FOMAR LABORATORY ASSIGNMENT 1

April 8, 2021

0.1 [FIB UPC] SPRING 2021

0.1.1 Daniel Santiago Corona

---

[2]:
```
import numpy as np
import sympy as sp
import math

sp.init_printing(use_unicode=True)
```

0.2 STEPS 1 & 2 : Symbolic rotation matrix generation

The following function generates a symbolic expression for a rotation matrix given the three euler angles, generating the expression equivalent to ROTZ(alpha) * ROTY(beta) * ROTX(gamma)

[3]:
```
def sym_euler_rot():

    alpha , beta , gamma = sp.symbols(u"alpha beta gamma")

    rotX = sp.Matrix([[1,0,0],
                      [0,sp.cos(gamma),-sp.sin(gamma)],
                      [0,sp.sin(gamma),sp.cos(gamma)]])

    rotY = sp.Matrix([[sp.cos(beta),0,sp.sin(beta)],
                      [0,1,0],
                      [-sp.sin(beta),0,sp.cos(beta)]])

    rotZ = sp.Matrix([[sp.cos(alpha),-sp.sin(alpha),0],
                      [sp.sin(alpha),sp.cos(alpha),0],
                      [0,0,1]])

    return rotZ * rotY * rotX
```

[4]:
```
#TESTING
sym_euler_rot()
```

[4]:

$$\begin{bmatrix} \cos(\alpha)\cos(\beta) & -\sin(\alpha)\cos(\gamma)+\sin(\beta)\sin(\gamma)\cos(\alpha) & \sin(\alpha)\sin(\gamma)+\sin(\beta)\cos(\alpha)\cos(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma)+\cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma)-\sin(\gamma)\cos(\alpha) \\ -\sin(\beta) & \sin(\gamma)\cos(\beta) & \cos(\beta)\cos(\gamma) \end{bmatrix}$$

The previous equation is not only the generic general 3x3 rotation matrix expressed in terms of the 3 variables alpha, beta and gamma (which will be used in Rz, Ry and Rx respectively), but also provide equations to relate the values of the matrix positions to the values of the angles alpha, beta and gamma. We will use this relation in the next section to find the angles given a rotation matrix.

0.3 STEP 3: Find the Euler angles that determine a given rotation matrix

The following function will perform two tasks: 1. Determine the three Euler angles that determine a given rotation matrix 2. (Re)constructs a 3D rotation matrix given three independent parameters (given the M(1,1), M(3,1) and M(3,3) parameters of the original matrix)

For these purposes the function takes as an input only three elements of the matrix, chosen to be the M(1,1), M(3,1) and M(3,3) elements of the matrix. These were chosen because they are enough (as independent values) to reconstruct the matrix and therefore to calculate the corresponding Euler values.

The function's output consists of a tuple containing a NumPy array with the 3 euler values (alpha, beta and gamma for a RzRyRx rotation) and a NumPy array with the values of the reconstructed matrix.

[5]:
```
def rot_zyx(m11,m31,m33):

    try:
        beta = -math.asin(m31)
        alpha = math.acos(m11/math.cos(beta))
        gamma = math.acos(m33/math.cos(beta))
    except:
        print("CAN'T CONSTRUCT A ROTATION MATRIX FROM THOSE PARAMETERS")
        print("USE VALID M11,M31,M33 VALUES OF A ROTATION MATRIX")
        return

    rotZYX = np.array([[math.cos(alpha)*math.cos(beta) , -math.cos(gamma)*math.
↪sin(alpha) + math.cos(alpha)*math.sin(beta)*math.sin(gamma) , math.
↪cos(alpha)*math.cos(gamma)*math.sin(beta) + math.sin(alpha)*math.sin(gamma)],
                [math.cos(beta)*math.sin(alpha) , math.cos(alpha)*math.cos(gamma)␣
↪+ math.sin(alpha)*math.sin(beta)*math.sin(gamma) , math.cos(gamma)*math.
↪sin(alpha)*math.sin(beta) - math.cos(alpha)*math.sin(gamma)],
                [-math.sin(beta) , math.cos(beta)*math.sin(gamma) , math.
↪cos(beta)*math.cos(gamma)]])

    return np.array([alpha,beta,gamma]),rotZYX
```

The problem with the previous function is the singularities that might appear, since **asin(0) =**

0 we lose information when **m31 = 0**, and we will not be able to tell the rotation in the Y axis properly. Furthermore, when the value of **m31 = 1** we find a division by 0 which will mess our calculations.

[6]:
```
#TESTING FOR THE GIVEN EXAMPLE (THE OUTPUT HAS BEEN TRANSFORMED FROM RADIANS TO␣
↪DEGREES)
ang, arr = rot_zyx(0.53632,0.0,0.58379)
print(arr)
print()
print(ang * 180/math.pi)
```

```
[[ 0.53632    -0.49272736  0.68525952]
 [ 0.84401473  0.31309825 -0.43544073]
 [ 0.           0.8119047   0.58379   ]]

[57.56652508 -0.         54.28244502]
```

0.4 STEP 4 & 5 & 6: Reading a rotation matrix and writing its Euler angles

The following code will read a rotation matrix in the format specified by the lab's guidelines from a file named "rijxcolumnes" which must be in the same directory as this program, it will output the euler angles in a file named "fisef.out" which will also be located at the script's directory.

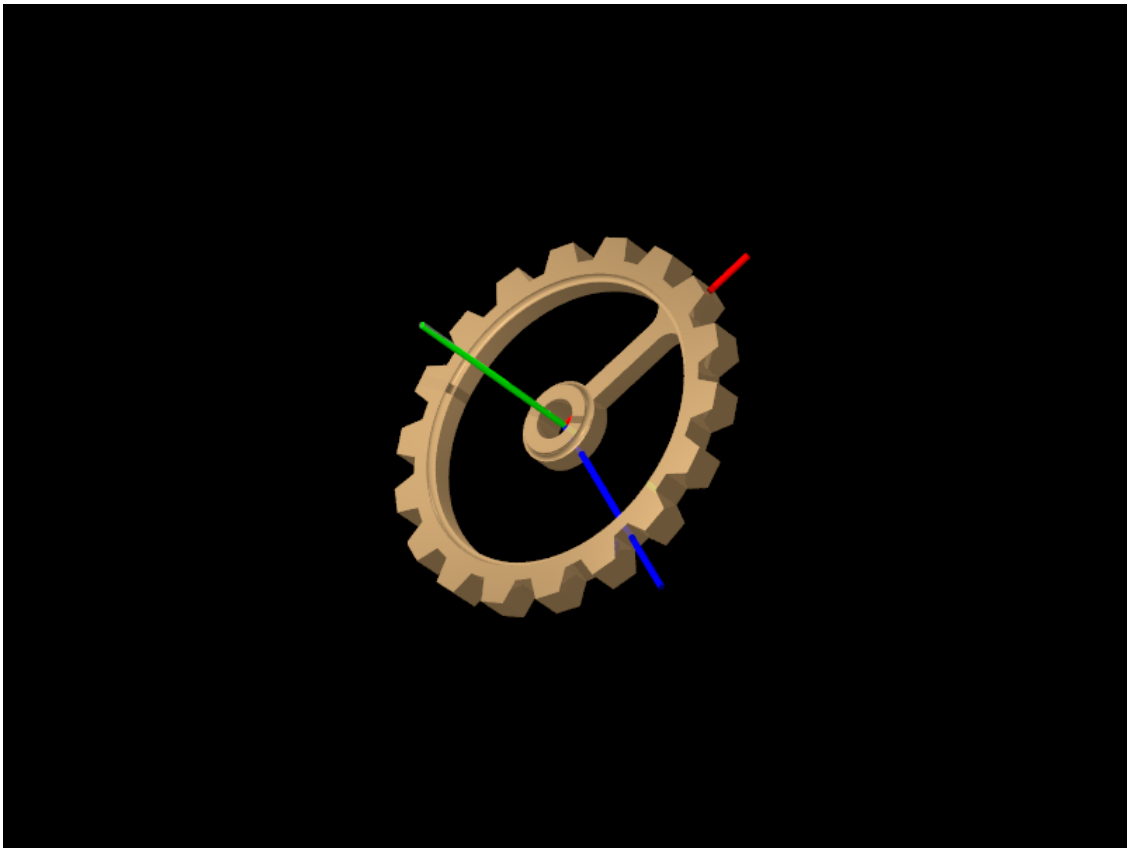
[9]:
```
try:
    file = open("rijxcolumnes","r")
    text = file.read()
    file.close();
except:
    print("A FILE NAMED 'rijxcolumnes' HAS NOT BEEN FOUND")
    quit()

num_list = text.split()
rij_cols = [float(n[0:-1]) for n in num_list[0:-1]]
rij_cols.append(float(num_list[-1]))

c1 = np.array(rij_cols[0:3])
c2 = np.array(rij_cols[3:6])
c3 = np.array(rij_cols[6:])

R = np.column_stack((c1,c2,c3))

angles , mat = rot_zyx(R[0][0],R[2][0],R[2][2])

angles = angles * 180/math.pi
output = "{} , {} , {}".format(angles[0],angles[1],angles[2])

file = open("fisef.out", "w")
file.write(output)
```

```
file.close()
```

The angles obtained with our algorithm are proven to be correct, as the image generated with the command

```
povray +w800 +h600 +a0.2 p1_verifica.pov
```

shows no difference between the frame of reference calculated by the original rotation matrix and the one calculated by our Euler angles.



0.5 STEP 7 & 8 & 9: Visualizing an animated rotation

After getting to play with and understand the fundamentals of *.pov* files we generated animated versions of the rotations calculated by our algorithm. Knowing all animations would start with the identity matrix as initial state, we used as a final state the rotation matrix given as an example. The multiple *.png* images generated were transformed to *.mpg and .gif* files with the commands:

```
convert -delay 5 *png movie.gif
convert -delay 5 *png movie.mpg
```

The result of this first animation can be seen in the *example_rot* files (both gif and mpg).

After this, we modified the value of the variable responsible for the time change (named Factor) in the *.pov* files from a trivial

```
#declare Factor=clock;
```

to

```
#declare Factor=sin(pi*clock)*sin(pi*clock);
```

And the appearance of the sinusoidal function in the time variable will now provoke a cyclic animation, as one can see in the files *example_sin_rot* (both gif and mpg).

0.6 ID Card rotation matrix generation and visualization

Using the algorithms we wrote down earlier we will generate a unique rotation matrix using digits from a fake ID card.

```
ang , arr = rot_zyx(0.50,0.31,0.04) #39 40 13 05
ang * 180/math.pi
```

This code will generate not only the 3 euler angles (in degrees) that we will need to describe the rotation but also the reconstructed matrix that we also need to feed POV-RAY.

[10]:
```
ang , arr = rot_zyx(0.50,0.31,0.04) #39 40 13 05
print(ang * 180/math.pi)
print("-- -- -- -- --")
print(arr)
```

```
[ 58.27060774 -18.05923049  87.58870337]
-- -- -- -- --
[[ 0.5        -0.19867166  0.84292916]
 [ 0.80864083 -0.24130804 -0.53653558]
 [ 0.31        0.94989473  0.04      ]]
```

The videos generated for this matrix can be found in this project in both .mpg and .gif files, and also for the trivial and the sinusoidal time factor, named *dni_rot and dni_sin_rot.*

1 – – – – – – – – – – –

1.1 STEP 10 : Choosing a different Euler angles characterization

- New symbolic rotation matrix : Instead of constructing a general rotation matrix using ROTz*ROTy*ROTx we will now use the ROTx*ROTy*ROTx configuration, as it is a valid Euler angles configuration.

[11]:
```
def alt_sym_euler_rot():

    alpha , beta , gamma = sp.symbols(u"alpha beta gamma")

    rotX2 = sp.Matrix([[1,0,0],
                       [0,sp.cos(gamma),-sp.sin(gamma)],
                       [0,sp.sin(gamma),sp.cos(gamma)]])

    rotY = sp.Matrix([[sp.cos(beta),0,sp.sin(beta)],
                      [0,1,0],
                      [-sp.sin(beta),0,sp.cos(beta)]])
```

```
    rotX = sp.Matrix([[1,0,0],
                      [0,sp.cos(alpha),-sp.sin(alpha)],
                      [0,sp.sin(alpha),sp.cos(alpha)]])

    return rotX * rotY * rotX2
```

[12]:
```
#Testing
alt_sym_euler_rot()
```

[12]:
$$\begin{bmatrix} \cos(\beta) & \sin(\beta)\sin(\gamma) & \sin(\beta)\cos(\gamma) \\ \sin(\alpha)\sin(\beta) & -\sin(\alpha)\sin(\gamma)\cos(\beta)+\cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\beta)\cos(\gamma)-\sin(\gamma)\cos(\alpha) \\ -\sin(\beta)\cos(\alpha) & \sin(\alpha)\cos(\gamma)+\sin(\gamma)\cos(\alpha)\cos(\beta) & -\sin(\alpha)\sin(\gamma)+\cos(\alpha)\cos(\beta)\cos(\gamma) \end{bmatrix}$$

Now we need to repeat our function to generate both a regenerated matrix and the Euler angles that codify the rotation. Notice the angle *gamma* now requires a more sofisticated operation because it needs to solve an equation of the form $\mathbf{A}\boldsymbol{sinx} + \boldsymbol{B}\mathbf{cosx} = \mathbf{C}$.

[13]:
```
def rot_xyx2(m11,m31,m33):

    try:
        beta = math.acos(m11)
        alpha = math.acos(-m31/math.sin(beta))

        A = -math.sin(alpha)
        B = math.cos(beta) * math.cos(alpha)
        gamma = -math.asin(m33/math.sqrt(A*A + B*B)) - math.atan(B/A)
    except:
        print("CAN'T CONSTRUCT A ROTATION MATRIX FROM THOSE PARAMETERS")
        print("USE VALID M11,M31,M13 VALUES OF A ROTATION MATRIX")
        return

    rotXYX = np.array([[math.cos(beta) , math.sin(beta)*math.sin(gamma) , math.
↪cos(gamma)*math.sin(beta) ],
                [math.sin(alpha)*math.sin(beta) , -math.sin(alpha)*math.
↪sin(gamma)*math.cos(beta) + math.cos(alpha)*math.cos(gamma) , -math.
↪sin(alpha)*math.cos(beta)*math.cos(gamma) - math.sin(gamma)*math.cos(alpha)␣
↪],
                [-math.sin(beta)*math.cos(alpha) , math.sin(alpha)*math.
↪cos(gamma) + math.sin(gamma)*math.cos(alpha)*math.cos(beta) , -math.
↪sin(alpha)*math.sin(gamma) + math.cos(alpha)*math.cos(beta)*math.cos(gamma)␣
↪]])

    return np.array([alpha,beta,gamma]),rotXYX
```

[14]:
```
#TESTING
angles,x = rot_xyx2(0.50,0.31,0.04)
angles = angles * 180/math.pi
```

```
print(angles,'\n'*2,x)

#for fil in x:
 #   for col in fil:
  #      print(col, end=', ')
```

```
[110.97479154  60.         -13.26211932]

 [[ 0.5        -0.19867166  0.84292916]
 [ 0.80864083 -0.24130804 -0.53653558]
 [ 0.31        0.94989473  0.04      ]]
```

Repeating the reading and writing of the files involved in the *POV-RAY* process. The two versions for this reading and writing, each one for a different Euler configuration, will be attached as two separate python scripts for ease of use.

[15]:
```
try:
    file = open("rijxcolumnes","r")
    text = file.read()
    file.close();
except:
    print("A FILE NAMED 'rijxcolumnes' HAS NOT BEEN FOUND")
    quit()

num_list = text.split()
rij_cols = [float(n[0:-1]) for n in num_list[0:-1]]
rij_cols.append(float(num_list[-1]))

c1 = np.array(rij_cols[0:3])
c2 = np.array(rij_cols[3:6])
c3 = np.array(rij_cols[6:])

R = np.column_stack((c1,c2,c3))

angles , mat = rot_xyx2(R[0][0],R[2][0],R[2][2])

angles = angles * 180/math.pi
output = "{} , {} , {}".format(angles[0],angles[1],angles[2])

file = open("fisef.out", "w")
file.write(output)
file.close()
```

Now, after modifying the POV_RAY scripts to adapt to the new rotations by adding the following lines:

```
#fopen Fis "fisef.out" read
#read (Fis,f1x2,f1y,f1x)
...
```

```
object{Roda_dentada rotate Factor*f1x*x rotate Factor*f1y*y rotate Factor*f1x2*x}

object {SRef rotate Factor*f1x*x rotate Factor*f1y*y rotate Factor*f1x2*x}
```

We can execute POV_RAY again to obtain the results in the *.gif and .mpg* files XYX_dni_rot and XYX_dni_sin_rot, one can easily realize that the rotation even if performed over different rotation axis and angles is equivalent as it ends with the same pose as the previous set of rotations.

2 Conclusions

All tasks in this laboratory assignment have been succesfully completed, the final report will organize the different scripts, test inputs and media generated in different directories. The theory concepts worked in this assignment were already thoroughly worked at class, so the biggest difficulties were related to learn POV-RAY and the use of Python as a numerical computing environment.