Robótica Computacional

Samuel Martín Morales alu0101359526@ull.edu.es

23 de diciembre de 2023

Índice general

Capítulo 1 Introducción

Durante el desarrollo de la asignatura *Robótica Computacional* se ha realizado la implementación de tres prácticas relacionadas con el contenido teórico que se ha visto a lo largo de las semanas lectivas. Además de las tres prácticas mencionadas, se ha solicitado una cuarta práctica denominada como *Filtro de partículas*, la cual se ha determinado como práctica opcional. Por ello, en este documento se detallará el desarrollo de las tres primeras prácticas, dejando la cuarta práctica como trabajo futuro, debido a la falta de tiempo para su desarrollo.

Capítulo 2 Cinemática directa mediante Denavit Hartenberg

2.1. ¿Qué es la Cinemática Directa?

La cinemática de los manipuladores es la ciencia que estudia el movimiento de los componentes físicos del robot sin tener en cuenta o considerar las fuerzas que intervienen en el movimiento. Dentro de la cinemática se realiza el estudio de la velocidad, la posición, la aceleración con respecto al tiempo o a cualquier otra variable. Por tanto, la cinemática directa es el problema estático geométrico de calcular la posición y la orientación del efector terminal de manipulador. Para poder enfrentar este problema es necesario utilizar herramientas como las matrices de rotación y traslación, las cuales permiten representar el movimiento de un sistema de referencia a otro.

En cuanto al caso concreto de la cinemática directa mediante Denavit Hartenberg, se trata de un método para describir la cinemática de un manipulador articulado. Este método fue desarrollado por Jacques Denavit y Richard Hartenberg en 1955. El método de Denavit-Hartenberg es un método de notación para describir los sistemas de coordenadas de los eslabones de un robot manipulador. Este método se basa en la asignación de un sistema de coordenadas a cada uno de los eslabones del robot, de tal forma que se pueda describir la posición y orientación de cada uno de los eslabones con respecto a su eslabón anterior. Para ello, se utiliza una serie de parámetros que se asignan a cada uno de los eslabones del robot, los cuales son:

- **Distancia** d_i: Distancia entre los ejes Z_{i-1} y Z_i a lo largo del eje X_i.
- **Ángulo** θ_i: Ángulo entre los ejes X_{i-1} y X_i medido alrededor del eje Z_{i-1}.
- **Distancia** a_i: Distancia entre los ejes X_{i-1} y X_i a lo largo del eje Z_{i-1}.
- **Ángulo** α_i: Ángulo entre los ejes Z_{i-1} y Z_i medido alrededor del eje X_i.
- **Ángulo** θ_i: Ángulo entre los ejes X_{i-1} y X_i medido alrededor del eje Z_{i-1}.

2.2. Explicación del código suministrado

El script en python suministrado para la implementación de la práctica de cinemática directa mediante Denavit Hartenberg se puede observar a continuación:

Listing 2.1: Script de Cinemática Directa

```
# Ejemplo:
# ./cdDH.py 30 45

import sys
from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# ***************************************************************************
```

```
# Declaración de funciones

def ramal(I,prev=[],base=0):
  # Convierte el robot a una secuencia de puntos para representar
  O = []
  if I:
    if isinstance(I[0][0],list):
      for j in range(len(I[0])):
        O.extend(ramal(I[0][j], prev, base or j < len(I[0])-1))
    else:
      O = [I[0]]
      O.extend(ramal(I[1:],I[0],base))
      if base:
        O.append(prev)
  return O

def muestra_robot(O,ef=[]):
  # Pinta en 3D
  OR = ramal(O)
  OT = np.array(OR).T
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  # Bounding box cúbico para simular el ratio de aspecto correcto
  max_range = np.array([OT[0].max()-OT[0].min()
                        ,OT[1].max()-OT[1].min()
                        ,OT[2].max()-OT[2].min()
                        ]).max()
  Xb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten()
     + 0.5*(OT[0].max()+OT[0].min()))
  Yb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten()
     + 0.5*(OT[1].max()+OT[1].min()))
  Zb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten()
     + 0.5*(OT[2].max()+OT[2].min()))
  for xb, yb, zb in zip(Xb, Yb, Zb):
     ax.plot([xb], [yb], [zb], 'w')
  ax.plot3D(OT[0],OT[1],OT[2],marker='s')
  ax.plot3D([0],[0],[0],marker='o',color='k',ms=10)
  if not ef:
    ef = OR[-1]
  ax.plot3D([ef[0]],[ef[1]],[ef[2]],marker='s',color='r')
  ax.set_xlabel('X')
  ax.set_ylabel('Y')
  ax.set_zlabel('Z')
  plt.show()
  return

def arbol_origenes(O,base=0,sufijo=''):
  # Da formato a los origenes de coordenadas para mostrarlos por pantalla
  if isinstance(O[0],list):
    for i in range(len(O)):
      if isinstance(O[i][0],list):
        for j in range(len(O[i])):
          arbol_origenes(O[i][j],i+base,sufijo+str(j+1))
      else:
        print('(O'+str(i+base)+sufijo+')0\t= '+str([round(j,3) for j in O[i]]))
  else:
```

```
    print('(0'+str(base)+sufijo+')0\t= '+str([round(j,3) for j in 0]))

def muestra_origenes(O,final=0):
  # Muestra los orígenes de coordenadas para cada articulación
  print('Orígenes de coordenadas:')
  arbol_origenes(O)
  if final:
    print('E.Final = '+str([round(j,3) for j in final]))

def matriz_T(d,theta,a,alpha):
  # Calcula la matriz T (ángulos de entrada en grados)
  th=theta*pi/180;
  al=alpha*pi/180;
  return [[cos(th), -sin(th)*cos(al),  sin(th)*sin(al), a*cos(th)]
         ,[sin(th),  cos(th)*cos(al), -sin(al)*cos(th), a*sin(th)]
         ,[      0,          sin(al),          cos(al),         d]
         ,[      0,                0,                0,         1]
         ]
# ************************************************************************


# Introducción de los valores de las articulaciones
nvar=2 # Número de variables
if len(sys.argv) != nvar+1:
  sys.exit('El número de articulaciones no es el correcto ('+str(nvar)+')')
p=[float(i) for i in sys.argv[1:nvar+1]]

# Parámetros D-H:
#        1    2
d  = [   0,   0]
th = [p[0],p[1]]
a  = [  10,   5]
al = [   0,   0]

# Orígenes para cada articulación
o00=[0,0,0,1]
o11=[0,0,0,1]
o22=[0,0,0,1]

# Cálculo matrices transformación
T01=matriz_T(d[0],th[0],a[0],al[0])
T12=matriz_T(d[1],th[1],a[1],al[1])
T02=np.dot(T01,T12)

# Transformación de cada articulación
o10 =np.dot(T01, o11).tolist()
o20 =np.dot(T02, o22).tolist()

# Mostrar resultado de la cinemática directa
muestra_origenes([o00,o10,o20])
muestra_robot   ([o00,o10,o20])
input()
```

Como se puede ver en el script adjunto anteriormente, realiza la cinemática directa de un robot con dos articulaciones rotativas (**para el caso del robot del script inicial**) . Utiliza la representación de Denavit-Hartenberg (D-H) para describir la geometría del robot y calcula las matrices de transformación

homogénea para cada articulación. Los ángulos de las articulaciones se introducen como argumentos de línea de comandos.

El script utiliza funciones para convertir la descripción del robot en una secuencia de puntos para su representación visual en 3D. La salida incluye los orígenes de coordenadas y la representación gráfica del robot utilizando la biblioteca *matplotlib*.

En términos generales, el script demuestra cómo determinar la posición y orientación final de un robot manipulador dados los ángulos de sus articulaciones. Los resultados de la cinemática directa se muestran tanto numéricamente como en una representación gráfica del robot en un entorno tridimensional.

2.3. Implementación de la Cinemática Directa

Tras la explicación del funcionamiento del script en *Python* adjunto en el apartado anterior, se procede a realizar la implementación de la cinemática directa mediante Denavit Hartenberg de una serie de robots manipuladores que se pueden observar en el pdf adjunto [5]. Pero, para no extender demasiado el documento, se va a tomar como ejemplo la implementación de la cinemática directa mediante Denavit Hartenberg del manipulador número *5* del pdf adjunto [5].

