```
from __future__ import print_function
'''
***************************************************************************
Modern Robotics: Mechanics, Planning, and Control.
Code Library
***************************************************************************
Author: Huan Weng, Bill Hunt, Jarvis Schultz, Mikhail Todes,
Email: huanweng@u.northwestern.edu
Date: January 2018
***************************************************************************
Language: Python
Also available in: MATLAB, Mathematica
Required library: numpy
Optional library: matplotlib
***************************************************************************
'''

'''
*** IMPORTS ***
'''

import numpy as np

'''
*** BASIC HELPER FUNCTIONS ***
'''

def NearZero(z):
    """Determines whether a scalar is small enough to be treated as zero
    :param z: A scalar input to check
    :return: True if z is close to zero, false otherwise
    Example Input:
        z = -1e-7
    Output:
        True
    """
    return abs(z) < 1e-6

def Normalize(V):
    """Normalizes a vector
    :param V: A vector
    :return: A unit vector pointing in the same direction as z
    Example Input:
        V = np.array([1, 2, 3])
    Output:
        np.array([0.26726124, 0.53452248, 0.80178373])
    """
    return V / np.linalg.norm(V)

'''
*** CHAPTER 3: RIGID-BODY MOTIONS ***
'''

def RotInv(R):
```

```
    """Inverts a rotation matrix
    :param R: A rotation matrix
    :return: The inverse of R
    Example Input:
        R = np.array([[0, 0, 1],
                      [1, 0, 0],
                      [0, 1, 0]])
    Output:
        np.array([[0, 1, 0],
                  [0, 0, 1],
                  [1, 0, 0]])
    """
    return np.array(R).T

def VecToso3(omg):
    """Converts a 3-vector to an so(3) representation
    :param omg: A 3-vector
    :return: The skew symmetric representation of omg
    Example Input:
        omg = np.array([1, 2, 3])
    Output:
        np.array([[ 0, -3,  2],
                  [ 3,  0, -1],
                  [-2,  1,  0]])
    """
    return np.array([[0,      -omg[2],  omg[1]],
                     [omg[2],       0, -omg[0]],
                     [-omg[1], omg[0],       0]])

def so3ToVec(so3mat):
    """Converts an so(3) representation to a 3-vector
    :param so3mat: A 3x3 skew-symmetric matrix
    :return: The 3-vector corresponding to so3mat
    Example Input:
        so3mat = np.array([[ 0, -3,  2],
                           [ 3,  0, -1],
                           [-2,  1,  0]])
    Output:
        np.array([1, 2, 3])
    """
    return np.array([so3mat[2][1], so3mat[0][2], so3mat[1][0]])

def AxisAng3(expc3):
    """Converts a 3-vector of exponential coordinates for rotation into
    axis-angle form
    :param expc3: A 3-vector of exponential coordinates for rotation
    :return omghat: A unit rotation axis
    :return theta: The corresponding rotation angle
    Example Input:
        expc3 = np.array([1, 2, 3])
    Output:
        (np.array([0.26726124, 0.53452248, 0.80178373]), 3.7416573867739413)
    """
    return (Normalize(expc3), np.linalg.norm(expc3))
```

```
def MatrixExp3(so3mat):
    """Computes the matrix exponential of a matrix in so(3)
    :param so3mat: A 3x3 skew-symmetric matrix
    :return: The matrix exponential of so3mat
    Example Input:
        so3mat = np.array([[ 0, -3,  2],
                           [ 3,  0, -1],
                           [-2,  1,  0]])
    Output:
        np.array([[-0.69492056,  0.71352099,  0.08929286],
                  [-0.19200697, -0.30378504,  0.93319235],
                  [ 0.69297817,  0.6313497 ,  0.34810748]])
    """
    omgtheta = so3ToVec(so3mat)
    if NearZero(np.linalg.norm(omgtheta)):
        return np.eye(3)
    else:
        theta = AxisAng3(omgtheta)[1]
        omgmat = so3mat / theta
        return np.eye(3) + np.sin(theta) * omgmat \
               + (1 - np.cos(theta)) * np.dot(omgmat, omgmat)

def MatrixLog3(R):
    """Computes the matrix logarithm of a rotation matrix
    :param R: A 3x3 rotation matrix
    :return: The matrix logarithm of R
    Example Input:
        R = np.array([[0, 0, 1],
                      [1, 0, 0],
                      [0, 1, 0]])
    Output:
        np.array([[          0, -1.20919958,  1.20919958],
                  [ 1.20919958,           0, -1.20919958],
                  [-1.20919958,  1.20919958,           0]])
    """
    acosinput = (np.trace(R) - 1) / 2.0
    if acosinput >= 1:
        return np.zeros((3, 3))
    elif acosinput <= -1:
        if not NearZero(1 + R[2][2]):
            omg = (1.0 / np.sqrt(2 * (1 + R[2][2]))) \
                  * np.array([R[0][2], R[1][2], 1 + R[2][2]])
        elif not NearZero(1 + R[1][1]):
            omg = (1.0 / np.sqrt(2 * (1 + R[1][1]))) \
                  * np.array([R[0][1], 1 + R[1][1], R[2][1]])
        else:
            omg = (1.0 / np.sqrt(2 * (1 + R[0][0]))) \
                  * np.array([1 + R[0][0], R[1][0], R[2][0]])
        return VecToso3(np.pi * omg)
    else:
        theta = np.arccos(acosinput)
        return theta / 2.0 / np.sin(theta) * (R - np.array(R).T)
```

```
def RpToTrans(R, p):
    """Converts a rotation matrix and a position vector into homogeneous
    transformation matrix
    :param R: A 3x3 rotation matrix
    :param p: A 3-vector
    :return: A homogeneous transformation matrix corresponding to the inputs
    Example Input:
        R = np.array([[1, 0,  0],
                      [0, 0, -1],
                      [0, 1,  0]])
        p = np.array([1, 2, 5])
    Output:
        np.array([[1, 0,  0, 1],
                  [0, 0, -1, 2],
                  [0, 1,  0, 5],
                  [0, 0,  0, 1]])
    """
    return np.r_[np.c_[R, p], [[0, 0, 0, 1]]]

def TransToRp(T):
    """Converts a homogeneous transformation matrix into a rotation matrix
    and position vector
    :param T: A homogeneous transformation matrix
    :return R: The corresponding rotation matrix,
    :return p: The corresponding position vector.
    Example Input:
        T = np.array([[1, 0,  0, 0],
                      [0, 0, -1, 0],
                      [0, 1,  0, 3],
                      [0, 0,  0, 1]])
    Output:
        (np.array([[1, 0,  0],
                   [0, 0, -1],
                   [0, 1,  0]]),
         np.array([0, 0, 3]))
    """
    T = np.array(T)
    return T[0: 3, 0: 3], T[0: 3, 3]

def TransInv(T):
    """Inverts a homogeneous transformation matrix
    :param T: A homogeneous transformation matrix
    :return: The inverse of T
    Uses the structure of transformation matrices to avoid taking a matrix
    inverse, for efficiency.
