg( link):
  return (
    sym_translation_z(link[2])*sy.Matrix(syms_convert3x3to4x4(sym_rotation_z(link[3]
)))*
    sym_translation_x(link[0])*sy.Matrix(syms_convert3x3to4x4(sym_rotation_x(link[1]
)))
    )

def get_A0n( link_list):
  A0i = np.identity(4)
  A0n = []
  for link in link_list:
    A0i = A0i*denavit_hartenberg( link)
    A0n.append(A0i)
  return A0n

def sym_get_A0n( link_list):
  A0i = np.identity(4)
  A0n = []
  for link in link_list:
    A0i = A0i*sym_denavit_hartenberg( link)
    A0n.append(A0i)
  return A0n


def jacobian( link_list):
  A0n = get_A0n( link_list)
  # rotational vectors
  R = [np.matrix(np.identity(3))]
  R = R + [end[:3,:3] for end in A0n]
  # positions of each end effector
  O = [sy.Matrix([[0],[0],[0]])]
  O = O + [np.matrix(end[:,3][:3]) for end in A0n]
  # unit z vector
  k = np.matrix([[0],[0],[1]])
  J_v = []
  J_w = []
  for i in range(len(A0n)):
    # if theta_i is 0 then joint is prismatic
```

```
    if (link_list[i][3] == 0):
      print np.shape(sy.Matrix(R[i]*k))
      print sy.Matrix(R[i]*k)
      print np.shape(sy.Matrix([[0],[0],[0]]))
      J_v.append(sy.Matrix(R[i]*k))
      J_w.append(sy.Matrix([[0],[0],[0]]))
    # if theta_i is not 0 then joint is revolute
    else:
      J_v.append( np.matrix(np.cross( (R[i]*k).T, (O[-1]-O[i]).T)).T )
      J_w.append(np.matrix(R[i]*k))
  J = [np.vstack( (J_v[i], J_w[i])) for i in range(len(J_v))]
  J = np.hstack(J)
  return J


def symbolic_jacobian( link_list):
  A0n = sym_get_A0n( link_list)
  # rotational vectors
  R = [sy.Matrix(np.identity(3))]
  R = R + [end[:3,:3] for end in A0n]
  # positions of each end effector
  O = [sy.Matrix([[0],[0],[0]])]
  O = O + [sy.Matrix(end[:,3][:3]) for end in A0n]
  # unit z vector
  k = sy.Matrix([[0],[0],[1]])
  J_v = []
  J_w = []
  for i in range(len(A0n)):
    # if theta_i is 0 then joint is prismatic
    if (link_list[i][3] == 0):
      J_v.append(sy.Matrix(R[i]*k))
      J_w.append(sy.Matrix([[0],[0],[0]]))
    # if theta_i is not 0 then joint is revolute
    else:
      J_v.append((sy.Matrix(R[i]*k).cross(O[-1]-O[i])))
      J_w.append(sy.Matrix(R[i]*k))
  J = [sy.Matrix.vstack( J_v[i], J_w[i]) for i in range(len(J_v))]
  J = sy.Matrix.hstack(J)
  Je= J[0]
  for i in range(len(J)-1):
    j = i+1
    Je = sy.Matrix.hstack(Je, J[j])
  return Je


#def rotational_velocity_jacobian( link_list)
#def jacobian(link_list)
```