2.3.1. Representación gráfica del manipulador

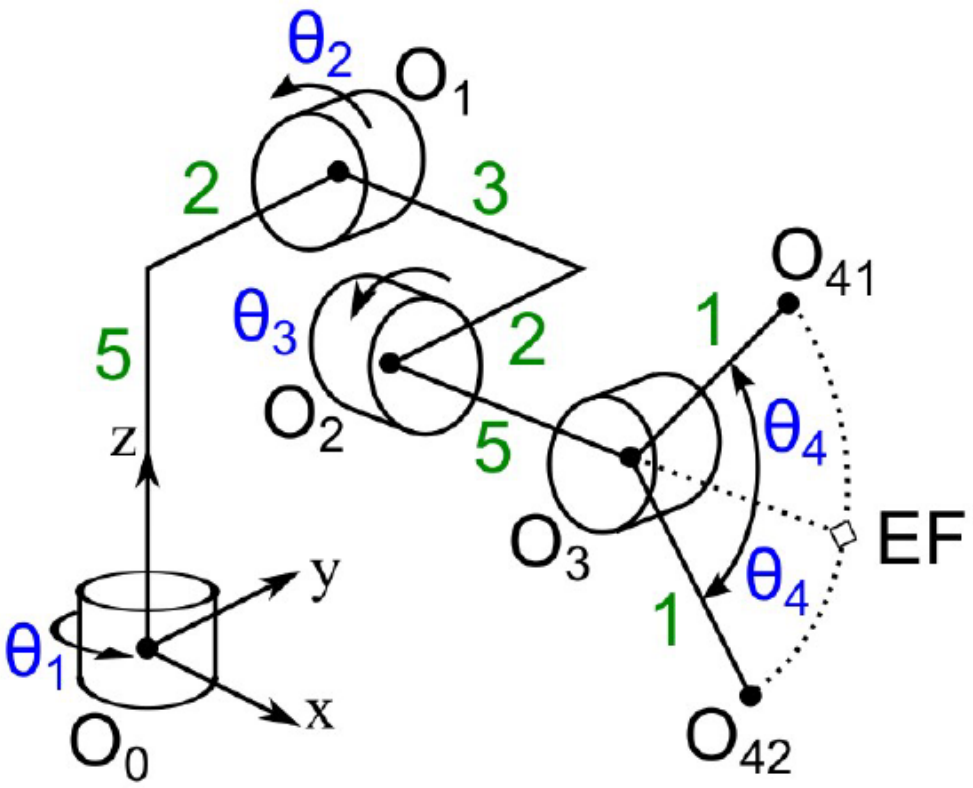

Figura 2.1: Representación gráfica del manipulador número 5

2.3.2. Implementación de la cinemática directa del robot número 5

Tras la visualización del robot número 5 en la figura 2.1, se procede a realizar la implementación de la cinemática directa del robot. Para ello, se va a utilizar el script en *Python* adjunto en el apartado anterior, modificando los parámetros de la tabla Denavit Hartenberg de la siguiente forma:

Cuadro 2.1: Parámetros D-H

Articulación	$\mathbf{d_i}$	$\theta_\mathbf{i}$	$\mathbf{a_i}$	$\alpha_\mathbf{i}$
05	5	$p[0]$	0	-90
1	2	0	0	0
15	0	$p[1]$	3	0
2	-2	-90	0	-90
3	5	$p[2]$	0	90
41	0	$-p[3]+90$	1	0
42	0	$p[3]+90$	1	0
ef	0	90	1	0

Una vez se tienen modificados los parámetros de la tabla, se procede a la modificación del número de manipuladores permitidos por el script, de tal manera que finalmente este toma el siguiente aspecto:

Listing 2.2: Script de Cinemática Directa del Manipulador 5

```
# Ejemplo:
# python3 cin_dir_5.py 10 20 30 30

# Cabe destacar que la ejecución de este script tiene la siguiente estructura de
    ejecución:
# python3 cin_dir_5.py <ángulo-o0> <ángulo-o2> <ángulo-o3> <ángulo-51-52>

import sys
from math import pi, cos, sin
import numpy as np
import matplotlib
# De esta manera se establece el motor gráfico que se debe de usar para MacOS
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# ******************************************************************************
# Declaración de funciones

def ramal(I,prev=[],base=0):
  # Convierte el robot a una secuencia de puntos para representar
  O = []
  if I:
    if isinstance(I[0][0],list):
      for j in range(len(I[0])):
        O.extend(ramal(I[0][j], prev, base or j < len(I[0])-1))
    else:
      O = [I[0]]
      O.extend(ramal(I[1:],I[0],base))
      if base:
        O.append(prev)
  return O
```

```
32
33 def muestra_robot(O,ef=[]):
34   # Pinta en 3D
35   OR = ramal(O)
36   OT = np.array(OR).T
37   fig = plt.figure()
38   ax = fig.add_subplot(111, projection='3d')
39   # Bounding box cúbico para simular el ratio de aspecto correcto
40   max_range = np.array([OT[0].max()-OT[0].min()
41                         ,OT[1].max()-OT[1].min()
42                         ,OT[2].max()-OT[2].min()
43                         ]).max()
44   Xb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten()
45      + 0.5*(OT[0].max()+OT[0].min()))
46   Yb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten()
47      + 0.5*(OT[1].max()+OT[1].min()))
48   Zb = (0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten()
49      + 0.5*(OT[2].max()+OT[2].min()))
50   for xb, yb, zb in zip(Xb, Yb, Zb):
51      ax.plot([xb], [yb], [zb], 'w')
52   ax.plot3D(OT[0],OT[1],OT[2],marker='s')
53   ax.plot3D([0],[0],[0],marker='o',color='k',ms=10)
54   if not ef:
55     ef = OR[-1]
56   ax.plot3D([ef[0]],[ef[1]],[ef[2]],marker='s',color='r')
57   ax.set_xlabel('X')
58   ax.set_ylabel('Y')
59   ax.set_zlabel('Z')
60   plt.show()
61   return
62
63 def arbol_origenes(O,base=0,sufijo=''):
64   # Da formato a los origenes de coordenadas para mostrarlos por pantalla
65   if isinstance(O[0],list):
66     for i in range(len(O)):
67       if isinstance(O[i][0],list):
68         for j in range(len(O[i])):
69           arbol_origenes(O[i][j],i+base,sufijo+str(j+1))
70       else:
71         print(('(O'+str(i+base)+sufijo+')0\t= '+str([round(j,3) for j in O[i]])))
72   else:
73     print(('(O'+str(base)+sufijo+')0\t= '+str([round(j,3) for j in O])))
74
75 def muestra_origenes(O,final=0):
76   # Muestra los orígenes de coordenadas para cada articulación
77   print('Orígenes de coordenadas:')
78   arbol_origenes(O)
79   if final:
80     print(('E.Final = '+str([round(j,3) for j in final])))
81
82 def matriz_T(d,theta,a,alpha):
83   # Calcula la matriz T (ángulos de entrada en grados)
84   th=theta*pi/180;
85   al=alpha*pi/180;
86   return [[cos(th), -sin(th)*cos(al),  sin(th)*sin(al), a*cos(th)]
87          ,[sin(th),  cos(th)*cos(al), -sin(al)*cos(th), a*sin(th)]
```

```
88          ,[      0,          sin(al),          cos(al),         d]
89          ,[      0,                0,                0,         1]
90          ]
91  # ************************************************************************
92
93  plt.ion() # Modo interactivo
94  # Introducción de los valores de las articulaciones
95  nvar=4 # Número de variables
96  if len(sys.argv) != nvar+1:
97    sys.exit('El número de articulaciones no es el correcto ('+str(nvar)+')')
98  p=[float(i) for i in sys.argv[1:nvar+1]]
99
100 # Parámetros D-H:
101 #        05      1      15          2       3          41          42          ef
102 d  = [   5,      2,     0,          -2,     5,         0,          0,          0
        ]
103 th = [   p[0],   0,     p[1],       -90,    p[2],      -p[3] + 90, p[3] + 90,  90
      ]
104 a  = [   0,      0,     3,          0,      0,         1,          1,          1
        ]
105 al = [   -90,    0,     0,          -90,    90,        0,          0,          0
        ]
106
107 # Orígenes para cada articulación
108 o00=[0,0,0,1]
109 o05=[0,0,0,1]
110 o11=[0,0,0,1]
111 o15=[0,0,0,1]
112 o22=[0,0,0,1]
113 o33=[0,0,0,1]
114 o441=[0,0,0,1]
115 o442=[0,0,0,1]
116 oeff=[0,0,0,1]
117
118 # Cálculo matrices transformación
119 T005 = matriz_T(d[0],th[0],a[0],al[0])
120 T051 = matriz_T(d[1],th[1],a[1],al[1])
121 T115 = matriz_T(d[2],th[2],a[2],al[2])
122 T225 = matriz_T(d[3],th[3],a[3],al[3])
123 T033 = matriz_T(d[4],th[4],a[4],al[4])
124 T041 = matriz_T(d[5],th[5],a[5],al[5])
125 T042 = matriz_T(d[6],th[6],a[6],al[6])
126 T0ef = matriz_T(d[7],th[7],a[7],al[7])
127
128
129 T02 = np.dot(T005, T051)
130 T03 = np.dot(T02, T115)
131 T04 = np.dot(T03, T225)
132 T05 = np.dot(T04, T033)
133 T06 = np.dot(T05, T041)
134 T07 = np.dot(T06, T042)
135 T0ef = np.dot(T07, T0ef)
136
137 # Transformación de cada articulación
138 o10 = np.dot(T005, o05).tolist()
139 o20 = np.dot(T02, o11).tolist()
```

```
o30 = np.dot(T03, o15).tolist()
o40 = np.dot(T04, o22).tolist()
o50 = np.dot(T05, o33).tolist()
o60 = np.dot(T06, o441).tolist()
o70 = np.dot(T07, o442).tolist()
oeff = np.dot(T0ef, oeff).tolist()

# Mostrar resultado de la cinemática directa
muestra_origenes([o00,o10, o20, o30, o40, o50, [o60], [o70]]], oeff)
muestra_robot   ([o00,o10, o20, o30, o40, o50, [o60], [o70]]], oeff)
eval(input())

plt.show(block=True)
```

2.4. Ejecución del código

Para la ejecución del código, se debe ejecutar el siguiente comando en la terminal:

Listing 2.3: Ejecución del script de Cinemática Directa del Manipulador 5

```
$  python3 Manipulador-5.py 10 20 30 30
```