    Example input:
        T = np.array([[1, 0,  0, 0],
                      [0, 0, -1, 0],
                      [0, 1,  0, 3],
                      [0, 0,  0, 1]])
    Output:
        np.array([[1,  0, 0,  0],
                  [0,  0, 1, -3],
                  [0, -1, 0,  0],
```

```python
              [0,  0, 0,  1]])
    """
    R, p = TransToRp(T)
    Rt = np.array(R).T
    return np.r_[np.c_[Rt, -np.dot(Rt, p)], [[0, 0, 0, 1]]]

def VecTose3(V):
    """Converts a spatial velocity vector into a 4x4 matrix in se3
    :param V: A 6-vector representing a spatial velocity
    :return: The 4x4 se3 representation of V
    Example Input:
        V = np.array([1, 2, 3, 4, 5, 6])
    Output:
        np.array([[ 0, -3,  2, 4],
                  [ 3,  0, -1, 5],
                  [-2,  1,  0, 6],
                  [ 0,  0,  0, 0]])
    """
    return np.r_[np.c_[VecToso3([V[0], V[1], V[2]]), [V[3], V[4], V[5]]],
                 np.zeros((1, 4))]

def se3ToVec(se3mat):
    """ Converts an se3 matrix into a spatial velocity vector
    :param se3mat: A 4x4 matrix in se3
    :return: The spatial velocity 6-vector corresponding to se3mat
    Example Input:
        se3mat = np.array([[ 0, -3,  2, 4],
                           [ 3,  0, -1, 5],
                           [-2,  1,  0, 6],
                           [ 0,  0,  0, 0]])
    Output:
        np.array([1, 2, 3, 4, 5, 6])
    """
    return np.r_[[se3mat[2][1], se3mat[0][2], se3mat[1][0]],
                 [se3mat[0][3], se3mat[1][3], se3mat[2][3]]]

def Adjoint(T):
    """Computes the adjoint representation of a homogeneous transformation
    matrix
    :param T: A homogeneous transformation matrix
    :return: The 6x6 adjoint representation [AdT] of T
    Example Input:
        T = np.array([[1, 0,  0, 0],
                      [0, 0, -1, 0],
                      [0, 1,  0, 3],
                      [0, 0,  0, 1]])
    Output:
        np.array([[1, 0,  0, 0, 0,  0],
                  [0, 0, -1, 0, 0,  0],
                  [0, 1,  0, 0, 0,  0],
                  [0, 0,  3, 1, 0,  0],
                  [3, 0,  0, 0, 0, -1],
                  [0, 0,  0, 0, 1,  0]])
    """
```

```
    R, p = TransToRp(T)
    return np.r_[np.c_[R, np.zeros((3, 3))],
                 np.c_[np.dot(VecToso3(p), R), R]]

def ScrewToAxis(q, s, h):
    """Takes a parametric description of a screw axis and converts it to a
    normalized screw axis
    :param q: A point lying on the screw axis
    :param s: A unit vector in the direction of the screw axis
    :param h: The pitch of the screw axis
    :return: A normalized screw axis described by the inputs
    Example Input:
        q = np.array([3, 0, 0])
        s = np.array([0, 0, 1])
        h = 2
    Output:
        np.array([0, 0, 1, 0, -3, 2])
    """
    return np.r_[s, np.cross(q, s) + np.dot(h, s)]

def AxisAng6(expc6):
    """Converts a 6-vector of exponential coordinates into screw axis-angle
    form
    :param expc6: A 6-vector of exponential coordinates for rigid-body motion
                  S*theta
    :return S: The corresponding normalized screw axis
    :return theta: The distance traveled along/about S
    Example Input:
        expc6 = np.array([1, 0, 0, 1, 2, 3])
    Output:
        (np.array([1.0, 0.0, 0.0, 1.0, 2.0, 3.0]), 1.0)
    """
    theta = np.linalg.norm([expc6[0], expc6[1], expc6[2]])
    if NearZero(theta):
        theta = np.linalg.norm([expc6[3], expc6[4], expc6[5]])
    return (np.array(expc6 / theta), theta)

def MatrixExp6(se3mat):
    """Computes the matrix exponential of an se3 representation of
    exponential coordinates
    :param se3mat: A matrix in se3
    :return: The matrix exponential of se3mat
    Example Input:
        se3mat = np.array([[0,          0,           0,          0],
                           [0,          0, -1.57079632, 2.35619449],
                           [0, 1.57079632,           0, 2.35619449],
                           [0,          0,           0,          0]])
    Output:
        np.array([[1.0, 0.0,  0.0, 0.0],
                  [0.0, 0.0, -1.0, 0.0],
                  [0.0, 1.0,  0.0, 3.0],
                  [  0,   0,    0,   1]])
    """
    se3mat = np.array(se3mat)
```

```
    omgtheta = so3ToVec(se3mat[0: 3, 0: 3])
    if NearZero(np.linalg.norm(omgtheta)):
        return np.r_[np.c_[np.eye(3), se3mat[0: 3, 3]], [[0, 0, 0, 1]]]
    else:
        theta = AxisAng3(omgtheta)[1]
        omgmat = se3mat[0: 3, 0: 3] / theta
        return np.r_[np.c_[MatrixExp3(se3mat[0: 3, 0: 3]),
                           np.dot(np.eye(3) * theta \
                                  + (1 - np.cos(theta)) * omgmat \
                                  + (theta - np.sin(theta)) \
                                    * np.dot(omgmat,omgmat),
                                  se3mat[0: 3, 3]) / theta],
                     [[0, 0, 0, 1]]]

def MatrixLog6(T):
    """Computes the matrix logarithm of a homogeneous transformation matrix
    :param R: A matrix in SE3
    :return: The matrix logarithm of R
    Example Input:
        T = np.array([[1, 0,  0, 0],
                      [0, 0, -1, 0],
                      [0, 1,  0, 3],
                      [0, 0,  0, 1]])
    Output:
        np.array([[0,          0,           0,           0]
                  [0,          0, -1.57079633,  2.35619449]
                  [0, 1.57079633,           0,  2.35619449]
                  [0,          0,           0,           0]])
    """
    R, p = TransToRp(T)
    omgmat = MatrixLog3(R)
    if np.array_equal(omgmat, np.zeros((3, 3))):
        return np.r_[np.c_[np.zeros((3, 3)),
                           [T[0][3], T[1][3], T[2][3]]],
                     [[0, 0, 0, 0]]]
    else:
        theta = np.arccos((np.trace(R) - 1) / 2.0)
        return np.r_[np.c_[omgmat,
                           np.dot(np.eye(3) - omgmat / 2.0 \
                           + (1.0 / theta - 1.0 / np.tan(theta / 2.0) / 2) \
                              * np.dot(omgmat,omgmat) / theta,[T[0][3],
                                                               T[1][3],
                                                               T[2][3]])],
                     [[0, 0, 0, 0]]]

def ProjectToSO3(mat):
    """Returns a projection of mat into SO(3)
    :param mat: A matrix near SO(3) to project to SO(3)
    :return: The closest matrix to R that is in SO(3)
    Projects a matrix mat to the closest matrix in SO(3) using singular-value
    decomposition (see
    http://hades.mech.northwestern.edu/index.php/Modern_Robotics_Linear_Algebra_Review).
    This function is only appropriate for matrices close to SO(3).
    Example Input:
```

```
        mat = np.array([[ 0.675,  0.150,  0.720],
                        [ 0.370,  0.771, -0.511],
                        [-0.630,  0.619,  0.472]])
    Output:
        np.array([[ 0.67901136,  0.14894516,  0.71885945],
                  [ 0.37320708,  0.77319584, -0.51272279],
                  [-0.63218672,  0.61642804,  0.46942137]])
    """
    U, s, Vh = np.linalg.svd(mat)
    R = np.dot(U, Vh)
    if np.linalg.det(R) < 0:
    # In this case the result may be far from mat.
        R[:, s[2, 2]] = -R[:, s[2, 2]]
    return R

def ProjectToSE3(mat):
    """Returns a projection of mat into SE(3)
    :param mat: A 4x4 matrix to project to SE(3)
    :return: The closest matrix to T that is in SE(3)
    Projects a matrix mat to the closest matrix in SE(3) using singular-value
    decomposition (see
    http://hades.mech.northwestern.edu/index.php/Modern_Robotics_Linear_Algebra_Review).
    This function is only appropriate for matrices close to SE(3).
    Example Input:
        mat = np.array([[ 0.675,  0.150,  0.720,  1.2],
                        [ 0.370,  0.771, -0.511,  5.4],
                        [-0.630,  0.619,  0.472,  3.6],
                        [ 0.003,  0.002,  0.010,  0.9]])
    Output:
        np.array([[ 0.67901136,  0.14894516,  0.71885945,  1.2 ],
                  [ 0.37320708,  0.77319584, -0.51272279,  5.4 ],
                  [-0.63218672,  0.61642804,  0.46942137,  3.6 ],
                  [ 0.        ,  0.        ,  0.        ,  1.  ]])
    """
    mat = np.array(mat)
    return RpToTrans(ProjectToSO3(mat[:3, :3]), mat[:3, 3])

def DistanceToSO3(mat):
    """Returns the Frobenius norm to describe the distance of mat from the
    SO(3) manifold
    :param mat: A 3x3 matrix
    :return: A quantity describing the distance of mat from the SO(3)
             manifold
    Computes the distance from mat to the SO(3) manifold using the following
    method:
    If det(mat) <= 0, return a large number.
    If det(mat) > 0, return norm(mat^T.mat - I).
    Example Input:
        mat = np.array([[ 1.0,  0.0,   0.0 ],
                        [ 0.0,  0.1,  -0.95],
                        [ 0.0,  1.0,   0.1 ]])
    Output:
        0.08835
    """
```

```
    if np.linalg.det(mat) > 0:
        return np.linalg.norm(np.dot(np.array(mat).T, mat) - np.eye(3))
    else:
        return 1e+9

def DistanceToSE3(mat):
    """Returns the Frobenius norm to describe the distance of mat from the
    SE(3) manifold
    :param mat: A 4x4 matrix
    :return: A quantity describing the distance of mat from the SE(3)
             manifold
    Computes the distance from mat to the SE(3) manifold using the following
    method:
    Compute the determinant of matR, the top 3x3 submatrix of mat.
    If det(matR) <= 0, return a large number.
    If det(matR) > 0, replace the top 3x3 submatrix of mat with matR^T.matR,
    and set the first three entries of the fourth column of mat to zero. Then
    return norm(mat - I).
    Example Input:
        mat = np.array([[ 1.0,  0.0,   0.0,   1.2 ],
                        [ 0.0,  0.1,  -0.95,  1.5 ],
                        [ 0.0,  1.0,   0.1,  -0.9 ],
                        [ 0.0,  0.0,   0.1,   0.98 ]])
    Output:
        0.134931
    """
    matR = np.array(mat)[0: 3, 0: 3]
    if np.linalg.det(matR) > 0:
        return np.linalg.norm(np.r_[np.c_[np.dot(np.transpose(matR), matR),
                                          np.zeros((3, 1))],
                                    [np.array(mat)[3, :]]] - np.eye(4))
    else:
        return 1e+9

def TestIfSO3(mat):
    """Returns true if mat is close to or on the manifold SO(3)
    :param mat: A 3x3 matrix
    :return: True if mat is very close to or in SO(3), false otherwise
    Computes the distance d from mat to the SO(3) manifold using the
    following method:
    If det(mat) <= 0, d = a large number.
    If det(mat) > 0, d = norm(mat^T.mat - I).
    If d is close to zero, return true. Otherwise, return false.
    Example Input:
        mat = np.array([[1.0, 0.0,  0.0 ],
                        [0.0, 0.1, -0.95],
                        [0.0, 1.0,  0.1 ]])
    Output:
        False
    """
    return abs(DistanceToSO3(mat)) < 1e-3

def TestIfSE3(mat):
    """Returns true if mat is close to or on the manifold SE(3)
```

```
    :param mat: A 4x4 matrix
    :return: True if mat is very close to or in SE(3), false otherwise
    Computes the distance d from mat to the SE(3) manifold using the
    following method:
    Compute the determinant of the top 3x3 submatrix of mat.
    If det(mat) <= 0, d = a large number.
    If det(mat) > 0, replace the top 3x3 submatrix of mat with mat^T.mat, and
    set the first three entries of the fourth column of mat to zero.
    Then d = norm(T - I).
    If d is close to zero, return true. Otherwise, return false.
    Example Input:
        mat = np.array([[1.0, 0.0,   0.0,  1.2],
                        [0.0, 0.1, -0.95,  1.5],
                        [0.0, 1.0,   0.1, -0.9],
                        [0.0, 0.0,   0.1, 0.98]])
    Output:
        False
    """
    return abs(DistanceToSE3(mat)) < 1e-3

'''
*** CHAPTER 4: FORWARD KINEMATICS ***
'''

def FKinBody(M, Blist, thetalist):
    """Computes forward kinematics in the body frame for an open chain robot
    :param M: The home configuration (position and orientation) of the end-
          effector
    :param Blist: The joint screw axes in the end-effector frame when the
              manipulator is at the home position, in the format of a
              matrix with axes as the columns
    :param thetalist: A list of joint coordinates
    :return: A homogeneous transformation matrix representing the end-
         effector frame when the joints are at the specified coordinates
         (i.t.o Body Frame)
    Example Input:
        M = np.array([[-1, 0,  0, 0],
                      [ 0, 1,  0, 6],
                      [ 0, 0, -1, 2],
                      [ 0, 0,  0, 1]])
        Blist = np.array([[0, 0, -1, 2, 0,   0],
                          [0, 0,  0, 0, 1,   0],
                          [0, 0,  1, 0, 0, 0.1]]).T
        thetalist = np.array([np.pi / 2.0, 3, np.pi])
    Output:
        np.array([[0, 1,  0,         -5],
                  [1, 0,  0,          4],
                  [0, 0, -1, 1.68584073],
                  [0, 0,  0,          1]])
    """
    T = np.array(M)
    for i in range(len(thetalist)):
        T = np.dot(T, MatrixExp6(VecTose3(np.array(Blist)[:, i] \
                                          * thetalist[i])))
```

```
    return T

def FKinSpace(M, Slist, thetalist):
    """Computes forward kinematics in the space frame for an open chain robot
    :param M: The home configuration (position and orientation) of the end-
              effector
    :param Slist: The joint screw axes in the space frame when the
                  manipulator is at the home position, in the format of a
                  matrix with axes as the columns
    :param thetalist: A list of joint coordinates
    :return: A homogeneous transformation matrix representing the end-
             effector frame when the joints are at the specified coordinates
             (i.t.o Space Frame)
    Example Input:
        M = np.array([[-1, 0,  0, 0],
                      [ 0, 1,  0, 6],
                      [ 0, 0, -1, 2],
                      [ 0, 0,  0, 1]])
        Slist = np.array([[0, 0,  1,  4, 0,    0],
                          [0, 0,  0,  0, 1,    0],
                          [0, 0, -1, -6, 0, -0.1]]).T
        thetalist = np.array([np.pi / 2.0, 3, np.pi])
    Output:
        np.array([[0, 1,  0,         -5],
                  [1, 0,  0,          4],
                  [0, 0, -1, 1.68584073],
                  [0, 0,  0,          1]])
    """
    T = np.array(M)
    for i in range(len(thetalist) - 1, -1, -1):
        T = np.dot(MatrixExp6(VecTose3(np.array(Slist)[:, i] \
                                       * thetalist[i])), T)
    return T

'''
*** CHAPTER 5: VELOCITY KINEMATICS AND STATICS***
'''

def JacobianBody(Blist, thetalist):
    """Computes the body Jacobian for an open chain robot
    :param Blist: The joint screw axes in the end-effector frame when the
                  manipulator is at the home position, in the format of a
                  matrix with axes as the columns
    :param thetalist: A list of joint coordinates
    :return: The body Jacobian corresponding to the inputs (6xn real
             numbers)
    Example Input:
        Blist = np.array([[0, 0, 1,   0, 0.2, 0.2],
                          [1, 0, 0,   2,   0,   3],
                          [0, 1, 0,   0,   2,   1],
                          [1, 0, 0, 0.2, 0.3, 0.4]]).T
        thetalist = np.array([0.2, 1.1, 0.1, 1.2])
    Output:
        np.array([[-0.04528405, 0.99500417,           0,   1]
```

```
            [ 0.74359313, 0.09304865,  0.36235775,   0]
            [-0.66709716, 0.03617541, -0.93203909,   0]
            [ 2.32586047,    1.66809,  0.56410831, 0.2]
            [-1.44321167, 2.94561275,  1.43306521, 0.3]
            [-2.06639565, 1.82881722, -1.58868628, 0.4]])
    """
    Jb = np.array(Blist).copy().astype(np.float)
    T = np.eye(4)
    for i in range(len(thetalist) - 2, -1, -1):
        T = np.dot(T,MatrixExp6(VecTose3(np.array(Blist)[:, i + 1] \
                                * -thetalist[i + 1])))
        Jb[:, i] = np.dot(Adjoint(T), np.array(Blist)[:, i])
    return Jb

def JacobianSpace(Slist, thetalist):
    """Computes the space Jacobian for an open chain robot
    :param Slist: The joint screw axes in the space frame when the
            manipulator is at the home position, in the format of a
            matrix with axes as the columns
    :param thetalist: A list of joint coordinates
    :return: The space Jacobian corresponding to the inputs (6xn real
        numbers)
    Example Input:
        Slist = np.array([[0, 0, 1,   0, 0.2, 0.2],
                          [1, 0, 0,   2,   0,   3],
                          [0, 1, 0,   0,   2,   1],
                          [1, 0, 0, 0.2, 0.3, 0.4]]).T
        thetalist = np.array([0.2, 1.1, 0.1, 1.2])
    Output:
        np.array([[  0, 0.98006658, -0.09011564,  0.95749426]
            [  0, 0.19866933,   0.4445544,  0.28487557]
            [  1,          0,  0.89120736, -0.04528405]
            [  0, 1.95218638, -2.21635216, -0.51161537]
            [0.2, 0.43654132, -2.43712573,  2.77535713]
            [0.2, 2.96026613,  3.23573065,  2.22512443]])
    """
    Js = np.array(Slist).copy().astype(np.float)
    T = np.eye(4)
    for i in range(1, len(thetalist)):
        T = np.dot(T, MatrixExp6(VecTose3(np.array(Slist)[:, i - 1] \
                                * thetalist[i - 1])))
        Js[:, i] = np.dot(Adjoint(T), np.array(Slist)[:, i])
    return Js

'''
*** CHAPTER 6: INVERSE KINEMATICS ***
'''

def IKinBody(Blist, M, T, thetalist0, eomg, ev):
    """Computes inverse kinematics in the body frame for an open chain robot
    :param Blist: The joint screw axes in the end-effector frame when the
            manipulator is at the home position, in the format of a
            matrix with axes as the columns
    :param M: The home configuration of the end-effector
```

```
:param T: The desired end-effector configuration Tsd
:param thetalist0: An initial guess of joint angles that are close to
                   satisfying Tsd
:param eomg: A small positive tolerance on the end-effector orientation
             error. The returned joint angles must give an end-effector
             orientation error less than eomg
:param ev: A small positive tolerance on the end-effector linear position
           error. The returned joint angles must give an end-effector
           position error less than ev
:return thetalist: Joint angles that achieve T within the specified
                   tolerances,
:return success: A logical value where TRUE means that the function found
                 a solution and FALSE means that it ran through the set
                 number of maximum iterations without finding a solution
                 within the tolerances eomg and ev.
Uses an iterative Newton-Raphson root-finding method.
The maximum number of iterations before the algorithm is terminated has
been hardcoded in as a variable called maxiterations. It is set to 20 at
the start of the function, but can be changed if needed.
Example Input:
    Blist = np.array([[0, 0, -1, 2, 0,   0],
                      [0, 0,  0, 0, 1,   0],
                      [0, 0,  1, 0, 0, 0.1]]).T
    M = np.array([[-1, 0,  0, 0],
                  [ 0, 1,  0, 6],
                  [ 0, 0, -1, 2],
                  [ 0, 0,  0, 1]])
    T = np.array([[0, 1,  0,     -5],
                  [1, 0,  0,      4],
                  [0, 0, -1, 1.6858],
                  [0, 0,  0,      1]])
    thetalist0 = np.array([1.5, 2.5, 3])
    eomg = 0.01
    ev = 0.001
Output:
    (np.array([1.57073819, 2.999667, 3.14153913]), True)
"""
thetalist = np.array(thetalist0).copy()
i = 0
maxiterations = 20
Vb = se3ToVec(MatrixLog6(np.dot(TransInv(FKinBody(M, Blist, \
                                                  thetalist)), T)))
err = np.linalg.norm([Vb[0], Vb[1], Vb[2]]) > eomg \
      or np.linalg.norm([Vb[3], Vb[4], Vb[5]]) > ev
while err and i < maxiterations:
    thetalist = thetalist \
                + np.dot(np.linalg.pinv(JacobianBody(Blist, \
                                                     thetalist)), Vb)
    i = i + 1
    Vb \
    = se3ToVec(MatrixLog6(np.dot(TransInv(FKinBody(M, Blist, \
                                                   thetalist)), T)))
    err = np.linalg.norm([Vb[0], Vb[1], Vb[2]]) > eomg \
          or np.linalg.norm([Vb[3], Vb[4], Vb[5]]) > ev
```

```
    return (thetalist, not err)

def IKinSpace(Slist, M, T, thetalist0, eomg, ev):
    """Computes inverse kinematics in the space frame for an open chain robot
    :param Slist: The joint screw axes in the space frame when the
                  manipulator is at the home position, in the format of a
                  matrix with axes as the columns
    :param M: The home configuration of the end-effector
    :param T: The desired end-effector configuration Tsd
    :param thetalist0: An initial guess of joint angles that are close to
                       satisfying Tsd
    :param eomg: A small positive tolerance on the end-effector orientation
                 error. The returned joint angles must give an end-effector
                 orientation error less than eomg
    :param ev: A small positive tolerance on the end-effector linear position
               error. The returned joint angles must give an end-effector
               position error less than ev
    :return thetalist: Joint angles that achieve T within the specified
                       tolerances,
    :return success: A logical value where TRUE means that the function found
                     a solution and FALSE means that it ran through the set
                     number of maximum iterations without finding a solution
                     within the tolerances eomg and ev.
    Uses an iterative Newton-Raphson root-finding method.
    The maximum number of iterations before the algorithm is terminated has
    been hardcoded in as a variable called maxiterations. It is set to 20 at
    the start of the function, but can be changed if needed.
    Example Input:
        Slist = np.array([[0, 0,  1,  4, 0,    0],
                          [0, 0,  0,  0, 1,    0],
                          [0, 0, -1, -6, 0, -0.1]]).T
        M = np.array([[-1, 0,  0, 0],
                      [ 0, 1,  0, 6],
                      [ 0, 0, -1, 2],
                      [ 0, 0,  0, 1]])
        T = np.array([[0, 1,  0,     -5],
                      [1, 0,  0,      4],
                      [0, 0, -1, 1.6858],
                      [0, 0,  0,      1]])
        thetalist0 = np.array([1.5, 2.5, 3])
        eomg = 0.01
        ev = 0.001
    Output:
        (np.array([ 1.57073783,  2.99966384,  3.1415342 ]), True)
    """
    thetalist = np.array(thetalist0).copy()
    i = 0
    maxiterations = 20
    Tsb = FKinSpace(M,Slist, thetalist)
    Vs = np.dot(Adjoint(Tsb), \
                se3ToVec(MatrixLog6(np.dot(TransInv(Tsb), T))))
    err = np.linalg.norm([Vs[0], Vs[1], Vs[2]]) > eomg \
          or np.linalg.norm([Vs[3], Vs[4], Vs[5]]) > ev
    while err and i < maxiterations:
```

```
        thetalist = thetalist \
                  + np.dot(np.linalg.pinv(JacobianSpace(Slist, \
                                                        thetalist)), Vs)
        i = i + 1
        Tsb = FKinSpace(M, Slist, thetalist)
        Vs = np.dot(Adjoint(Tsb), \
                    se3ToVec(MatrixLog6(np.dot(TransInv(Tsb), T))))
        err = np.linalg.norm([Vs[0], Vs[1], Vs[2]]) > eomg \
              or np.linalg.norm([Vs[3], Vs[4], Vs[5]]) > ev
    return (thetalist, not err)

'''
*** CHAPTER 8: DYNAMICS OF OPEN CHAINS ***
'''

def ad(V):
    """Calculate the 6x6 matrix [adV] of the given 6-vector
    :param V: A 6-vector spatial velocity
    :return: The corresponding 6x6 matrix [adV]
    Used to calculate the Lie bracket [V1, V2] = [adV1]V2
    Example Input:
        V = np.array([1, 2, 3, 4, 5, 6])
    Output:
        np.array([[ 0, -3,  2,  0,  0,  0],
                  [ 3,  0, -1,  0,  0,  0],
                  [-2,  1,  0,  0,  0,  0],
                  [ 0, -6,  5,  0, -3,  2],
                  [ 6,  0, -4,  3,  0, -1],
                  [-5,  4,  0, -2,  1,  0]])
    """
    omgmat = VecToso3([V[0], V[1], V[2]])
    return np.r_[np.c_[omgmat, np.zeros((3, 3))],
                 np.c_[VecToso3([V[3], V[4], V[5]]), omgmat]]

def InverseDynamics(thetalist, dthetalist, ddthetalist, g, Ftip, Mlist, \
                    Glist, Slist):
    """Computes inverse dynamics in the space frame for an open chain robot
    :param thetalist: n-vector of joint variables
    :param dthetalist: n-vector of joint rates
    :param ddthetalist: n-vector of joint accelerations
    :param g: Gravity vector g
    :param Ftip: Spatial force applied by the end-effector expressed in frame
                 {n+1}
    :param Mlist: List of link frames {i} relative to {i-1} at the home
                  position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :return: The n-vector of required joint forces/torques
    This function uses forward-backward Newton-Euler iterations to solve the
    equation:
    taulist = Mlist(thetalist)ddthetalist + c(thetalist,dthetalist) \
              + g(thetalist) + Jtr(thetalist)Ftip
    Example Input (3 Link Robot):
```

```
    thetalist = np.array([0.1, 0.1, 0.1])
    dthetalist = np.array([0.1, 0.2, 0.3])
    ddthetalist = np.array([2, 1.5, 1])
    g = np.array([0, 0, -9.8])
    Ftip = np.array([1, 1, 1, 1, 1, 1])
    M01 = np.array([[1, 0, 0,        0],
                    [0, 1, 0,        0],
                    [0, 0, 1, 0.089159],
                    [0, 0, 0,        1]])
    M12 = np.array([[ 0, 0, 1,    0.28],
                    [ 0, 1, 0, 0.13585],
                    [-1, 0, 0,       0],
                    [ 0, 0, 0,       1]])
    M23 = np.array([[1, 0, 0,       0],
                    [0, 1, 0, -0.1197],
                    [0, 0, 1,   0.395],
                    [0, 0, 0,       1]])
    M34 = np.array([[1, 0, 0,       0],
                    [0, 1, 0,       0],
                    [0, 0, 1, 0.14225],
                    [0, 0, 0,       1]])
    G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
    G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
    G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
    Glist = np.array([G1, G2, G3])
    Mlist = np.array([M01, M12, M23, M34])
    Slist = np.array([[1, 0, 1,      0, 1,     0],
                      [0, 1, 0, -0.089, 0,     0],
                      [0, 1, 0, -0.089, 0, 0.425]]).T
Output:
    np.array([74.69616155, -33.06766016, -3.23057314])
"""
n = len(thetalist)
Mi = np.eye(4)
Ai = np.zeros((6, n))
AdTi = [[None]] * (n + 1)
Vi = np.zeros((6, n + 1))
Vdi = np.zeros((6, n + 1))
Vdi[:, 0] = np.r_[[0, 0, 0], -np.array(g)]
AdTi[n] = Adjoint(TransInv(Mlist[n]))
Fi = np.array(Ftip).copy()
taulist = np.zeros(n)
for i in range(n):
    Mi = np.dot(Mi,Mlist[i])
    Ai[:, i] = np.dot(Adjoint(TransInv(Mi)), np.array(Slist)[:, i])
    AdTi[i] = Adjoint(np.dot(MatrixExp6(VecTose3(Ai[:, i] * \
                                        -thetalist[i])), \
                             TransInv(Mlist[i])))
    Vi[:, i + 1] = np.dot(AdTi[i], Vi[:,i]) + Ai[:, i] * dthetalist[i]
    Vdi[:, i + 1] = np.dot(AdTi[i], Vdi[:, i]) \
                   + Ai[:, i] * ddthetalist[i] \
                   + np.dot(ad(Vi[:, i + 1]), Ai[:, i]) * dthetalist[i]
for i in range (n - 1, -1, -1):
    Fi = np.dot(np.array(AdTi[i + 1]).T, Fi) \
```

```
            + np.dot(np.array(Glist[i]), Vdi[:, i + 1]) \
            - np.dot(np.array(ad(Vi[:, i + 1])).T, \
                     np.dot(np.array(Glist[i]), Vi[:, i + 1]))
        taulist[i] = np.dot(np.array(Fi).T, Ai[:, i])
    return taulist

def MassMatrix(thetalist, Mlist, Glist, Slist):
    """Computes the mass matrix of an open chain robot based on the given
    configuration
    :param thetalist: A list of joint variables
    :param Mlist: List of link frames i relative to i-1 at the home position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :return: The numerical inertia matrix M(thetalist) of an n-joint serial
             chain at the given configuration thetalist
    This function calls InverseDynamics n times, each time passing a
    ddthetalist vector with a single element equal to one and all other
    inputs set to zero.
    Each call of InverseDynamics generates a single column, and these columns
    are assembled to create the inertia matrix.
    Example Input (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
    Output:
        np.array([[ 2.25433380e+01, -3.07146754e-01, -7.18426391e-03]
                  [-3.07146754e-01,  1.96850717e+00,  4.32157368e-01]
                  [-7.18426391e-03,  4.32157368e-01,  1.91630858e-01]])
    """
    n = len(thetalist)
    M = np.zeros((n, n))
```

```
    for i in range (n):
        ddthetalist = [0] * n
        ddthetalist[i] = 1
        M[:, i] = InverseDynamics(thetalist, [0] * n, ddthetalist, \
                                  [0, 0, 0], [0, 0, 0, 0, 0, 0], Mlist, \
                                  Glist, Slist)
    return M

def VelQuadraticForces(thetalist, dthetalist, Mlist, Glist, Slist):
    """Computes the Coriolis and centripetal terms in the inverse dynamics of
    an open chain robot
    :param thetalist: A list of joint variables,
    :param dthetalist: A list of joint rates,
    :param Mlist: List of link frames i relative to i-1 at the home position,
    :param Glist: Spatial inertia matrices Gi of the links,
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns.
    :return: The vector c(thetalist,dthetalist) of Coriolis and centripetal
             terms for a given thetalist and dthetalist.
    This function calls InverseDynamics with g = 0, Ftip = 0, and
    ddthetalist = 0.
    Example Input (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        dthetalist = np.array([0.1, 0.2, 0.3])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
    Output:
        np.array([0.26453118, -0.05505157, -0.00689132])
    """
    return InverseDynamics(thetalist, dthetalist, [0] * len(thetalist), \
                           [0, 0, 0], [0, 0, 0, 0, 0, 0], Mlist, Glist, \
                           Slist)
```

```
def GravityForces(thetalist, g, Mlist, Glist, Slist):
    """Computes the joint forces/torques an open chain robot requires to
    overcome gravity at its configuration
    :param thetalist: A list of joint variables
    :param g: 3-vector for gravitational acceleration
    :param Mlist: List of link frames i relative to i-1 at the home position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :return grav: The joint forces/torques required to overcome gravity at
                  thetalist
    This function calls InverseDynamics with Ftip = 0, dthetalist = 0, and
    ddthetalist = 0.
    Example Inputs (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        g = np.array([0, 0, -9.8])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
    Output:
        np.array([28.40331262, -37.64094817, -5.4415892])
    """
    n = len(thetalist)
    return InverseDynamics(thetalist, [0] * n, [0] * n, g, \
                           [0, 0, 0, 0, 0, 0], Mlist, Glist, Slist)

def EndEffectorForces(thetalist, Ftip, Mlist, Glist, Slist):
    """Computes the joint forces/torques an open chain robot requires only to
    create the end-effector force Ftip
    :param thetalist: A list of joint variables
    :param Ftip: Spatial force applied by the end-effector expressed in frame
                 {n+1}
```

```
    :param Mlist: List of link frames i relative to i-1 at the home position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :return: The joint forces and torques required only to create the
             end-effector force Ftip
    This function calls InverseDynamics with g = 0, dthetalist = 0, and
    ddthetalist = 0.
    Example Input (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        Ftip = np.array([1, 1, 1, 1, 1, 1])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
    Output:
        np.array([1.40954608, 1.85771497, 1.392409])
    """
    n = len(thetalist)
    return InverseDynamics(thetalist, [0] * n, [0] * n, [0, 0, 0], Ftip, \
                           Mlist, Glist, Slist)

def ForwardDynamics(thetalist, dthetalist, taulist, g, Ftip, Mlist, \
                    Glist, Slist):
    """Computes forward dynamics in the space frame for an open chain robot
    :param thetalist: A list of joint variables
    :param dthetalist: A list of joint rates
    :param taulist: An n-vector of joint forces/torques
    :param g: Gravity vector g
    :param Ftip: Spatial force applied by the end-effector expressed in frame
                 {n+1}
    :param Mlist: List of link frames i relative to i-1 at the home position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
```

```
                 of a matrix with axes as the columns
    :return: The resulting joint accelerations
    This function computes ddthetalist by solving:
    Mlist(thetalist) * ddthetalist = taulist - c(thetalist,dthetalist) \
                                     - g(thetalist) - Jtr(thetalist) * Ftip
    Example Input (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        dthetalist = np.array([0.1, 0.2, 0.3])
        taulist = np.array([0.5, 0.6, 0.7])
        g = np.array([0, 0, -9.8])
        Ftip = np.array([1, 1, 1, 1, 1, 1])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
    Output:
        np.array([-0.97392907, 25.58466784, -32.91499212])
    """
    return np.dot(np.linalg.inv(MassMatrix(thetalist, Mlist, Glist, \
                                           Slist)), \
                  np.array(taulist) \
                  - VelQuadraticForces(thetalist, dthetalist, Mlist, \
                                       Glist, Slist) \
                  - GravityForces(thetalist, g, Mlist, Glist, Slist) \
                  - EndEffectorForces(thetalist, Ftip, Mlist, Glist, \
                                      Slist))

def EulerStep(thetalist, dthetalist, ddthetalist, dt):
    """Compute the joint angles and velocities at the next timestep using            from here
    first order Euler integration
    :param thetalist: n-vector of joint variables
    :param dthetalist: n-vector of joint rates
    :param ddthetalist: n-vector of joint accelerations
    :param dt: The timestep delta t
```

```
    :return thetalistNext: Vector of joint variables after dt from first
                           order Euler integration
    :return dthetalistNext: Vector of joint rates after dt from first order
                            Euler integration
    Example Inputs (3 Link Robot):
        thetalist = np.array([0.1, 0.1, 0.1])
        dthetalist = np.array([0.1, 0.2, 0.3])
        ddthetalist = np.array([2, 1.5, 1])
        dt = 0.1
    Output:
        thetalistNext:
        array([ 0.11,  0.12,  0.13])
        dthetalistNext:
        array([ 0.3 ,  0.35,  0.4 ])
    """
    return thetalist + dt * np.array(dthetalist), \
           dthetalist + dt * np.array(ddthetalist)

def InverseDynamicsTrajectory(thetamat, dthetamat, ddthetamat, g, \
                              Ftipmat, Mlist, Glist, Slist):
    """Calculates the joint forces/torques required to move the serial chain
    along the given trajectory using inverse dynamics
    :param thetamat: An N x n matrix of robot joint variables
    :param dthetamat: An N x n matrix of robot joint velocities
    :param ddthetamat: An N x n matrix of robot joint accelerations
    :param g: Gravity vector g
    :param Ftipmat: An N x 6 matrix of spatial forces applied by the end-
                    effector (If there are no tip forces the user should
                    input a zero and a zero matrix will be used)
    :param Mlist: List of link frames i relative to i-1 at the home position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :return: The N x n matrix of joint forces/torques for the specified
             trajectory, where each of the N rows is the vector of joint
             forces/torques at each time step
    Example Inputs (3 Link Robot):
        from __future__ import print_function
        import numpy as np
        import modern_robotics as mr
        # Create a trajectory to follow using functions from Chapter 9
        thetastart =  np.array([0, 0, 0])
        thetaend =  np.array([np.pi / 2, np.pi / 2, np.pi / 2])
        Tf = 3
        N= 1000
        method = 5
        traj = mr.JointTrajectory(thetastart, thetaend, Tf, N, method)
        thetamat = np.array(traj).copy()
        dthetamat = np.zeros((1000,3 ))
        ddthetamat = np.zeros((1000, 3))
        dt = Tf / (N - 1.0)
        for i in range(np.array(traj).shape[0] - 1):
            dthetamat[i + 1, :] = (thetamat[i + 1, :] - thetamat[i, :]) / dt
            ddthetamat[i + 1, :] \
```

```
        = (dthetamat[i + 1, :] - dthetamat[i, :]) / dt
    # Initialize robot description (Example with 3 links)
    g =  np.array([0, 0, -9.8])
    Ftipmat = np.ones((N, 6))
    M01 = np.array([[1, 0, 0,        0],
                    [0, 1, 0,        0],
                    [0, 0, 1, 0.089159],
                    [0, 0, 0,        1]])
    M12 = np.array([[ 0, 0, 1,    0.28],
                    [ 0, 1, 0, 0.13585],
                    [-1, 0, 0,       0],
                    [ 0, 0, 0,       1]])
    M23 = np.array([[1, 0, 0,       0],
                    [0, 1, 0, -0.1197],
                    [0, 0, 1,   0.395],
                    [0, 0, 0,       1]])
    M34 = np.array([[1, 0, 0,       0],
                    [0, 1, 0,       0],
                    [0, 0, 1, 0.14225],
                    [0, 0, 0,       1]])
    G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
    G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
    G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
    Glist = np.array([G1, G2, G3])
    Mlist = np.array([M01, M12, M23, M34])
    Slist = np.array([[1, 0, 1,      0, 1,     0],
                      [0, 1, 0, -0.089, 0,     0],
                      [0, 1, 0, -0.089, 0, 0.425]]).T
    taumat \
    = mr.InverseDynamicsTrajectory(thetamat, dthetamat, ddthetamat, g, \
                                   Ftipmat, Mlist, Glist, Slist)
# Output using matplotlib to plot the joint forces/torques
    Tau1 = taumat[:, 0]
    Tau2 = taumat[:, 1]
    Tau3 = taumat[:, 2]
    timestamp = np.linspace(0, Tf, N)
    try:
        import matplotlib.pyplot as plt
    except:
        print('The result will not be plotted due to a lack of package matplotlib')
    else:
        plt.plot(timestamp, Tau1, label = "Tau1")
        plt.plot(timestamp, Tau2, label = "Tau2")
        plt.plot(timestamp, Tau3, label = "Tau3")
        plt.ylim (-40, 120)
        plt.legend(loc = 'lower right')
        plt.xlabel("Time")
        plt.ylabel("Torque")
        plt.title("Plot of Torque Trajectories")
        plt.show()
"""
thetamat = np.array(thetamat).T
dthetamat = np.array(dthetamat).T
ddthetamat = np.array(ddthetamat).T
```

```
    Ftipmat = np.array(Ftipmat).T
    taumat = np.array(thetamat).copy()
    for i in range(np.array(thetamat).shape[1]):
        taumat[:, i] \
        = InverseDynamics(thetamat[:, i], dthetamat[:, i], \
                          ddthetamat[:, i], g, Ftipmat[:, i], Mlist, \
                          Glist, Slist)
    taumat = np.array(taumat).T
    return taumat

def ForwardDynamicsTrajectory(thetalist, dthetalist, taumat, g, Ftipmat, \
                              Mlist, Glist, Slist, dt, intRes):
    """Simulates the motion of a serial chain given an open-loop history of
    joint forces/torques
    :param thetalist: n-vector of initial joint variables
    :param dthetalist: n-vector of initial joint rates
    :param taumat: An N x n matrix of joint forces/torques, where each row is
                   the joint effort at any time step
    :param g: Gravity vector g
    :param Ftipmat: An N x 6 matrix of spatial forces applied by the end-
                    effector (If there are no tip forces the user should
                    input a zero and a zero matrix will be used)
    :param Mlist: List of link frames {i} relative to {i-1} at the home
                  position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :param dt: The timestep between consecutive joint forces/torques
    :param intRes: Integration resolution is the number of times integration
                   (Euler) takes places between each time step. Must be an
                   integer value greater than or equal to 1
    :return thetamat: The N x n matrix of robot joint angles resulting from
                      the specified joint forces/torques
    :return dthetamat: The N x n matrix of robot joint velocities
    This function calls a numerical integration procedure that uses
    ForwardDynamics.
    Example Inputs (3 Link Robot):
        from __future__ import print_function
        import numpy as np
        import modern_robotics as mr
        thetalist = np.array([0.1, 0.1, 0.1])
        dthetalist = np.array([0.1, 0.2, 0.3])
        taumat = np.array([[3.63, -6.58, -5.57], [3.74, -5.55,  -5.5],
                           [4.31, -0.68, -5.19], [5.18,  5.63, -4.31],
                           [5.85,  8.17, -2.59], [5.78,  2.79,  -1.7],
                           [4.99,  -5.3, -1.19], [4.08, -9.41,  0.07],
                           [3.56, -10.1,  0.97], [3.49, -9.41,  1.23]])
        # Initialize robot description (Example with 3 links)
        g = np.array([0, 0, -9.8])
        Ftipmat = np.ones((np.array(taumat).shape[0], 6))
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
```

```
    M12 = np.array([[ 0, 0, 1,    0.28],
                    [ 0, 1, 0, 0.13585],
                    [-1, 0, 0,       0],
                    [ 0, 0, 0,       1]])
    M23 = np.array([[1, 0, 0,       0],
                    [0, 1, 0, -0.1197],
                    [0, 0, 1,   0.395],
                    [0, 0, 0,       1]])
    M34 = np.array([[1, 0, 0,       0],
                    [0, 1, 0,       0],
                    [0, 0, 1, 0.14225],
                    [0, 0, 0,       1]])
    G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
    G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
    G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
    Glist = np.array([G1, G2, G3])
    Mlist = np.array([M01, M12, M23, M34])
    Slist = np.array([[1, 0, 1,      0, 1,     0],
                      [0, 1, 0, -0.089, 0,     0],
                      [0, 1, 0, -0.089, 0, 0.425]]).T
    dt = 0.1
    intRes = 8
    thetamat,dthetamat \
    = mr.ForwardDynamicsTrajectory(thetalist, dthetalist, taumat, g, \
                                   Ftipmat, Mlist, Glist, Slist, dt, \
                                   intRes)
# Output using matplotlib to plot the joint angle/velocities
    theta1 = thetamat[:, 0]
    theta2 = thetamat[:, 1]
    theta3 = thetamat[:, 2]
    dtheta1 = dthetamat[:, 0]
    dtheta2 = dthetamat[:, 1]
    dtheta3 = dthetamat[:, 2]
    N = np.array(taumat).shape[0]
    Tf = np.array(taumat).shape[0] * dt
        timestamp = np.linspace(0, Tf, N)
        try:
            import matplotlib.pyplot as plt
    except:
        print('The result will not be plotted due to a lack of package matplotlib')
    else:
        plt.plot(timestamp, theta1, label = "Theta1")
        plt.plot(timestamp, theta2, label = "Theta2")
        plt.plot(timestamp, theta3, label = "Theta3")
        plt.plot(timestamp, dtheta1, label = "DTheta1")
        plt.plot(timestamp, dtheta2, label = "DTheta2")
        plt.plot(timestamp, dtheta3, label = "DTheta3")
        plt.ylim (-12, 10)
        plt.legend(loc = 'lower right')
        plt.xlabel("Time")
        plt.ylabel("Joint Angles/Velocities")
        plt.title("Plot of Joint Angles and Joint Velocities")
        plt.show()
"""
```

```
    taumat = np.array(taumat).T
    Ftipmat = np.array(Ftipmat).T
    thetamat = taumat.copy().astype(np.float)
    thetamat[:, 0] = thetalist
    dthetamat = taumat.copy().astype(np.float)
    dthetamat[:, 0] = dthetalist
    for i in range(np.array(taumat).shape[1] - 1):
        for j in range(intRes):
            ddthetalist \
            = ForwardDynamics(thetalist, dthetalist, taumat[:, i], g, \
                              Ftipmat[:, i], Mlist, Glist, Slist)
            thetalist,dthetalist = EulerStep(thetalist, dthetalist, \
                                             ddthetalist, 1.0 * dt / intRes)
        thetamat[:, i + 1] = thetalist
        dthetamat[:, i + 1] = dthetalist
    thetamat = np.array(thetamat).T
    dthetamat = np.array(dthetamat).T
    return thetamat, dthetamat

'''
*** CHAPTER 9: TRAJECTORY GENERATION ***
'''

def CubicTimeScaling(Tf, t):
    """Computes s(t) for a cubic time scaling
    :param Tf: Total time of the motion in seconds from rest to rest
    :param t: The current time t satisfying 0 < t < Tf
    :return: The path parameter s(t) corresponding to a third-order
             polynomial motion that begins and ends at zero velocity
    Example Input:
        Tf = 2
        t = 0.6
    Output:
        0.216
    """
    return 3 * (1.0 * t / Tf) ** 2 - 2 * (1.0 * t / Tf) ** 3

def QuinticTimeScaling(Tf, t):
    """Computes s(t) for a quintic time scaling
    :param Tf: Total time of the motion in seconds from rest to rest
    :param t: The current time t satisfying 0 < t < Tf
    :return: The path parameter s(t) corresponding to a fifth-order
             polynomial motion that begins and ends at zero velocity and zero
             acceleration
    Example Input:
        Tf = 2
        t = 0.6
    Output:
        0.16308
    """
    return 10 * (1.0 * t / Tf) ** 3 - 15 * (1.0 * t / Tf) ** 4 \
           + 6 * (1.0 * t / Tf) ** 5

def JointTrajectory(thetastart, thetaend, Tf, N, method):
```

```
    """Computes a straight-line trajectory in joint space
    :param thetastart: The initial joint variables
    :param thetaend: The final joint variables
    :param Tf: Total time of the motion in seconds from rest to rest
    :param N: The number of points N > 1 (Start and stop) in the discrete
              representation of the trajectory
    :param method: The time-scaling method, where 3 indicates cubic (third-
                   order polynomial) time scaling and 5 indicates quintic
                   (fifth-order polynomial) time scaling
    :return: A trajectory as an N x n matrix, where each row is an n-vector
             of joint variables at an instant in time. The first row is
             thetastart and the Nth row is thetaend . The elapsed time
             between each row is Tf / (N - 1)
    Example Input:
        thetastart = np.array([1, 0, 0, 1, 1, 0.2, 0,1])
        thetaend = np.array([1.2, 0.5, 0.6, 1.1, 2, 2, 0.9, 1])
        Tf = 4
        N = 6
        method = 3
    Output:
        np.array([[     1,     0,      0,      1,     1,    0.2,      0, 1]
                  [1.0208, 0.052, 0.0624, 1.0104, 1.104, 0.3872, 0.0936, 1]
                  [1.0704, 0.176, 0.2112, 1.0352, 1.352, 0.8336, 0.3168, 1]
                  [1.1296, 0.324, 0.3888, 1.0648, 1.648, 1.3664, 0.5832, 1]
                  [1.1792, 0.448, 0.5376, 1.0896, 1.896, 1.8128, 0.8064, 1]
                  [   1.2,   0.5,    0.6,    1.1,     2,      2,    0.9, 1]])
    """
    N = int(N)
    timegap = Tf / (N - 1.0)
    traj = np.zeros((len(thetastart), N))
    for i in range(N):
        if method == 3:
            s = CubicTimeScaling(Tf, timegap * i)
        else:
            s = QuinticTimeScaling(Tf, timegap * i)
        traj[:, i] = s * np.array(thetaend) + (1 - s) * np.array(thetastart)
    traj = np.array(traj).T
    return traj

def ScrewTrajectory(Xstart, Xend, Tf, N, method):
    """Computes a trajectory as a list of N SE(3) matrices corresponding to
      the screw motion about a space screw axis
    :param Xstart: The initial end-effector configuration
    :param Xend: The final end-effector configuration
    :param Tf: Total time of the motion in seconds from rest to rest
    :param N: The number of points N > 1 (Start and stop) in the discrete
              representation of the trajectory
    :param method: The time-scaling method, where 3 indicates cubic (third-
                   order polynomial) time scaling and 5 indicates quintic
                   (fifth-order polynomial) time scaling
    :return: The discretized trajectory as a list of N matrices in SE(3)
             separated in time by Tf/(N-1). The first in the list is Xstart
             and the Nth is Xend
    Example Input:
```

```
        Xstart = np.array([[1, 0, 0, 1],
                           [0, 1, 0, 0],
                           [0, 0, 1, 1],
                           [0, 0, 0, 1]])
        Xend = np.array([[0, 0, 1, 0.1],
                         [1, 0, 0,   0],
                         [0, 1, 0, 4.1],
                         [0, 0, 0,   1]])
        Tf = 5
        N = 4
        method = 3
    Output:
        [np.array([[1, 0, 0, 1]
                   [0, 1, 0, 0]
                   [0, 0, 1, 1]
                   [0, 0, 0, 1]]),
         np.array([[0.904, -0.25, 0.346, 0.441]
                   [0.346, 0.904, -0.25, 0.529]
                   [-0.25, 0.346, 0.904, 1.601]
                   [    0,     0,     0,     1]]),
         np.array([[0.346, -0.25, 0.904, -0.117]
                   [0.904, 0.346, -0.25,  0.473]
                   [-0.25, 0.904, 0.346,  3.274]
                   [    0,     0,     0,      1]]),
         np.array([[0, 0, 1, 0.1]
                   [1, 0, 0,   0]
                   [0, 1, 0, 4.1]
                   [0, 0, 0,   1]])]
    """
    N = int(N)
    timegap = Tf / (N - 1.0)
    traj = [[None]] * N
    for i in range(N):
        if method == 3:
            s = CubicTimeScaling(Tf, timegap * i)
        else:
            s = QuinticTimeScaling(Tf, timegap * i)
        traj[i] \
        = np.dot(Xstart, MatrixExp6(MatrixLog6(np.dot(TransInv(Xstart), \
                                               Xend)) * s))
    return traj

def CartesianTrajectory(Xstart, Xend, Tf, N, method):
    """Computes a trajectory as a list of N SE(3) matrices corresponding to
    the origin of the end-effector frame following a straight line
    :param Xstart: The initial end-effector configuration
    :param Xend: The final end-effector configuration
    :param Tf: Total time of the motion in seconds from rest to rest
    :param N: The number of points N > 1 (Start and stop) in the discrete
              representation of the trajectory
    :param method: The time-scaling method, where 3 indicates cubic (third-
                   order polynomial) time scaling and 5 indicates quintic
                   (fifth-order polynomial) time scaling
    :return: The discretized trajectory as a list of N matrices in SE(3)
```

```
        separated in time by Tf/(N-1). The first in the list is Xstart
        and the Nth is Xend
    This function is similar to ScrewTrajectory, except the origin of the
    end-effector frame follows a straight line, decoupled from the rotational
    motion.
    Example Input:
        Xstart = np.array([[1, 0, 0, 1],
                           [0, 1, 0, 0],
                           [0, 0, 1, 1],
                           [0, 0, 0, 1]])
        Xend = np.array([[0, 0, 1, 0.1],
                         [1, 0, 0,   0],
                         [0, 1, 0, 4.1],
                         [0, 0, 0,   1]])
        Tf = 5
        N = 4
        method = 5
    Output:
        [np.array([[1, 0, 0, 1]
                   [0, 1, 0, 0]
                   [0, 0, 1, 1]
                   [0, 0, 0, 1]]),
         np.array([[ 0.937, -0.214,  0.277, 0.811]
                   [ 0.277,  0.937, -0.214,     0]
                   [-0.214,  0.277,  0.937, 1.651]
                   [     0,      0,      0,     1]]),
         np.array([[ 0.277, -0.214,  0.937, 0.289]
                   [ 0.937,  0.277, -0.214,     0]
                   [-0.214,  0.937,  0.277, 3.449]
                   [     0,      0,      0,     1]]),
         np.array([[0, 0, 1, 0.1]
                   [1, 0, 0,   0]
                   [0, 1, 0, 4.1]
                   [0, 0, 0,   1]])]
    """
    N = int(N)
    timegap = Tf / (N - 1.0)
    traj = [[None]] * N
    Rstart, pstart = TransToRp(Xstart)
    Rend, pend = TransToRp(Xend)
    for i in range(N):
        if method == 3:
            s = CubicTimeScaling(Tf, timegap * i)
        else:
            s = QuinticTimeScaling(Tf, timegap * i)
        traj[i] \
        = np.r_[np.c_[np.dot(Rstart, \
        MatrixExp3(MatrixLog3(np.dot(np.array(Rstart).T,Rend)) * s)), \
                   s * np.array(pend) + (1 - s) * np.array(pstart)], \
                   [[0, 0, 0, 1]]]
    return traj

'''
*** CHAPTER 11: ROBOT CONTROL ***
```

```
'''

def ComputedTorque(thetalist, dthetalist, eint, g, Mlist, Glist, Slist, \
                   thetalistd, dthetalistd, ddthetalistd, Kp, Ki, Kd):
    """Computes the joint control torques at a particular time instant
    :param thetalist: n-vector of joint variables
    :param dthetalist: n-vector of joint rates
    :param eint: n-vector of the time-integral of joint errors
    :param g: Gravity vector g
    :param Mlist: List of link frames {i} relative to {i-1} at the home
                  position
    :param Glist: Spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :param thetalistd: n-vector of reference joint variables
    :param dthetalistd: n-vector of reference joint velocities
    :param ddthetalistd: n-vector of reference joint accelerations
    :param Kp: The feedback proportional gain (identical for each joint)
    :param Ki: The feedback integral gain (identical for each joint)
    :param Kd: The feedback derivative gain (identical for each joint)
    :return: The vector of joint forces/torques computed by the feedback
             linearizing controller at the current instant
    Example Input:
        thetalist = np.array([0.1, 0.1, 0.1])
        dthetalist = np.array([0.1, 0.2, 0.3])
        eint = np.array([0.2, 0.2, 0.2])
        g = np.array([0, 0, -9.8])
        M01 = np.array([[1, 0, 0,        0],
                        [0, 1, 0,        0],
                        [0, 0, 1, 0.089159],
                        [0, 0, 0,        1]])
        M12 = np.array([[ 0, 0, 1,    0.28],
                        [ 0, 1, 0, 0.13585],
                        [-1, 0, 0,       0],
                        [ 0, 0, 0,       1]])
        M23 = np.array([[1, 0, 0,       0],
                        [0, 1, 0, -0.1197],
                        [0, 0, 1,   0.395],
                        [0, 0, 0,       1]])
        M34 = np.array([[1, 0, 0,       0],
                        [0, 1, 0,       0],
                        [0, 0, 1, 0.14225],
                        [0, 0, 0,       1]])
        G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
        G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
        G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
        Glist = np.array([G1, G2, G3])
        Mlist = np.array([M01, M12, M23, M34])
        Slist = np.array([[1, 0, 1,      0, 1,     0],
                          [0, 1, 0, -0.089, 0,     0],
                          [0, 1, 0, -0.089, 0, 0.425]]).T
        thetalistd = np.array([1.0, 1.0, 1.0])
        dthetalistd = np.array([2, 1.2, 2])
        ddthetalistd = np.array([0.1, 0.1, 0.1])
```

```
        Kp = 1.3
        Ki = 1.2
        Kd = 1.1
    Output:
        np.array([133.00525246, -29.94223324, -3.03276856])
    """
    e = np.subtract(thetalistd, thetalist)
    return np.dot(MassMatrix(thetalist, Mlist, Glist, Slist), \
                  Kp * e + Ki * (np.array(eint) + e) \
                  + Kd * np.subtract(dthetalistd, dthetalist)) \
           + InverseDynamics(thetalist, dthetalist, ddthetalistd, g, \
                             [0, 0, 0, 0, 0, 0], Mlist, Glist, Slist)

def SimulateControl(thetalist, dthetalist, g, Ftipmat, Mlist, Glist, \
                    Slist, thetamatd, dthetamatd, ddthetamatd, gtilde, \
                    Mtildelist, Gtildelist, Kp, Ki, Kd, dt, intRes):
    """Simulates the computed torque controller over a given desired
    trajectory
    :param thetalist: n-vector of initial joint variables
    :param dthetalist: n-vector of initial joint velocities
    :param g: Actual gravity vector g
    :param Ftipmat: An N x 6 matrix of spatial forces applied by the end-
                    effector (If there are no tip forces the user should
                    input a zero and a zero matrix will be used)
    :param Mlist: Actual list of link frames i relative to i-1 at the home
                  position
    :param Glist: Actual spatial inertia matrices Gi of the links
    :param Slist: Screw axes Si of the joints in a space frame, in the format
                  of a matrix with axes as the columns
    :param thetamatd: An Nxn matrix of desired joint variables from the
                      reference trajectory
    :param dthetamatd: An Nxn matrix of desired joint velocities
    :param ddthetamatd: An Nxn matrix of desired joint accelerations
    :param gtilde: The gravity vector based on the model of the actual robot
                   (actual values given above)
    :param Mtildelist: The link frame locations based on the model of the
                       actual robot (actual values given above)
    :param Gtildelist: The link spatial inertias based on the model of the
                       actual robot (actual values given above)
    :param Kp: The feedback proportional gain (identical for each joint)
    :param Ki: The feedback integral gain (identical for each joint)
    :param Kd: The feedback derivative gain (identical for each joint)
    :param dt: The timestep between points on the reference trajectory
    :param intRes: Integration resolution is the number of times integration
                   (Euler) takes places between each time step. Must be an
                   integer value greater than or equal to 1
    :return taumat: An Nxn matrix of the controllers commanded joint forces/
                    torques, where each row of n forces/torques corresponds
                    to a single time instant
    :return thetamat: An Nxn matrix of actual joint angles
    The end of this function plots all the actual and desired joint angles
    using matplotlib and random libraries.
    Example Input:
        from __future__ import print_function
```

```
import numpy as np
from modern_robotics import JointTrajectory
thetalist = np.array([0.1, 0.1, 0.1])
dthetalist = np.array([0.1, 0.2, 0.3])
# Initialize robot description (Example with 3 links)
g = np.array([0, 0, -9.8])
M01 = np.array([[1, 0, 0,        0],
                [0, 1, 0,        0],
                [0, 0, 1, 0.089159],
                [0, 0, 0,        1]])
M12 = np.array([[ 0, 0, 1,    0.28],
                [ 0, 1, 0, 0.13585],
                [-1, 0, 0,       0],
                [ 0, 0, 0,       1]])
M23 = np.array([[1, 0, 0,       0],
                [0, 1, 0, -0.1197],
                [0, 0, 1,   0.395],
                [0, 0, 0,       1]])
M34 = np.array([[1, 0, 0,       0],
                [0, 1, 0,       0],
                [0, 0, 1, 0.14225],
                [0, 0, 0,       1]])
G1 = np.diag([0.010267, 0.010267, 0.00666, 3.7, 3.7, 3.7])
G2 = np.diag([0.22689, 0.22689, 0.0151074, 8.393, 8.393, 8.393])
G3 = np.diag([0.0494433, 0.0494433, 0.004095, 2.275, 2.275, 2.275])
Glist = np.array([G1, G2, G3])
Mlist = np.array([M01, M12, M23, M34])
Slist = np.array([[1, 0, 1,      0, 1,     0],
                  [0, 1, 0, -0.089, 0,     0],
                  [0, 1, 0, -0.089, 0, 0.425]]).T
dt = 0.01
# Create a trajectory to follow
thetaend = np.array([np.pi / 2, np.pi, 1.5 * np.pi])
Tf = 1
N = int(1.0 * Tf / dt)
method = 5
traj = mr.JointTrajectory(thetalist, thetaend, Tf, N, method)
thetamatd = np.array(traj).copy()
dthetamatd = np.zeros((N, 3))
ddthetamatd = np.zeros((N, 3))
dt = Tf / (N - 1.0)
for i in range(np.array(traj).shape[0] - 1):
    dthetamatd[i + 1, :] \
    = (thetamatd[i + 1, :] - thetamatd[i, :]) / dt
    ddthetamatd[i + 1, :] \
    = (dthetamatd[i + 1, :] - dthetamatd[i, :]) / dt
# Possibly wrong robot description (Example with 3 links)
gtilde = np.array([0.8, 0.2, -8.8])
Mhat01 = np.array([[1, 0, 0,   0],
                   [0, 1, 0,   0],
                   [0, 0, 1, 0.1],
                   [0, 0, 0,   1]])
Mhat12 = np.array([[ 0, 0, 1, 0.3],
                   [ 0, 1, 0, 0.2],
```

```
                    [-1, 0, 0,   0],
                    [ 0, 0, 0,   1]])
    Mhat23 = np.array([[1, 0, 0,    0],
                    [0, 1, 0, -0.2],
                    [0, 0, 1,  0.4],
                    [0, 0, 0,    1]])
    Mhat34 = np.array([[1, 0, 0,   0],
                    [0, 1, 0,   0],
                    [0, 0, 1, 0.2],
                    [0, 0, 0,   1]])
    Ghat1 = np.diag([0.1, 0.1, 0.1, 4, 4, 4])
    Ghat2 = np.diag([0.3, 0.3, 0.1, 9, 9, 9])
    Ghat3 = np.diag([0.1, 0.1, 0.1, 3, 3, 3])
    Gtildelist = np.array([Ghat1, Ghat2, Ghat3])
    Mtildelist = np.array([Mhat01, Mhat12, Mhat23, Mhat34])
    Ftipmat = np.ones((np.array(traj).shape[0], 6))
    Kp = 20
    Ki = 10
    Kd = 18
    intRes = 8
    taumat,thetamat \
    = mr.SimulateControl(thetalist, dthetalist, g, Ftipmat, Mlist, \
                      Glist, Slist, thetamatd, dthetamatd, \
                      ddthetamatd, gtilde, Mtildelist, Gtildelist, \
                      Kp, Ki, Kd, dt, intRes)
"""
Ftipmat = np.array(Ftipmat).T
thetamatd = np.array(thetamatd).T
dthetamatd = np.array(dthetamatd).T
ddthetamatd = np.array(ddthetamatd).T
m,n = np.array(thetamatd).shape
thetacurrent = np.array(thetalist).copy()
dthetacurrent = np.array(dthetalist).copy()
eint = np.zeros((m,1)).reshape(m,)
taumat = np.zeros(np.array(thetamatd).shape)
thetamat = np.zeros(np.array(thetamatd).shape)
for i in range(n):
    taulist \
    = ComputedTorque(thetacurrent, dthetacurrent, eint, gtilde, \
                  Mtildelist, Gtildelist, Slist, thetamatd[:, i], \
                  dthetamatd[:, i], ddthetamatd[:, i], Kp, Ki, Kd)
    for j in range(intRes):
        ddthetalist \
        = ForwardDynamics(thetacurrent, dthetacurrent, taulist, g, \
                       Ftipmat[:, i], Mlist, Glist, Slist)
        thetacurrent, dthetacurrent \
        = EulerStep(thetacurrent, dthetacurrent, ddthetalist, \
                 1.0 * dt / intRes)
    taumat[:, i] = taulist
    thetamat[:, i] = thetacurrent
    eint = np.add(eint, dt * np.subtract(thetamatd[:, i], thetacurrent))
# Output using matplotlib to plot
try:
    import matplotlib.pyplot as plt
```

```
except:
    print('The result will not be plotted due to a lack of package matplotlib')
else:
    links = np.array(thetamat).shape[0]
    N = np.array(thetamat).shape[1]
    Tf = N * dt
    timestamp = np.linspace(0, Tf, N)
    for i in range(links):
        col = [np.random.uniform(0, 1), np.random.uniform(0, 1),
               np.random.uniform(0, 1)]
        plt.plot(timestamp, thetamat[i, :], "-", color=col, \
                 label = ("ActualTheta" + str(i + 1)))
        plt.plot(timestamp, thetamatd[i, :], ".", color=col, \
                 label = ("DesiredTheta" + str(i + 1)))
    plt.legend(loc = 'upper left')
    plt.xlabel("Time")
    plt.ylabel("Joint Angles")
    plt.title("Plot of Actual and Desired Joint Angles")
    plt.show()
taumat = np.array(taumat).T
thetamat = np.array(thetamat).T
return (taumat, thetamat)
```