Practicing Coordinate Transforms

In this notebook we will be practicing how to transform vectors between the various reference frames used for navigating and controlling an autonomous quadrotor. For this work we will be relying heavily on the tf/transformations.py library.

In [21]:
```
from __future__ import division, print_function
import numpy as np
import transformations as tft

# Variable Notation:
# v__x: vector expressed in "x" frame
# q_x_y: quaternion of "x" frame with respect to "y" frame
# p_x_y__z: position of "x" frame with respect to "y" frame expressed in
# v_x_y__z: velocity of "x" frame with respect to "y" frame expressed in
# R_x2y: rotation matrix that maps vector represented in frame "x" to rep
#
# Frame Subscripts:
# dc = downward-facing camera (body-fixed, non-inertial frame. Origin: do
# fc = forward-facing camera (body-fixed, non-inertial frame. Origin: for
# bu = body-up frame (body-fixed, non-inertial frame. Origin: drone cente
# bd = body-down frame (body-fixed, non-inertial frame. Origin: drone cen
# lenu = local East-North-Up world frame (world-fixed, inertial frame. Or
# lned = local North-East-Down world frame (world-fixed, inertial frame.
# m = marker frame (inertial or non-inertial, depending on motion of mark
```

Concept Questions

Velocities of Relative Frames

Just based on your understanding of what the different reference frames represent, can you find the following velocities just by inspection? The first is filled out for you to show answer format:

1. v_bu_bu__bu = [1.0, 0.0, 0.0]
2. v_dc_dc__dc =
3. v_bd_bd__fc =
4. v_dc_dc__lned =
5. v_dc_bu__lenu =

In [22]:
```
# Place your answers here in markdown or comment form

# YOUR CODE HERE
# 1. [1.0, 0.0, 0.0]
# 2. [0.0, 0.0, 0.0]
# 3. [0.0, 0.0, 0.0]
# 4. [0.0, 0.0, 0.0]
# 5. [0.0, 0.0, 0.0]
```

Valid Operations with Relative Frames

Which of these operations are valid and which are invalid. If valid, write the resulting variable name. If invalid, give a brief explanation why. The first is filled out for you to show answer format:

1. p_m_fc__lenu - p_bu_fc__lenu => p_m_bu__lenu
2. np.dot(R_bu2lenu, v_bu_lenu__bu) =>
3. np.dot(R_bu2lenu, v_bu_lenu__lenu) => invalid, R_bu2lenu maps a vector expressed in bu to a vector expressed in lenu, but v_bu_lenu__lenu is already expressed in lenu therefore this will generate a non-sensical answer
4. v_dc_m__dc - p_m_lenu__dc =>
5. p_bu_fc__lned - np.dot(R_m2lned, p_m_fc__m) =>

In [23]:
```
# Place your answers here in markdown or comment form

# YOUR CODE HERE
# 2. v_bu_lenu__lenu
# 4. invalid, it is non-sensical to ubstract velocity and position
# 5. np.dot(R_m2lned, p_m_fc__m) => p_m_fc__lned
#    p_bu_fc__lned - p_m_fc__lned => p_bu_m__lned
```

Fixed/Static Relative Rotations

This rotation matrices are constant, they don't change regardless of the motion of the quadrotor. We can use this knowledge to "hard code" a set of transformations into a class we call `StaticTransforms` which can be used throughout our flight code.

In the next code block, you will need to complete some of the components and variable definitions of the `StaticTransforms` class

In [24]:
```
class StaticTransforms():

    def __init__(self):
        pass

    # local ENU and local NED
    R_lenu2lned = np.array([[0.0, 1.0, 0.0, 0.0],
                            [1.0, 0.0, 0.0, 0.0],
                            [0.0, 0.0,-1.0, 0.0],
                            [0.0, 0.0, 0.0, 0.0]])

    # body-up and body-down
    R_bu2bd = tft.rotation_matrix(np.pi, (1,0,0))

    # downward camera and body-down
    # YOUR CODE HERE
    R_dc2bd = tft.identity_matrix()

    # forward camera and body-down
    R_fc2bd = np.array([[0.0, 0.0, 1.0, 0.0],
                        [1.0, 0.0, 0.0, 0.0],
                        [0.0, 1.0, 0.0, 0.0],
```

```
                             [0.0, 0.0, 0.0, 1.0]])

    # Find inverse rotation matrices
    R_lned2lenu = R_lenu2lned.T
    R_bd2bu = R_bu2bd.T
    R_bd2dc = R_dc2bd.T
    R_bd2fc = R_fc2bd.T

    # Find concatenated rotation matrices from downward-camera to forward
    R_dc2fc = tft.concatenate_matrices(R_bd2fc, R_dc2bd)
    R_fc2dc = R_dc2fc.T
    R_dc2bu = tft.concatenate_matrices(R_bd2bu, R_dc2bd)
    R_bu2dc = R_dc2bu.T
    R_fc2bu = tft.concatenate_matrices(R_bd2bu, R_fc2bd)
    R_bu2fc = R_fc2bu.T

    def coord_transform(self, v__fin, fin, fout):
        ''' transform vector v which is represented in frame fin into its
        Args:
        - v__fin: 3D vector represented in fin coordinates
        - fin: string describing input coordinate frame (bd, bu, fc, dc,
        - fout: string describing output coordinate frame (bd, bu, fc, dc
        Returns
        - v__fout: vector v represent in fout coordinates
        '''

        # trivial transform, checking input shape
        if fin==fout:
            v4__fin = list(v__fin)+[0.0]
            R = tft.identity_matrix()
            v4__fout = np.dot(R, v4__fin)
            v__fout = v4__fout[0:3]
            return v__fout

        # check for existence of rotation matrix
        R_str = 'R_{}2{}'.format(fin, fout)
        try:
            R_fin2fout = getattr(self, R_str)
        except AttributeError:
            err = 'No static transform exists from {} to {}.'.format(fin,
            err += ' Are you sure these frames are not moving relative to
            raise AttributeError(err)

        # perform transform
        v4__fin = list(v__fin) + [0.0]
        # YOUR CODE HERE
        v4__fout = np.dot(R_fin2fout, v4__fin)
        v__fout = v4__fout[0:3]
        return v__fout

st = StaticTransforms()
```

In [25]:

```
assert np.allclose(st.coord_transform([1.0, 0.0, 0.0], 'bu', 'bu'), [1.0,
assert np.allclose(st.coord_transform([0.08511008, 0.38572187, 0.51372079
assert np.allclose(st.coord_transform([0.0, 0.0, 1.0], 'fc', 'bd'), [1.0,
assert np.allclose(st.coord_transform([0.0, 0.0, 1.0], 'dc', 'bu'), [0.0,
```

In [26]:

```
# Let's assume the quadrotor has some velocity v1_bd_lned__bd which is th
# body-down frame with respect to the local NED world frame expressed in
```

```
# Using the fixed relative rotations, calculate it's expression in the bo
v1_bd_lned__bd = [1.0, 0.0, 0.0]

# YOUR CODE HERE
v1_bd_lned__bu = st.coord_transform(v1_bd_lned__bd, "bd", "bu")
v1_bd_lned__dc = st.coord_transform(v1_bd_lned__bd, "bd", "dc")
v1_bd_lned__fc = st.coord_transform(v1_bd_lned__bd, "bd", "fc")

print(v1_bd_lned__bu)
print(v1_bd_lned__dc)
print(v1_bd_lned__fc)

# Let's assume the quadrotor has some velocity v2_bd_lned__bd which is th
# body-down frame with respect to the local NED world frame expressed in
# Using the fixed relative rotations, calculate it's expression in the bo

v2_bd_lned__bd = [0.147, 0.798, 1.221]

# YOUR CODE HERE

v2_bd_lned__bu = st.coord_transform(v2_bd_lned__bd, "bd", "bu")
v2_bd_lned__dc = st.coord_transform(v2_bd_lned__bd, "bd", "dc")
v2_bd_lned__fc = st.coord_transform(v2_bd_lned__bd, "bd", "fc")

print(v2_bd_lned__bu)
print(v2_bd_lned__dc)
print(v2_bd_lned__fc)

# Let's assume the quadrotor has some velocity v3_dc_lenu__dc which is th
# downward-camera frame with respect to the local ENU world frame express
# Using the static transforms, calculate it's expression in the body-down
v3_dc_lenu__dc = [4.853, 2.979, 1.884]

# YOUR CODE HERE

v3_dc_lenu__bd = st.coord_transform(v3_dc_lenu__dc, "dc", "bd")
v3_dc_lenu__fc = st.coord_transform(v3_dc_lenu__dc, "dc", "fc")
v3_dc_lenu__bu = st.coord_transform(v3_dc_lenu__dc, "dc", "bu")

print(v3_dc_lenu__bd)
print(v3_dc_lenu__fc)
print(v3_dc_lenu__bu)

# Let's assume the quadrotor has some velocity v4_fc_lenu__bd which is th
# forward-camera frame with respect to the local ENU world frame expresse
# Using the static transforms, calculate it's expression in the forward-c

v4_fc_lenu__bd = [0.0, 0.0, -1.0]

# YOUR CODE HERE

v4_fc_lenu__fc = st.coord_transform(v4_fc_lenu__bd, "bd", "fc")
v4_fc_lenu__dc = st.coord_transform(v4_fc_lenu__bd, "bd", "dc")
v4_fc_lenu__bu = st.coord_transform(v4_fc_lenu__bd, "bd", "bu")

print(v4_fc_lenu__fc)
print(v4_fc_lenu__dc)
print(v4_fc_lenu__bu)
```

```
[1. 0. 0.]
[1. 0. 0.]
[0. 0. 1.]
[ 0.147 -0.798 -1.221]
[0.147 0.798 1.221]
[0.798 1.221 0.147]
[4.853 2.979 1.884]
[2.979 1.884 4.853]
[ 4.853 -2.979 -1.884]
[ 0. -1.  0.]
[ 0.  0. -1.]
[ 0.0000000e+00 -1.2246468e-16  1.0000000e+00]
```

In [27]:

```python
# Autograder, do not modify

assert np.allclose(v1_bd_lned__bd, [1.0, 0.0, 0.0])
assert np.allclose(v2_bd_lned__bd, [0.147, 0.798, 1.221])
assert np.allclose(v3_dc_lenu__dc, [4.853, 2.979, 1.884])
assert np.allclose(v4_fc_lenu__bd, [0.0, 0.0, -1.0])
```

Dynamic Relative Rotations

In the previous section we looked at reference frames that remain fixed relative to one another (i.e. reference frames that are all attached to quadrotor or reference frames, or reference frames that are all associated with inertial local world frames). Now were going to look at reference frames that may be moving relative to one another, such as a body-fixed frame and the local world frame.

For such moving frames, we often can't create rotation matrices by inspection. Furthermore, such rotations need to be calculated automatically by the quadrotor's flight computer in real-time. This is the job of the *state estimator* that runs onboard the flight computer. The state estimator will output estimates of the relative rotations between local world frame and the body frame.

More specifically, the topic `mavros/local_position/pose` provides `PoseStamped` messages that contain the orientation of the body-down frame with respect to the local ENU frame in the form of a `Quaternion` .

Therefore, when using MAVROS, you could use a assignment such as the one below to find `q_bu_lenu` :

```
q_bu_lenu = pose_stamped_msg.pose.orientation
```

Below is a function that we can use when flying the drone to transforms vectors in an arbitrary reference frame to the local ENU reference frame, assuming that we have access to the `mavros/local_position/pose` topic to tell us `q_bu_lenu` (in this case we assume they are velocity vectors)

In [28]:

```python
def get_lenu_velocity(q_bu_lenu, v__fin, fin='bu', static_transforms=None
        '''tranforms a vector represented in fin frame to vector in lenu
        Args:
        - v__fin: 3D vector represented in input frame coordinates
```

```
    - fin: string describing input coordinate frame (bd, bu, fc, dc)
    Returns:
    - v__lenu: 3D vector v represented in local ENU world frame
    '''

    # create static transforms if none given
    if static_transforms is None:
        static_transforms = StaticTransforms()

    if fin=='lenu':
        v__lenu = v__fin

    elif fin=='lned':
        v__lenu = static_transforms.coord_transform(v__fin, 'lned', '

    else:
        # create rotation matrix from quaternion
        R_bu2lenu = tft.quaternion_matrix(q_bu_lenu)

        # represent vector v in body-down coordinates
        v__bu = static_transforms.coord_transform(v__fin, fin, 'bu')

        # calculate lenu representation of v
        v__lenu = np.dot(R_bu2lenu, list(v__bu)+[0.0])

    v__lenu = np.array(v__lenu[0:3])
    return v__lenu
```

ROS tf2 Library

The problems we have attempted to address in this module (i.e. managing multiple reference frames) are by no means unique to quadrotors and we are not the first people to write functions to solve such problems. The functionality of managing different reference frames is ubiquitous throughout robotics, aerospace engineering, mechanical engineering, computer graphics, etc. and many libraries have been written for handling such functionality. When working with ROS, the most important of such libraries is the tf (now tf2) library. While we have access to this library on the drone, we have not made use of it here because it obscures some of the underlying mathematics that we hope for you to learn and it requires additional setup steps when defining new frames that we don't intend to teach. If you are curious to know more about how ROS manages large numbers of reference frames simultaneously, we encourage you to read up more on `tf` .

NOTE: `tf` in the context of ROS should not be confused with TensorFlow which is often abbreviated as tf in code. These libraries have completely different purposes.