Tras esto, se obtiene el siguiente resultado:

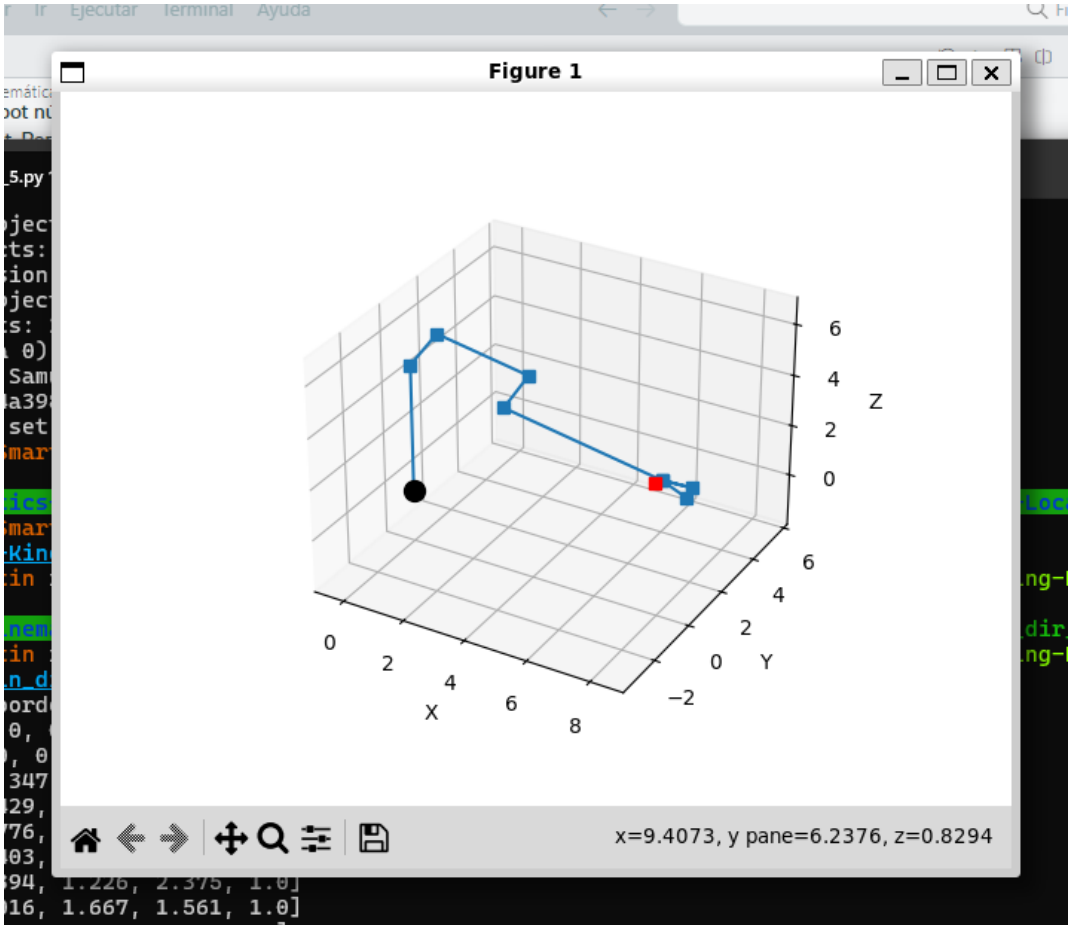

Figura 2.2: Resultado de la ejecución del script de Cinemática Directa del Manipulador 5

2.5. Parámetros modificados

Los parámetros modificados en la implementación de la cinemática directa del robot número 5 han sido los siguientes:

- **Distancia** d_i: Se ha modificado el valor de la distancia d_i de la articulación 05, de tal manera que se ha cambiado de 0 a 5.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 05, de tal manera que se ha cambiado de 0 a $p[0]$.
- **Ángulo** α_i: Se ha modificado el valor del ángulo α_i de la articulación 05, de tal manera que se ha cambiado de 0 a -90.
- **Distancia** d_i: Se ha modificado el valor de la distancia d_i de la articulación 1, de tal manera que se ha cambiado de 0 a 2.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 1, de tal manera que se ha cambiado de 0 a 0.
- **Distancia** a_i: Se ha modificado el valor de la distancia a_i de la articulación 15, de tal manera que se ha cambiado de 0 a 3.
- **Ángulo** α_i: Se ha modificado el valor del ángulo α_i de la articulación 15, de tal manera que se ha cambiado de 0 a 0.
- **Distancia** d_i: Se ha modificado el valor de la distancia d_i de la articulación 2, de tal manera que se ha cambiado de 0 a -2.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 2, de tal manera que se ha cambiado de 0 a -90.
- **Ángulo** α_i: Se ha modificado el valor del ángulo α_i de la articulación 2, de tal manera que se ha cambiado de 0 a -90.
- **Distancia** d_i: Se ha modificado el valor de la distancia d_i de la articulación 3, de tal manera que se ha cambiado de 0 a 5.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 3, de tal manera que se ha cambiado de 0 a $p[2]$.
- **Ángulo** α_i: Se ha modificado el valor del ángulo α_i de la articulación 3, de tal manera que se ha cambiado de 0 a 90.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 41, de tal manera que se ha cambiado de 0 a $-p[3] + 90$.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación 42, de tal manera que se ha cambiado de 0 a $p[3] + 90$.
- **Ángulo** θ_i: Se ha modificado el valor del ángulo θ_i de la articulación ef, de tal manera que se ha cambiado de 0 a 90.
- **Ángulo** α_i: Se ha modificado el valor del ángulo α_i de la articulación ef, de tal manera que se ha cambiado de 0 a 0.
- **Distancia** a_i: Se ha modificado el valor de la distancia a_i de la articulación ef, de tal manera que se ha cambiado de 0 a 1.

Como se puede observar, dependiendo del manipulador que quiera ser implementado, se deben modificar los parámetros de la tabla Denavit Hartenberg de una forma u otra. Es decir, de manera previa, se deben conocer los parámetros de la tabla Denavit Hartenberg del manipulador que se quiere implementar, o en su defecto, como para el caso de la práctica implementada y su modificación, estos valores deben de ser calculados previamente.

2.6. Mejoras en la implementación

Tras la explicación del script de cinemática directa suministrado, se pueden tomar como mejoras de esta práctica pues el hecho de la visualización final del manipulador, es decir, se podría realizar la implementación de la visualización del manipulador en otros tipos de entornos gráficos, como por ejemplo, *Blender*. Es decir, se podría realizar la implementación de la visualización del manipulador en un entorno gráfico más realista, en el cual se podría observar el manipulador en un entorno tridimensional más realista, de tal manera que el alumnado pueda comprender de manera mucho más sencilla y visual el funcionamiento de la cinemática directa mediante Denavit Hartenberg.

Con todo esto comentado, se podría realizar la implementación de la práctica en python de la misma manera que hasta el momento, pero tras la explicación de la matriz de Denavit Hartenberg, se podría trasladar la representación gráfica de python mediante el empleo de la biblioteca *matplotlib* a un entorno gráfico más realista como *Blender*, haciendo uso de librerías como *bpy* [6].

2.7. Conclusiones

Para concluir con la práctica de Cinemática Directa mediante Denavit Hartenberg, se puede decir que se ha conseguido realizar la implementación de la cinemática directa de un robot manipulador mediante el empleo de la representación de Denavit Hartenberg. Además, se ha conseguido realizar la implementación de la visualización del robot manipulador en un entorno gráfico mediante el empleo de la biblioteca *matplotlib*. Pero, como persona a la que le gusta el mundo de la robótica, me hubiera gustado realizar la implementación de la visualización del robot manipulador en un entorno gráfico más realista como *Blender*, o poder ver su funcionamiento dentro de un manipulador real, ya sea como explicación de la práctica por parte del profesorado, pero, me faltó como esa emoción de poder ver el resultado de la práctica en la realidad.

Capítulo 3 Cinemática inversa

3.1. ¿Qué es la Cinemática Inversa?

Mientras que la cinemática directa se ocupa de calcular la posición y orientación del extremo del robot dado un conjunto de parámetros de articulación, la *cinemática inversa* implica encontrar las configuraciones de las articulaciones necesarias para alcanzar una posición y orientación del extremo. La cinemática inversa es un problema más difícil que la cinemática directa, ya que puede haber más de una solución posible, o ninguna. Además, la cinemática inversa puede ser más difícil de calcular que la cinemática directa, ya que puede requerir la resolución de una ecuación no lineal.

La cinemática inversa es esencial en la programación de robots ya que permite determinar cómo deben moverse las articulaciones para lograr una tarea específica.

3.2. Explicación del código suministrado

El script en python suministrado para la implementación de la práctica de cinemática inversa se puede observar a continuación:

Listing 3.1: Script de Cinemática Inversa

```
import sys
from math import *
import numpy as np
import matplotlib.pyplot as plt
import colorsys as cs

# ******************************************************************************
# Declaración de funciones

def muestra_origenes(O,final=0):
  # Muestra los orígenes de coordenadas para cada articulación
  print('Origenes de coordenadas:')
  for i in range(len(O)):
    print('(O'+str(i)+')0\t= '+str([round(j,3) for j in O[i]]))
  if final:
    print('E.Final = '+str([round(j,3) for j in final]))

def muestra_robot(O,obj):
  # Muestra el robot graficamente
  plt.figure()
  plt.xlim(-L,L)
  plt.ylim(-L,L)
  T = [np.array(o).T.tolist() for o in O]
  for i in range(len(T)):
    plt.plot(T[i][0], T[i][1], '-o', color=cs.hsv_to_rgb(i/float(len(T)),1,1))
  plt.plot(obj[0], obj[1], '*')
  plt.pause(0.0001)
  plt.show()

```

```
#  input()
  plt.close()

def matriz_T(d,th,a,al):

  return [[cos(th), -sin(th)*cos(al),  sin(th)*sin(al), a*cos(th)]
         ,[sin(th),  cos(th)*cos(al), -sin(al)*cos(th), a*sin(th)]
         ,[      0,          sin(al),          cos(al),         d]
         ,[      0,                0,                0,         1]
         ]

def cin_dir(th,a):
  #Sea 'th' el vector de thetas
  #Sea 'a'  el vector de longitudes
  T = np.identity(4)
  o = [[0,0]]
  for i in range(len(th)):
    T = np.dot(T,matriz_T(0,th[i],a[i],0))
    tmp=np.dot(T,[0,0,0,1])
    o.append([tmp[0],tmp[1]])
  return o

# ****************************************************************************
# Cálculo de la cinemática inversa de forma iterativa por el método CCD

# valores articulares arbitrarios para la cinemática directa inicial
th=[0.,0.,0.]
a =[5.,5.,5.]
L = sum(a) # variable para representación gráfica
EPSILON = .01

#plt.ion() # modo interactivo

# introducción del punto para la cinemática inversa
if len(sys.argv) != 3:
  sys.exit("python " + sys.argv[0] + " x y")
objetivo=[float(i) for i in sys.argv[1:]]
O=cin_dir(th,a)
#O=zeros(len(th)+1) # Reservamos estructura en memoria
 # Calculamos la posicion inicial
print ("- Posicion inicial:")
muestra_origenes(O)

dist = float("inf")
prev = 0.
iteracion = 1
while (dist > EPSILON and abs(prev-dist) > EPSILON/100.):
  prev = dist
  O=[cin_dir(th,a)]
  # Para cada combinación de articulaciones:
  for i in range(len(th)):
    # cálculo de la cinemática inversa:

    O.append(cin_dir(th,a))

  dist = np.linalg.norm(np.subtract(objetivo,O[-1][-1]))
```

```
86    print ("\n- Iteracion " + str(iteracion) + ':')
87    muestra_origenes(O[-1])
88    muestra_robot(O,objetivo)
89    print ("Distancia al objetivo = " + str(round(dist,5)))
90    iteracion+=1
91    O[0]=O[-1]
92
93  if dist <= EPSILON:
94    print ("\n" + str(iteracion) + " iteraciones para converger.")
95  else:
96    print ("\nNo hay convergencia tras " + str(iteracion) + " iteraciones.")
97  print ("- Umbral de convergencia epsilon: " + str(EPSILON))
98  print ("- Distancia al objetivo:          " + str(round(dist,5)))
99  print ("- Valores finales de las articulaciones:")
100 for i in range(len(th)):
101   print ("  theta" + str(i+1) + " = " + str(round(th[i],3)))
102 for i in range(len(th)):
103   print ("  L" + str(i+1) + "     = " + str(round(a[i],3)))
```

Como se puede ver en el script adjunto anteriormente, se realiza la implementación de un script genérico que permite realizar el cálculo de la cinemática inversa de un robot mediante el método de Cierre de Cadena Cinemática (CCD). El método CCD es un algoritmo iterativo que comienza con una estimación inicial de la configuración de las articulaciones y luego mejora la estimación en cada iteración. El algoritmo CCD se puede resumir en los siguientes pasos:

1. Seleccionar un punto de destino para el extremo del robot.
2. Para cada articulación, comenzando con la articulación más alejada del extremo del robot, calcule el vector desde la articulación hasta el punto de destino.
3. Calcule el vector desde la articulación hasta el extremo del robot.
4. Calcule el ángulo entre estos dos vectores.
5. Gire la articulación por este ángulo.
6. Repita los pasos 2 a 5 hasta que el extremo del robot esté lo suficientemente cerca del punto de destino.
7. Repita los pasos 1 a 6 para cada punto de destino.
8. Repita los pasos 1 a 7 hasta que el robot alcance su posición final.
9. Repita los pasos 1 a 8 para cada posición final.
10. Repita los pasos 1 a 9 hasta que el robot complete su trayectoria.

Después de definir el algoritmo CCD, se determinan las funciones para mostrar orígenes de coordenadas y representar gráficamente el robot, el programa establece valores iniciales arbitrarios para las articulaciones y longitudes de eslabones. El usuario proporciona las coordenadas del punto objetivo, y utilizando la cinemática directa se calcula la posición inicial y se inicia un bucle iterativo para ajustar las articulaciones de manera que el extremo del robot se acerque al objetivo. En cada iteración, se muestra la posición actual, los orígenes de coordenadas y la distancia al objetivo. Las iteraciones continúan hasta que se alcanza un umbral de convergencia o se supera un límite predefinido de iteraciones.

Finalmente, se presentan los resultados, incluyendo la cantidad de iteraciones necesarias, el umbral de convergencia, la distancia al objetivo y lo valores finales de las articulaciones.

3.3. Implementación de la Cinemática Inversa

Tras la explicación del funcionamiento del script de *cinemática inversa* proporcionado, se realiza la implementación de esta de la siguiente manera:

Listing 3.2: Script de Cinemática Inversa implementado de manera correcta

```
import colorsys as cs
import matplotlib.pyplot as plt
import sys
from math import *
import numpy as np
import matplotlib
matplotlib.use('TkAgg')

# ******************************************************************************
# Declaración de funciones

def muestra_origenes(O, final=0):
    # Muestra los orígenes de coordenadas para cada articulación
    print('Origenes de coordenadas:')
    for i in range(len(O)):
        print('(O'+str(i)+')0\t= '+str([round(j, 3) for j in O[i]]))
    if final:
        print('E.Final = '+str([round(j, 3) for j in final]))


def muestra_robot(O, obj):
    # Muestra el robot graficamente
    plt.figure(1)
    plt.xlim(-L, L)
    plt.ylim(-L, L)
    T = [np.array(o).T.tolist() for o in O]
    for i in range(len(T)):
        plt.plot(T[i][0], T[i][1], '-o',
                 color=cs.hsv_to_rgb(i/float(len(T)), 1, 1))
    plt.plot(obj[0], obj[1], '*')
    plt.show()
    input("Continuar...")
    plt.close()

def matriz_T(d, th, a, al):
    # Calcula la matriz T (ángulos de entrada en grados)

    return [[cos(th), -sin(th)*cos(al),  sin(th)*sin(al), a*cos(th)], [sin(th),
       cos(th)*cos(al), -sin(al)*cos(th), a*sin(th)], [0,          sin(al),
                cos(al),         d], [0,                0,                0,
               1]
           ]

def cin_dir(th, a):
    # Sea 'th' el vector de thetas
    # Sea 'a'  el vector de longitudes
    T = np.identity(4)
    o = [[0, 0]]
    for i in range(len(th)):
        T = np.dot(T, matriz_T(0, th[i], a[i], 0))
```

```
48          tmp = np.dot(T, [0, 0, 0, 1])
49          o.append([tmp[0], tmp[1]])
50      return o
51
52  def read_input_file(file_name):
53      with open(file_name, 'r') as f:
54          lines = f.readlines()
55      ## Comprobación de errores en la estructura del fichero introducido
56      if len(lines) != 5:
57          sys.exit("El fichero de entrada debe de tener 5 líneas.")
58      type_arm = [int(i) for i in lines[0].split()]
59      th = [float(i) for i in lines[1].split()]
60      a = [float(i) for i in lines[2].split()]
61      upper_limit = [float(i) for i in lines[3].split()]
62      lower_limit = [float(i) for i in lines[4].split()]
63      return type_arm, th, a, upper_limit, lower_limit
64
65  # ******************************************************************************
66  # Cálculo de la cinemática inversa de forma iterativa por el método CCD
67  plt.ion()  # modo interactivo
68
69  # introducción del punto para la cinemática inversa
70  if sys.argv[1] == "-h" or sys.argv[1] == "--help":
71      print("<< Bienvenido al programa de ayuda de cinemática inversa >>")
72      print("Para ejecutar el programa, debe de seguir la siguiente estructura de
            ejecución:")
73      print("python ccdp3.py << FileName >> << x >> << y >>")
74      print("Donde FileName es el nombre del fichero de entrada")
75      print("y x e y son las coordenadas del punto al que se quiere llegar")
76      print("El fichero de entrada debe de tener la siguiente estructura (CADA VALOR
            SEPARADO POR ESPACIOS):")
77      print("<< Tipo de articulación >> (0 para rotacional, 1 para prismática)")
78      print("<< Theta >>")
79      print("<< a, Longitud >>")
80      print("<< Límite superior >>")
81      print("<< Límite inferior >>")
82      print("Ejemplo:")
83      print("=====================================================")
84      print("0 0 1 1")
85      print("10. 20. 0. 0.")
86      print("1. 1. 1. 1.")
87      print("90 90 0 0")
88      print("-90 -90 0 0")
89      print("=====================================================")
90      sys.exit()
91
92  if len(sys.argv) != 4:
93      sys.exit("python " + sys.argv[0] + " FileName" + " x y")
94  objetivo = [float(i) for i in sys.argv[2:]]
95
96  # Lectura del fichero de entrada
97  file_name = sys.argv[1]
98  type_arm, th, a, Upper_limit, Lower_limit = read_input_file(file_name)
99  L = sum(a)
100 EPSILON = .01 # umbral de convergencia
101
```

```
O = list(range(len(th)+1))  # Reservamos estructura en memoria
O[0] = cin_dir(th, a)  # Calculamos la posicion inicial
print("- Posicion inicial:")
muestra_origenes(O[0])

dist = float("inf")
prev = 0.
iteracion = 1
while (dist > EPSILON and abs(prev-dist) > EPSILON/100.):
    prev = dist
    # Para cada combinación de articulaciones:
    for i in range(len(th)):
        # cálculo de la cinemática inversa:
        current = len(th)-i-1
        if (type_arm[current] == 1): # Si es prismática
            w = np.sum(th[:current+1]) # Suma acumulativa de los ángulos anteriores
            u = [np.cos(w), np.sin(w)] # Vector unitario u
            d = np.dot(u, np.subtract(objetivo, O[i][-1])) # Distancia a recorrer
            a[current] += d # Se suma la distancia a recorrer
            # Se realiza una normalización de la longitud
            if (a[current] > Upper_limit[current]):
                a[current] = Upper_limit[current]
            if (a[current] < Lower_limit[current]):
                a[current] = Lower_limit[current]
        else: # Si es rotacional
            v1 = np.subtract(objetivo, O[i][current]) # Vector objetivo
            v2 = np.subtract(O[i][-1], O[i][current]) # Vector actual
            c1 = atan2(v1[1], v1[0]) # Ángulo asociado al vector v1
            c2 = atan2(v2[1], v2[0]) # Ángulo asociado al vector v2
            th[current] += c1 - c2 # Ajuste de ángulo para acercarse al objetivo
               final
            while th[current] > pi: # Se realiza una normalización del ángulo
                th[current] -= 2*pi
            while th[current] < -pi: # Se realiza una normalización del ángulo
                th[current] += 2*pi
            # Se realiza una normalización del ángulo
            if (th[current] > Upper_limit[current]):
                th[current] = Upper_limit[current]
            if (th[current] < Lower_limit[current]):
                th[current] = Lower_limit[current]
        O[i+1] = cin_dir(th, a)

    dist = np.linalg.norm(np.subtract(objetivo, O[-1][-1]))
    print("\n- Iteracion " + str(iteracion) + ':')
    muestra_origenes(O[-1])
    muestra_robot(O, objetivo)
    print("Distancia al objetivo = " + str(round(dist, 5)))
    iteracion += 1
    O[0] = O[-1]

if dist <= EPSILON:
    print("\n" + str(iteracion) + " iteraciones para converger.")
else:
    print("\nNo hay convergencia tras " + str(iteracion) + " iteraciones.")
print("- Umbral de convergencia epsilon: " + str(EPSILON))
print("- Distancia al objetivo:          " + str(round(dist, 5)))
```

```
157 print("- Valores finales de las articulaciones:")
158 for i in range(len(th)):
159     print("  theta" + str(i+1) + " = " + str(round(th[i], 3)))
160 for i in range(len(th)):
161     print("  L" + str(i+1) + "     = " + str(round(a[i], 3)))
162
163 plt.show(block=True)
```

Para comenzar con los cambios realizados dentro del script que se puede observar anteriormente, se realiza la implementación de la cinemática inversa donde se presenta un bucle iterativo basado en el método de Cierre de Cadena Cinemática (CCD) para calcular la cinemática inversa. El bucle continúa hasta que la distancia entre la posición actual del extremo del robot y el objetivo sea menor que un umbral de convergencia (EPSILON). Durante cada iteración, se ajustan las articulaciones del robot para acercar el extremo al objetivo.

El proceso iterativo incluye las siguientes etapas:

1. Se almacena la distancia actual como referencia (prev) para evaluar la convergencia.
2. Para cada articulación, se calcula la cinemática inversa, tomando en cuenta si la articulación es prismática o rotacional.
   a) En el caso de una articulación prismática, se ajusta la longitud del eslabón y se normaliza si excede los límites.
   b) Para articulaciones rotacionales, se determina el ángulo de ajuste necesario y se normaliza si es necesario.
3. Se actualiza la posición del robot después de cada ajuste.

Tras la implementación de la cinemática inversa, se procede a la inclusión de una función nueva dentro del script, denominada como *read_input_file*, la cual permite la lectura de un fichero de texto, en el cual se encuentran los valores de los distintos parámetros como el tipo de brazo, sus ángulos, los limites superiores, los límites inferiores. Un ejemplo de este fichero de texto se puede observar a continuación:

Listing 3.3: Ejemplo de fichero de entrada

```
1 0 1 0
2 50. 0. 20.
3 5. 5. 5.
4 90 0 90
5 -90 0 -90
```

Finalmente como último cambio realizado al script de cinemática inversa, se realiza la implementación de un ***menú de ayuda*** que permite al usuario conocer los parámetros que se deben introducir para la ejecución del script, junto con la forma de ejecutar el script.

3.4. Ejecución del código

Tras la explicación de la implementación de la cinemática inversa, se procede a la ejecución del script para la comprobación del funcionamiento del mismo. Para ello, se ejecuta el siguiente comando en la terminal:

Listing 3.4: Ejecución del script de Cinemática Inversa

```
1   $  python3 Inverse-Kinematics-result.py  input-file.txt 15 -10
```

Tras esto, se obtiene el siguiente resultado:

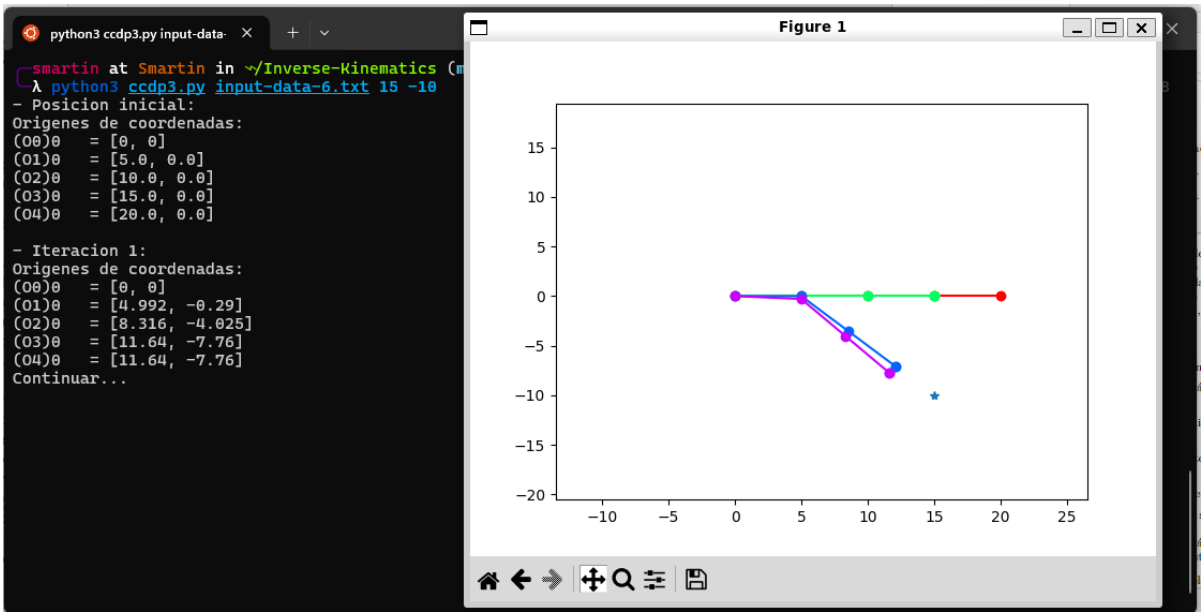

Figura 3.1: Resultado de la ejecución del script de Cinemática Inversa

3.5. Parámetros modificados

Para la implementación de cinemática inversa, se han realizado numerosas pruebas con los parámetros que existen dentro de los ficheros de entrada del programa, es por ello que se han realizado numerosas pruebas con los cambios que se producirían si por ejemplo si se realizan cambios en el tipo de articulación que existe en el brazo, es decir, 0 si se trata de una articulación rotacional y 1 si se trata de una articulación prismática.

Además, se han realizado pruebas con los cambios que se producirían si se modifican los valores en los *límites superiores* e *inferiores* de las articulaciones, es decir, si se cambian los valores de los límites superiores e inferiores de las articulaciones, se producen cambios en el resultado final de la cinemática inversa. Por último, se han realizado pruebas con los ángulos de las articulaciones, es decir, con los valores de *theta* o finalmente con la *longitud* de los distintos eslabones que componen el brazo.

3.6. Conclusiones

Para concluir, se puede decir que se ha conseguido realizar la implementación de la cinemática inversa de un robot manipulador mediante el empleo del método de Cierre de Cadena Cinemática (CCD). Esta práctica me ha resultado personalmente más interesante en cuanto a su desarrollo y en cuanto a su ejecución, ya que el hecho de poder ver como dependiendo de los valores que se le introduzcan a los distintos eslabones del brazo, se pueda alcanzar el objetivo final de manera más rápido o más lenta dependiendo de los valores que se le introduzcan, me ha resultado muy interesante. Además, cabe destacar que en cuanto a esta práctica no se me ha ocurrido ninguna mejora, ya que la forma de ver como se desplaza el brazo mediante la librería de *matplotlib*, me parece que de esta manera es mucho más cómoda y sencilla de entender y comprender el funcionamiento de la cinemática inversa.

Capítulo 4 Localización

4.1. ¿Qué es la localización?

La localización es el problema de estimar el estado de un robot en un entorno desconocido. El estado de un robot incluye su posición y orientación, así como cualquier otra información que pueda ser útil para determinar su comportamiento futuro. La localización es un problema difícil porque el entorno es desconocido y el robot no tiene acceso a información perfecta sobre su estado. En cambio, el robot debe utilizar información parcial y ruidosa para estimar su estado.

La localización de robots móviles consiste en determinar la posición del robot en relación a un mapa dado del entorno. El problema principal de la localización de un robot móvil reside en que su posición no puede ser medida directamente y debe ser inferida de datos sensoriales. Además, una única medición suele ser de manera habitual insuficiente y el robot debe integrar datos a lo largo del tiempo para determinar su posición.

En el caso de la práctica solicitada por la asignatura de *Robótica Computacional*, el robot móvil no tiene datos sensoriales, sino que se establecen una serie de *"balizas"* en el entorno, las cuales permiten al robot móvil conocer su posición en el entorno, además, se hace uso de otro robot móvil denominado como *robot rea;"*, el cual se encarga de marcar la diferenciación con respecto al robot móvil al que se debe de aplicar la localización cada vez que este se desvíe o se pierda.

4.2. Explicación del código suministrado

El código suministrado para la implementación de la práctica de localización se puede observar a continuación en dos ficheros distintos, el primero de ellos referente a los robots móviles:

Listing 4.1: Script de robot móvil para localización

```
from math import *
import random
import numpy as np
import copy

class robot:
  def __init__(self):
    self.x             = 0.
    self.y             = 0.
    self.orientation   = 0.
    self.forward_noise = 0.
    self.turn_noise    = 0.
    self.sense_noise   = 0.
    self.weight        = 1.
    self.old_weight    = 1.
    self.size          = 1.

  def copy(self):
    return copy.deepcopy(self)

  def set(self, new_x, new_y, new_orientation):
```

```
22      self.x = float(new_x)
23      self.y = float(new_y)
24      self.orientation = float(new_orientation)
25      while self.orientation >  pi: self.orientation -= 2*pi
26      while self.orientation < -pi: self.orientation += 2*pi
27
28    def set_noise(self, new_f_noise, new_t_noise, new_s_noise):
29      self.forward_noise = float(new_f_noise);
30      self.turn_noise    = float(new_t_noise);
31      self.sense_noise   = float(new_s_noise);
32
33    def pose(self):
34      return [self.x, self.y, self.orientation]
35
36    def sense1(self,landmark,noise):
37      return np.linalg.norm(np.subtract([self.x,self.y],landmark)) \
38                                          + random.gauss(0.,noise)
39
40    def sense(self, landmarks):
41      d = [self.sense1(l,self.sense_noise) for l in landmarks]
42      d.append(self.orientation + random.gauss(0.,self.sense_noise))
43      return d
44
45    def move(self, turn, forward):
46      self.orientation += float(turn) + random.gauss(0., self.turn_noise)
47      while self.orientation >  pi: self.orientation -= 2*pi
48      while self.orientation < -pi: self.orientation += 2*pi
49      dist = float(forward) + random.gauss(0., self.forward_noise)
50      self.x += cos(self.orientation) * dist
51      self.y += sin(self.orientation) * dist
52
53    def move_triciclo(self, turn, forward, largo):
54      dist = float(forward) + random.gauss(0., self.forward_noise)
55      self.orientation += dist * tan(float(turn)) / largo\
56                + random.gauss(0.0, self.turn_noise)
57      while self.orientation >  pi: self.orientation -= 2*pi
58      while self.orientation < -pi: self.orientation += 2*pi
59      self.x += cos(self.orientation) * dist
60      self.y += sin(self.orientation) * dist
61
62    def Gaussian(self, mu, sigma, x):
63      if sigma:
64        return exp(-(((mu-x)/sigma)**2)/2)/(sigma*sqrt(2*pi))
65      else:
66        return 0
67
68
69    def measurement_prob(self, measurements, landmarks):
70      self.weight = 0.
71      n=0
72      for i in range(len(measurements)-1):
73        self.weight += abs(self.sense1(landmarks[i],0) -measurements[i])
74        n=n+1
75      diff = self.orientation - measurements[-1]
76      while diff >  pi: diff -= 2*pi
77      while diff < -pi: diff += 2*pi
```

```
78      self.weight = self.weight + abs(diff)
79      self.weight=self.weight/(n+1)
80      return self.weight
81
82    def __repr__(self):
83      return '[x=%.6s y=%.6s orient=%.6s]' % \
84              (str(self.x), str(self.y), str(self.orientation))
```

En cuanto al segundo de los scripts, se trata del script referente a los cálculos de las distintas operaciones necesarias para la realización de la localización:

Listing 4.2: Script de localización

```
1  import sys
2  from math import *
3  from robot import robot
4  import random
5  import numpy as np
6  import matplotlib.pyplot as plt
7  from datetime import datetime
8
9  def distancia(a,b):
10   return np.linalg.norm(np.subtract(a[:2],b[:2]))
11
12 def angulo_rel(pose,p):
13   w = atan2(p[1]-pose[1],p[0]-pose[0])-pose[2]
14   while w >  pi: w -= 2*pi
15   while w < -pi: w += 2*pi
16   return w
17
18 def mostrar(objetivos,ideal,trayectoria):
19   objT   = np.array(objetivos).T.tolist()
20   trayT  = np.array(trayectoria).T.tolist()
21   ideT   = np.array(ideal).T.tolist()
22   bordes = [min(trayT[0]+objT[0]+ideT[0]),max(trayT[0]+objT[0]+ideT[0]),
23             min(trayT[1]+objT[1]+ideT[1]),max(trayT[1]+objT[1]+ideT[1])]
24   centro = [(bordes[0]+bordes[1])/2.,(bordes[2]+bordes[3])/2.]
25   radio  = max(bordes[1]-bordes[0],bordes[3]-bordes[2])*.75
26   plt.xlim(centro[0]-radio,centro[0]+radio)
27   plt.ylim(centro[1]-radio,centro[1]+radio)
28   # Representar objetivos y trayectoria
29   idealT = np.array(ideal).T.tolist()
30   plt.plot(idealT[0],idealT[1],'-g')
31   plt.plot(trayectoria[0][0],trayectoria[0][1],'or')
32   r = radio * .1
33   for p in trayectoria:
34     plt.plot([p[0],p[0]+r*cos(p[2])],[p[1],p[1]+r*sin(p[2])],'-r')
35     #plt.plot(p[0],p[1],'or')
36   objT   = np.array(objetivos).T.tolist()
37   plt.plot(objT[0],objT[1],'-.o')
38   plt.show()
39   input()
40   plt.clf()
41
42 def localizacion(balizas, real, ideal, centro, radio, mostrar=0):
43   # Buscar la  localizacin  ms probable del robot, a partir de su sistema
44   # sensorial, dentro de una  regin cuadrada de centro "centro" y lado "2*radio".
```

```
45
46    if mostrar:
47      plt.xlim(centro[0]-radio,centro[0]+radio)
48      plt.ylim(centro[1]-radio,centro[1]+radio)
49      imagen.reverse()
50      plt.imshow(imagen,extent=[centro[0]-radio,centro[0]+radio,\
51                                centro[1]-radio,centro[1]+radio])
52      balT = np.array(balizas).T.tolist();
53      plt.plot(balT[0],balT[1],'or',ms=10)
54      plt.plot(ideal.x,ideal.y,'D',c='#ff00ff',ms=10,mew=2)
55      plt.plot(real.x, real.y, 'D',c='#00ff00',ms=10,mew=2)
56      plt.show()
57      input()
58      plt.clf()
59
60  P_INICIAL = [0.,4.,0.] # Pose inicial ( posicin y orientacion)
61  V_LINEAL  = .7         # Velocidad lineal    (m/s)
62  V_ANGULAR = 140.       # Velocidad angular    (/s)
63  FPS       = 10.        #  Resolucin temporal (fps)
64
65  HOLONOMICO = 1
66  GIROPARADO = 0
67  LONGITUD   = .2
68
69  trayectorias = [
70      [[1,3]],
71      [[0,2],[4,2]],
72      [[2,4],[4,0],[0,0]],
73      [[2,4],[2,0],[0,2],[4,2]],
74      [[2+2*sin(.8*pi*i),2+2*cos(.8*pi*i)] for i in range(5)]
75      ]
76
77  if len(sys.argv)<2 or int(sys.argv[1])<0 or int(sys.argv[1])>=len(trayectorias):
78    sys.exit(sys.argv[0]+" <indice entre 0 y "+str(len(trayectorias)-1)+">")
79  objetivos = trayectorias[int(sys.argv[1])]
80
81  EPSILON = .1                # Umbral de distancia
82  V = V_LINEAL/FPS            # Metros por fotograma
83  W = V_ANGULAR*pi/(180*FPS)  # Radianes por fotograma
84
85  ideal = robot()
86  ideal.set_noise(0,0,.1)   # Ruido lineal / radial / de sensado
87  ideal.set(*P_INICIAL)     # operador 'splat'
88
89  real = robot()
90  real.set_noise(.01,.01,.1)  # Ruido lineal / radial / de sensado
91  real.set(*P_INICIAL)
92
93  random.seed(0)
94  tray_ideal = [ideal.pose()]  # Trayectoria percibida
95  tray_real = [real.pose()]     # Trayectoria seguida
96
97  tiempo  = 0.
98  espacio = 0.
99  random.seed(datetime.now())
100 for punto in objetivos:
```

```
  while distancia(tray_ideal[-1],punto) > EPSILON and len(tray_ideal) <= 1000:
    pose = ideal.pose()

    w = angulo_rel(pose,punto)
    if w > W:  w =  W
    if w < -W: w = -W
    v = distancia(pose,punto)
    if (v > V): v = V
    if (v < 0): v = 0

    if HOLONOMICO:
      if GIROPARADO and abs(w) > .01:
        v = 0
      ideal.move(w,v)
      real.move(w,v)
    else:
      ideal.move_triciclo(w,v,LONGITUD)
      real.move_triciclo(w,v,LONGITUD)
    tray_ideal.append(ideal.pose())
    tray_real.append(real.pose())

    espacio += v
    tiempo  += 1

if len(tray_ideal) > 1000:
  print ("<!> Trayectoria muy larga - puede que no se haya alcanzado la posicion
      final.")
print ("Recorrido: "+str(round(espacio,3))+"m / "+str(tiempo/FPS)+"s")
print ("Distancia real al objetivo: "+\
    str(round(distancia(tray_real[-1],objetivos[-1]),3))+"m")
mostrar(objetivos,tray_ideal,tray_real)  # Representacin grfica
```

Teniendo en cuenta los script adjuntos anteriormente, este, simula el movimiento de un robot en un entorno bidimensional y visualiza tanto la trayectoria planificada como la trayectoria real seguida por el robot. A continuación, se proporciona una explicación detallada de su funcionamiento:

El script comienza importando módulos esenciales, como sys para la manipulación de variables del intérprete, math para funciones matemáticas, y un módulo llamado robot que contiene la definición de la clase robot.

Se definen varias funciones útiles, como distancia para calcular la distancia euclidiana entre dos puntos, angulo_rel para determinar el ángulo relativo entre la posición actual del robot y un punto objetivo, y mostrar para visualizar gráficamente las trayectorias ideales y reales del robot.

A continuación, se establecen constantes como la pose inicial del robot (P_INICIAL), la velocidad lineal (V_LINEAL), la velocidad angular (V_ANGULAR), la resolución temporal en fotogramas por segundo (FPS), y parámetros que afectan el comportamiento del robot.

Se define una serie de trayectorias predefinidas en la lista trayectorias, y el script toma como entrada de usuario un índice que selecciona una de estas trayectorias.

El script inicializa objetos robot para representar al robot en condiciones ideales (ideal) y condiciones reales con ruido (real). Se establecen parámetros de ruido y pose inicial para ambos robots.

La simulación del movimiento del robot se lleva a cabo en un bucle, calculando velocidades lineales y angulares para alcanzar cada punto en la trayectoria. La simulación continúa hasta que la distancia al

objetivo es menor que un umbral (EPSILON) o hasta que se alcanza un límite de iteraciones.

Finalmente, se imprimen resultados como el recorrido total, la distancia real al objetivo, y se utiliza la función mostrar para representar gráficamente las trayectorias ideales y reales del robot. Es importante tener en cuenta que la funcionalidad del script depende del módulo robot.

4.3. Implementación de la localización

Tras la explicación del funcionamiento del script de *localización* proporcionado, se realiza la implementación de esta de la siguiente manera:

Listing 4.3: Script de localización implementado

```
import sys
from math import *
from robot import robot
import random
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

def distancia(a,b):
  return np.linalg.norm(np.subtract(a[:2],b[:2]))

def angulo_rel(pose,p):
  w = atan2(p[1]-pose[1],p[0]-pose[0])-pose[2]
  while w >  pi: w -= 2*pi
  while w < -pi: w += 2*pi
  return w

def mostrar(objetivos,ideal,trayectoria):
  objT   = np.array(objetivos).T.tolist()
  trayT  = np.array(trayectoria).T.tolist()
  ideT   = np.array(ideal).T.tolist()
  bordes = [min(trayT[0]+objT[0]+ideT[0]),max(trayT[0]+objT[0]+ideT[0]),
            min(trayT[1]+objT[1]+ideT[1]),max(trayT[1]+objT[1]+ideT[1])]
  centro = [(bordes[0]+bordes[1])/2.,(bordes[2]+bordes[3])/2.]
  radio  = max(bordes[1]-bordes[0],bordes[3]-bordes[2])*.75
  plt.xlim(centro[0]-radio,centro[0]+radio)
  plt.ylim(centro[1]-radio,centro[1]+radio)
  idealT = np.array(ideal).T.tolist()
  plt.plot(idealT[0],idealT[1],'-g')
  plt.plot(trayectoria[0][0],trayectoria[0][1],'or')
  r = radio * .1
  for p in trayectoria:
    plt.plot([p[0],p[0]+r*cos(p[2])],[p[1],p[1]+r*sin(p[2])],'-r')
  objT   = np.array(objetivos).T.tolist()
  plt.plot(objT[0],objT[1],'-.o')
  plt.show()
  input()
  plt.clf()

def localizacion(balizas, real, ideal, centro, radio, mostrar=0):
  imagen = []
  error_menor = sys.maxsize
  mejor_punto = []
```

```
  incremento = 0.05
  for j in np.arange(-radio, radio, incremento):
    imagen.append([])
    for i in np.arange(-radio, radio, incremento):
      x_componente = centro[0] + i
      y_componente = centro[1] + j
      ideal.set(x_componente, y_componente, ideal.orientation)
      error = real.measurement_prob(ideal.sense(balizas), balizas)
      imagen[-1].append(error)
      if (error < error_menor):
        error_menor = error
        mejor_punto = [x_componente, y_componente]
  ideal.set(mejor_punto[0], mejor_punto[1], real.orientation)
  print("Modificacion:", mejor_punto, error_menor)


  if mostrar:
    plt.xlim(centro[0]-radio,centro[0]+radio)
    plt.ylim(centro[1]-radio,centro[1]+radio)
    imagen.reverse()
    plt.imshow(imagen,extent=[centro[0]-radio,centro[0]+radio,\
                              centro[1]-radio,centro[1]+radio])
    balT = np.array(balizas).T.tolist();
    plt.plot(balT[0],balT[1],'or',ms=10)
    plt.plot(ideal.x,ideal.y,'D',c='#ff00ff',ms=10,mew=2)
    plt.plot(real.x, real.y, 'D',c='#00ff00',ms=10,mew=2)
    plt.show()
    input()
    plt.clf()

P_INICIAL = [0.,4.,0.] # Pose inicial ( posicin y orientacion)
V_LINEAL  = .7         # Velocidad lineal    (m/s)
V_ANGULAR = 140.       # Velocidad angular    (/s)
FPS       = 10.        #  Resolucin temporal (fps)

HOLONOMICO = 1
GIROPARADO = 0
LONGITUD   = .2

trayectorias = [
    [[1,3]],
    [[0,2],[4,2]],
    [[2,4],[4,0],[0,0]],
    [[2,4],[2,0],[0,2],[4,2]],
    [[2+2*sin(.8*pi*i),2+2*cos(.8*pi*i)] for i in range(5)]
    ]

if len(sys.argv)<2 or int(sys.argv[1])<0 or int(sys.argv[1])>=len(trayectorias):
  sys.exit(sys.argv[0]+" <indice entre 0 y "+str(len(trayectorias)-1)+">")
objetivos = trayectorias[int(sys.argv[1])]

EPSILON = .1                # Umbral de distancia
V = V_LINEAL/FPS            # Metros por fotograma
W = V_ANGULAR*pi/(180*FPS)  # Radianes por fotograma

ideal = robot()
```

```
ideal.set_noise(0,0,.1)   # Ruido lineal / radial / de sensado
ideal.set(*P_INICIAL)     # operador 'splat'

real = robot()
real.set_noise(.01,.01,.1)  # Ruido lineal / radial / de sensado
real.set(*P_INICIAL)

random.seed(0)
tray_ideal = [ideal.pose()]  # Trayectoria percibida
tray_real = [real.pose()]     # Trayectoria seguida

tiempo  = 0.
espacio = 0.
random.seed(datetime.now())

localizacion(objetivos,real,ideal, [0, 4],5,1)

for punto in objetivos:
  while distancia(tray_ideal[-1],punto) > EPSILON and len(tray_ideal) <= 1000:
    pose = ideal.pose()

    w = angulo_rel(pose,punto)
    if w > W:  w =  W
    if w < -W: w = -W
    v = distancia(pose,punto)
    if (v > V): v = V
    if (v < 0): v = 0

    if HOLONOMICO:
      if GIROPARADO and abs(w) > .01:
        v = 0
      ideal.move(w,v)
      real.move(w,v)
    else:
      ideal.move_triciclo(w,v,LONGITUD)
      real.move_triciclo(w,v,LONGITUD)
    tray_ideal.append(ideal.pose())
    tray_real.append(real.pose())

    if (real.measurement_prob(real.sense(objetivos), objetivos) > EPSILON):
      localizacion(objetivos, real, ideal, ideal.pose(), 0.2, mostrar=0)

    espacio += v
    tiempo  += 1

if len(tray_ideal) > 1000:
  print ("<!> Trayectoria muy larga - puede que no se haya alcanzado la posicion
      final.")
print ("Recorrido: "+str(round(espacio,3))+"m / "+str(tiempo/FPS)+"s")
print ("Distancia real al objetivo: "+\
    str(round(distancia(tray_real[-1],objetivos[-1]),3))+"m")
mostrar(objetivos,tray_ideal,tray_real)  # Representacin grfica
```

El script ejecuta una serie de trayectorias, cada una definida por una lista de puntos de destino. Durante la simulación, el robot ajusta su posición y orientación para seguir la trayectoria ideal, con la posibilidad de introducir ruido en las mediciones del sensor. Además, se realiza la implementación de localización basada

en la comparación de mediciones de sensores con la posición estimada del robot, utilizando un método de búsqueda en una cuadrícula para encontrar la posición más probable. Es decir, cuando se determina que el robot ideal se ha desviado una cierta variación con respecto al robot real, se aplica la localización para determinar la posición del robot real. Esta localización, se encarga de realizar una búsqueda del robot real dentro de una cuadrícula de píxeles, de tal manera que se determina la posición del robot real dentro de la cuadrícula de píxeles.

Al final de la simulación, se genera una representación gráfica de la trayectoria ideal y la trayectoria seguida por el robot. Este script proporciona una herramienta de simulación para evaluar el rendimiento de la localización del robot y su capacidad para seguir trayectorias específicas en un entorno simulado.

4.4. Ejecución del código

Tras la explicación de la implementación de la localización, se procede a la ejecución del script para la comprobación del funcionamiento del mismo. Para ello, se ejecuta el siguiente comando en la terminal:

Listing 4.4: Ejecución del script de Localización

```
$ python3 Localization-script-implementation.py 4
```

Tras esto, se obtiene el siguiente resultado:

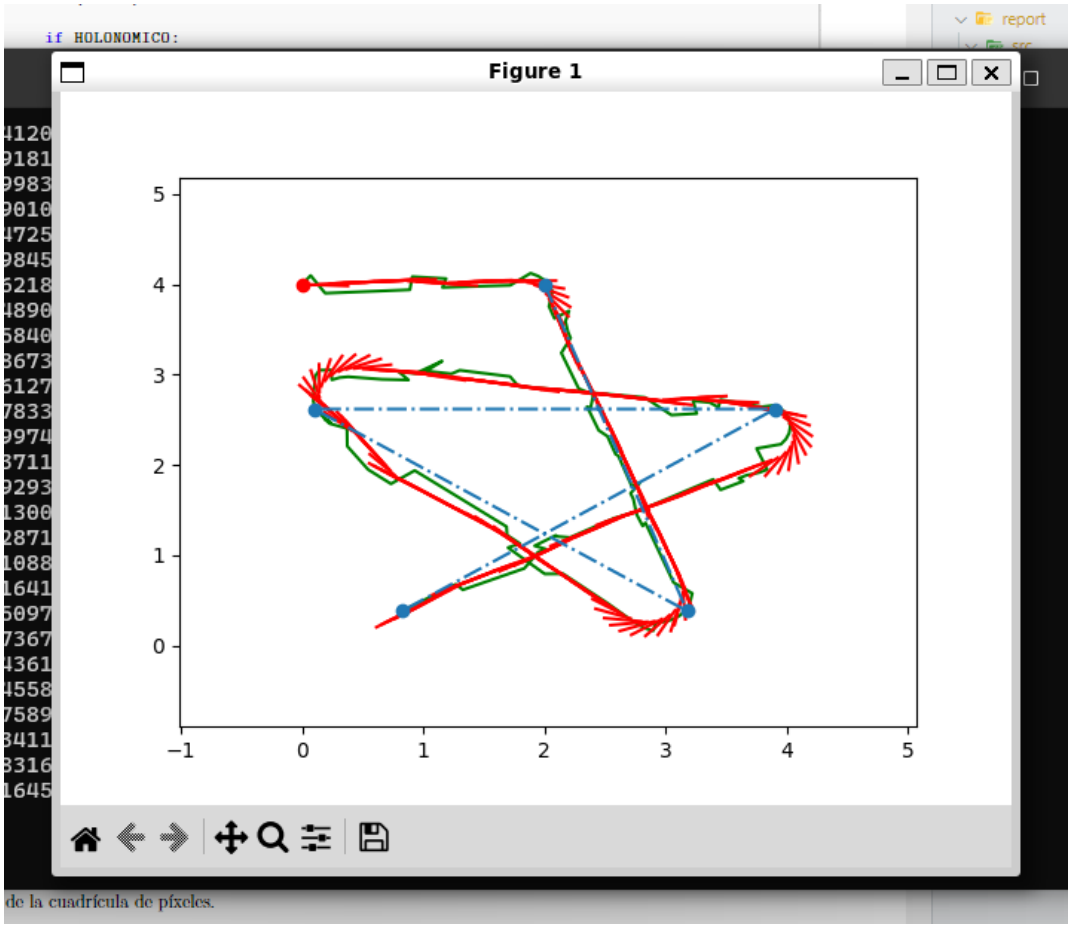

Figura 4.1: Resultado de la ejecución del script de Localización

4.5. Parámetros modificados

En cuanto a los parámetros modificados durante la implementación de la localización, se pueden tener en cuenta los siguientes parámetros por ejemplo:

Listing 4.5: Ejecución de la primera operación de Localización

```
1     localizacion(objetivos,real,ideal, [0, 4],5,1)
```

Como se puede observar anteriormente, se pueden modificar los parámetros de *centro*, que para este caso tome como ejemplo *[0, 4]*, los parámetros de *radio*, que para este caso tome como ejemplo *5*. Por tanto, dependiendo de los valores que sean establecidos en estos campos, el resultado de la localización será uno u otro.

Otro de los parámetros con los que se ha jugado un poco ha sido el radio de la localización en el momento en el que se determina que la distancia entre el robot real y el ideal es mayor que un umbral, este valor se puede observar en el trozo de código adjunto a continuación:

Listing 4.6: Localización del robot

```
1     if (real.measurement_prob(real.sense(objetivos), objetivos) > EPSILON):
2         localizacion(objetivos, real, ideal, ideal.pose(), 0.2, mostrar=0)
```

Como se puede observar en el trozo de código anterior, se puede modificar el valor del radio de la localización, el cual se encuentra en el campo *0.2*. Cuanto menor sea dicho valor, más preciso será el resultado de la localización, pero, por el contrario, cuanto mayor sea dicho valor, menos preciso será el resultado de esta.

4.6. Mejoras en la implementación

Tras la explicación del script de localización suministrado, se pueden tomar como mejoras de esta práctica pues el hecho de la visualización final del robot, es decir, se podría realizar la implementación de la visualización del robot en otros tipos de entornos gráficos, como por ejemplo, *Blender*. Es decir, se podría realizar la implementación de la visualización del robot en un entorno gráfico más realista, en el cual se pueda observar al robot desplazarse en tiempo real, y no mediante una serie de imágenes estáticas, de tal manera que el alumnado pueda comprender de manera mucho más sencilla y visual el funcionamiento de la localización, es decir, tener tanto al robot real como al robot ideal en tiempo real, de esta manera, cuando se aplique la localización, se pueda observar como el robot real se desplaza hasta la posición del robot ideal.

Para poder conectar los scripts en python con *Blender*, se puede hacer uso de la librería *bpy* [6], la cual permite la conexión de scripts en python con *Blender*, de tal manera que se podría realizar la implementación de la práctica en python de la misma manera que hasta el momento, pero, cuando se ejecute la operación de localización dentro del script, se pare la visualización en tiempo real de Blender para poder ejecutar una nueva visualización de la localización del robot.

4.7. Conclusiones

Para concluir con la práctica de Localización, se puede decir que se ha conseguido realizar la implementación de la localización de un robot móvil mediante el empleo de una serie de balizas en el entorno, las cuales permiten al robot móvil conocer su posición en el entorno, además, se hace uso de otro robot móvil

denominado como *robot real"*, el cual se encarga de marcar la diferenciación con respecto al robot móvil al que se debe de aplicar la localización cada vez que este se desvíe o se pierda.

Esta práctica me ha resultado personalmente la más interesante de las tres prácticas realizadas, ya que el hecho de poder ver como dependiendo de los valores que se le introduzcan a los distintos parámetros de la localización, se tenga que el robot sea más preciso o menos, me resulta muy interesante, porque para mi, la localización es una de las partes más importantes de la robótica, ya que si no se tiene una buena localización, el robot no podrá realizar las tareas que se le encomienden de manera correcta.

Capítulo 5 Filtro de partículas

5.1. ¿Qué es un filtro de partículas?

El filtro de partículas es un método de estimación de estado que utiliza una aproximación basada en muestras para representar la distribución de probabilidad de un estado. El filtro de partículas es un método no paramétrico, lo que significa que no hace suposiciones sobre la forma de la distribución de probabilidad. En cambio, el filtro de partículas representa la distribución de probabilidad mediante una colección de muestras, llamadas partículas, que se distribuyen de acuerdo con la distribución de probabilidad.

La idea fundamental se basa en representar la función de distribución posterior buscada con un conjunto de muestras aleatorias con pesos asociados y calcular estimados basados en estas muestras y pesos.

Bibliografía

[1] UdeSantigoVirtual. [2023]. Herramientas matemáticas. Universidad de Santiago de Chile. