Delivery 1

Física orientada a la Modelització i Animació Realista

Aarón Acosta Fernández

Júlia Gasull i Navarro

# Index

[**Index**](#_4t9caenbsdk5) **2**

[**Steps to follow**](#_ka9kfqcde3ob) **3**

[Step 1](#_7dmqn9ofv79y) 3

[Step 2](#_et3h0k371quz) 6

[Step 3, 4 & 5](#_csky7tio42rg) 7

[Step 6](#_vezwjdxeikkc) 9

[Step 7 & 8](#_z56zm96md0jr) 10

[Step 9](#_6unaqx6w91bp) 11

[Step 10](#_kfcurenpwfwp) 12

# 

# Steps to follow

## Step 1

1. Find out the (symbolic) general 3x3 rotation matrix expressed in terms of the 3 variables *fi\_i* **(Euler's angles)** entering in R1(z,fi\_z) R2(y,fi\_y) R3(x,fi\_x) (just making the product of the 3 matrices does it).

We have decided to use one of our DNIs to extract the R matrix. Since it is something mechanical, we have programmed a python script to be able to find this matrix without problems.

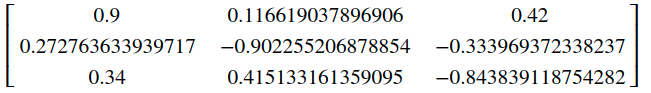
The code is the following:

|  |
| --- |
| #!/usr/bin/env python  # coding: utf-8  # # Building the complete rotation matrix from my ID number: [49244309]  import numpy as np  from sympy.solvers.solveset import nonlinsolve  from sympy import Symbol  from sympy import \*  import math  # #### Assigning value to 3 elements of the matrix based on my ID number  x1 = 0.90  x3 = 0.34  z1 = 0.42  # #### Finding out x2 and y1 elements using the unit vector property of rotation matrices  x2 = np.sqrt(1 - x1\*\*2 - x3\*\*2)  y1 = np.sqrt(1 - z1\*\*2 - x1\*\*2)  print(f'x2 = {x2}\ny1 = {y1}')  # #### Finding out the remaining values using the orthonormality and unit vector properties  y2 = Symbol('y\_2')  y3 = Symbol('y\_3')  z2 = Symbol('z\_2')  z3 = Symbol('z\_3')  #Defining the R matrix we want to find out  #Find y2 and y3.  c1\_c2\_eq = x1\*y1 + x2\*y2 + x3\*y3  c2\_c2\_eq = y1\*y1 + y2\*y2 + y3\*y3 - 1  #Two solutions were provided. The first one is chosen  y2, y3 = nonlinsolve([c1\_c2\_eq, c2\_c2\_eq] , [y2, y3]).args[0]  #Find z2 and z3  c1\_c3\_eq = x1\*z1 + x2\*z2 + x3\*z3  c2\_c3\_eq = y1\*z1 + y2\*z2 + y3\*z3  c3\_c3\_eq = z1\*z1 + z2\*z2 + z3\*z3 - 1  z2, z3 = nonlinsolve([c1\_c3\_eq, c2\_c3\_eq, c3\_c3\_eq] , [z2, z3]).args[0]  R = Matrix([[x1, y1, z1],  [x2, y2, z2],  [x3, y3, z3]])  # #### So, the R matrix is as follows  R  # --------------------------------------------------------  fi\_x = Symbol('fi\_x')  fi\_y = Symbol('fi\_y')  fi\_z = Symbol('fi\_z')  R = Matrix([[0.5363, 0.6830, -0.4958],  [-0.3020, 0.7039, 0.6429],  [0.7881, -0.1951, 0.5838]]  ).transpose()  R  R = Matrix([[0.5363, 0.6830, -0.4958],  [-0.3020, 0.7039, 0.6429],  [0.7881, -0.1951, 0.5838]]  ).transpose()  R  Rz = Matrix([[cos(fi\_z), -sin(fi\_z), 0],  [sin(fi\_z), cos(fi\_z), 0],  [0,0,1]])  Ry = Matrix([[cos(fi\_y), 0, sin(fi\_y)],  [0, 1, 0],  [-sin(fi\_y), 0, cos(fi\_y)]])  Rx = Matrix([[1,0,0],  [0, cos(fi\_x), -sin(fi\_x)],  [0, sin(fi\_x), cos(fi\_x)]])  R\_fin = Rz\*Ry\*Rx  equations\_system = (R\_fin[0,2] - R[0,2],  R\_fin[2,1] - R[2,1],  R\_fin[1,0] - R[1,0])  # \*\*Get the solutions to the equations system\*\*  solutions\_rad = solve(equations\_system, fi\_z, fi\_y, fi\_x)  # \*\*Angles are in radians. Convert them to degrees\*\*  solutions\_deg = []  for i in solutions:  solutions\_deg.append(tuple((math.degrees(i[0]),  math.degrees(i[1]), math.degrees(i[2]))))  print(math.degrees(i[0]), math.degrees(i[1]), math.degrees(i[2]))  # \*\*4th solution from "solutions" is the right one\*\*  solutions\_deg[4] |

For this, we have used the following three numbers:

* x1 = 0.90
* x3 = 0.34
* z1 = 0.42

The following rotation matrix has been generated:

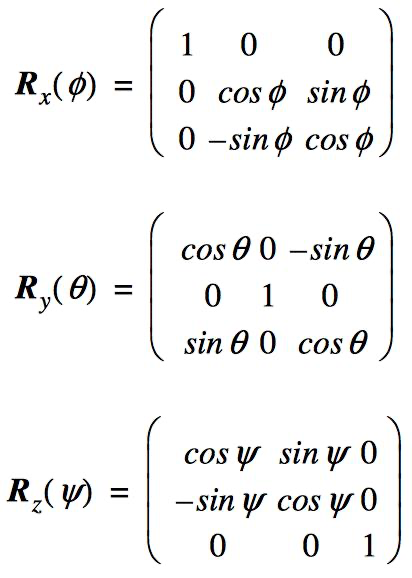


## 

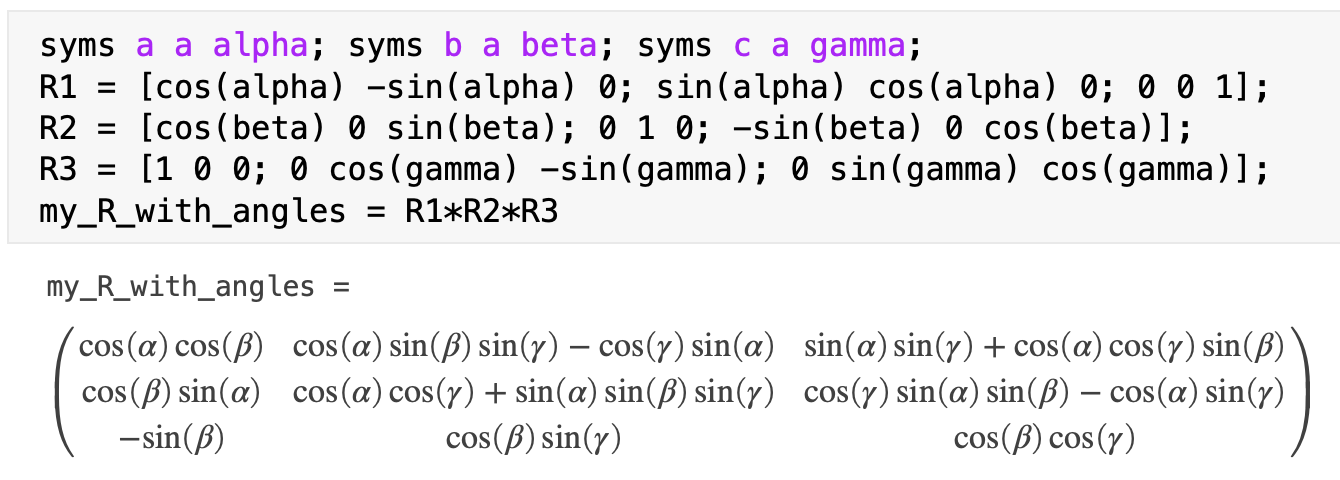
## Step 2

1. Write down the equations which relate the r\_ij values of the general 3x3 matrix to the fi\_z, fi\_y, fi\_x angles.

In this case, we have used matlab to be able to calculate this matrix, since we know that ...



We have made a small script in matlab that has allowed us to relate the angles to the fi\_i, where i = {x, y, z}.



## 

## Step 3, 4 & 5

1. Write an algorithm which allows to find fi\_z, fi\_y, fi\_x from the r\_ij values (see point 5 below)
2. Implement it in a program which will read the file **rijxcolumnes** and compute and print the values of fi\_z, fi\_y, fi\_x into the file fisef.out (separated by commas).
3. Execute your program, generating the file fisef.out (Note that this file exists already, so that you can check if your programs works correctly). If so, you will be able to complete the following points.

For this step, we have written the following algorithm, in python:

|  |
| --- |
| **Initial note:** since there are several solutions, we have taken, from all possible angles, what we found most realistic.  **Note added later:** we have realized that not all angles work, so we have chosen one that does. |
| #!/usr/bin/env python  # coding: utf-8  import numpy as np  from sympy.solvers.solveset import nonlinsolve  from sympy import Symbol  from sympy import \*  import math  import csv  R = Matrix()  #Read rijxcolumnes file  with open('rijxcolumnes') as file:  csv\_reader = csv.reader(file, delimiter=',')  for r in csv\_reader:  R = Matrix([[r[0], r[3], r[6]],  [r[1], r[4], r[7]],  [r[2], r[5], r[8]]])  R  # gamma -> fi\_x  # beta -> fi\_y  # alpha -> fi\_z  gamma = Symbol('gamma')  beta = Symbol('beta')  alpha = Symbol('alpha')  eq\_beta = -sin(beta) - R[2,0]  fi\_y = solve(eq\_beta, beta)[1]  fi\_y  eq\_alpha = cos(fi\_y)\*cos(alpha) - R[0,0]  fi\_z = solve(eq\_alpha, alpha)[1]  fi\_z  eq\_gamma = cos(fi\_y)\*sin(gamma) - R[2,1]  fi\_x = solve(eq\_gamma, gamma)[0]  fi\_x  print(math.degrees(fi\_x), math.degrees(fi\_y), math.degrees(fi\_z))  with open('fisef.out', 'w') as file:  writer = csv.writer(file, delimiter=',')  writer.writerow([round(math.degrees(fi\_z), 2),  round(math.degrees(fi\_y), 2), round(math.degrees(fi\_x), 2)]) |

The initial result obtained has been the following:

* fi\_x = 26.18
* fi\_y = -19.87
* fi\_z = 16.86

After doing several tests, we have seen that any of these angles do not work. Therefore, we have tried all that we could until we reached a solution:

* fi\_x = -26.1952
* fi\_y = 199.8768
* fi\_z = 196.8605

## 

## Step 6

1. Have a look into the file p1\_verifica.pov, particularly the lines:

**#fopen Rij "rijxcolumnes" read**

**#read (Rij,r11,r21,r31,r12,r22,r32,r13,r23,r33)**

**#fopen Fis "fisef.out" read**

**#read (Fis,f1z,f1y,f1x)**

The Pov-Ray code reads both the r\_ij elements and the Euler's angles (Fis,f1z,f1y,f1x). Note that the desired orientation coming from r\_ij is displayed in the form of a reference system in grey colour (objecte SRob, Sistema de Referencia objectiu):

**// Escena**

**object {SRob matrix < r11,r21,r31,r12,r22,r32,r13,r23,r33, 0.0, 0.0, 0.0 > }**

**//**

and on the other hand the representation of the object (and the coloured reference system fixed to it) is based on the Euler angles:

**object{Roda\_dentada rotate f1x\*x rotate f1y\*y rotate f1z\*z}**

**object {SRef rotate f1x\*x rotate f1y\*y rotate f1z\*z}**

Note that the first transformation applied to the coordinates of the object points is R(x), the second R(y) and the last R(z), i.e., the reverse order of the one used in point 1. This is a due to the fact that Pov-Ray always uses the universal reference system when applying transformations.

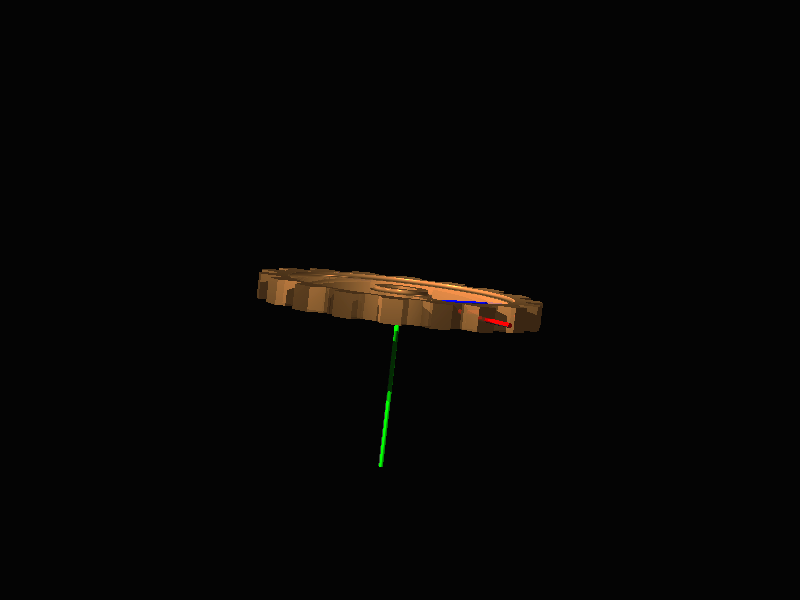
Render the scene:

povray +w800 +h600 +a0.2 p1\_verifica.pov

eom p1\_verifica.png

If everything went well, the 3 grey axes and the colour ones will coincide (meaning that the orientation generated by the 3 elementary rotations that you have calculated is correct, i.e, they reproduce the full 9 elements of the required rotation matrix). If this is the case, then the grey axes will not be visible, since they are inside the coloured ones, which are slightly larger in diameter.

The generated image has been the following:



## Step 7 & 8

1. Create a new directory, where you will generate a sequence of images displaying a continuously varying time dependent orientation. Copy the file rijxcolumnes and replace the values with that of the matrix that you built in point a). (Failing that, you can use instead any of the rotation matrices available in the subdirectory final). Copy also the various .pov and .ini files and execute your program again, generating the file fisef.out for your matrix. It is required that the sequence satisfies that at t=0 the orientation is given by the identity matrix (i.e., that shown by p1\_escena.pov) whilst at t=1 it must match the final orientation shown by p1\_verifica.pov.

Pov-Ray uses a variable named clock whose values are specified in an initialization file (p1\_anim.ini in this example), so that a single povray code can generate a varying sequence of images.

In order to achieve this, we will substitute the line

**rotate f1x\*x rotate f1y\*y rotate f1z\*z**

by

**rotate Factor\*f1x\*x rotate Factor\*f1y\*y rotate Factor\*f1z\*z**

where Factor is a clock dependent value which smoothly changes from 0 to 1 as time goes on. (This has already been done in the files p1\_anim.pov i p1\_anim.ini, and you just have to modify the parameters in p1\_anim.ini related to the number of frames contained in the interval between t=0 and t=1).

The effect of the time value (actual value of the clock variable) on the rendered image can be examined in a simple manner by means of the +K command line option. For example, **povray +w800 +h600 +a0.2 +K0.2 p1\_anim.pov** will generate the scene with an actual value of the clock variable equal to 0.2 Once you are confident that several clock values in the range [0-1] do provide reasonable results, you can execute **povray +w800 +h600 +a0.2 p1\_anim.ini**

1. Have a look at the bunch of png images. If you issue the command

**eom \*png**

you can quickly pass images forward by pressing the space key, and backwards pressing the backspace key. Generate an mpg file with the convert command.

Video link: [animate.mov](https://drive.google.com/file/d/18NtW5mCp_qHUbKxrNtuuxvuuxW0RLPWr/view?usp=sharing)

## 

## 

## Step 9

1. In the just created animation, we made the 3 Euler's angles change linearly with time, as the clock value grows uniformly between the values defined in p1\_anim.ini, and the lines

**// #declare Factor=sin(pi\*clock)\*sin(pi\*clock);**

**#declare Factor=clock;**

make trivially Factor to be equal to clock. Create a new animation in which the rotation angles Factor\*f1x, Factor\*f1y, Factor\*f1z follows a time dependence of the kind sin^2(t), just commenting and uncommenting the appropriate lines declaring the Factor variable in the file p1\_anim.pov .

Video link: [animate-sin.mov](https://drive.google.com/file/d/1eCPirckzCr7zINskGFI9YpIggxzHDl5r/view?usp=sharing)

## 

## Step 10

1. Finally repeat the above points using the same rotation matrix coming from your id# data with another Euler angles characterization, different from the conventional ordering R1(z,fi\_z) R2(y,fi\_y) R3(x,fi\_x), i.e, choosing another set of three elementary rotations (at your discretion). You will need to compute the symbolic contents of the general rotation matrix associated to your chosen order, build a new code which computes the 3 new Euler angle values, generates the file fisef.out, and modify the lines in p1\_anim.pov:

**#fopen Fis "fisef.out" read**

**#read (Fis,f1z,f1y,f1x)**

and

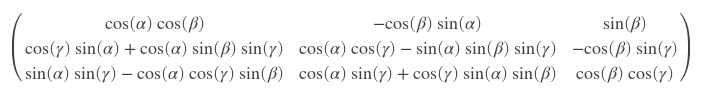
**rotate f1x\*x rotate f1y\*y rotate f1z\*z**

and/or

**rotate Factor\*f1x\*x rotate Factor\*f1y\*y rotate Factor\*f1z\*z**

so that they reflect accordingly the order you have chosen to parametrize the rotation matrix in terms of Euler angles.

R3\*R2\*R1:



Angles found:

* fi\_x = -7,3437
* fi\_y = 24,8346
* fi\_z = 158,4056

Video link: [animate](https://drive.google.com/file/d/1g1TdOGEnCDOx9_5OjDhQveCYX8yjD7id/view?usp=sharing)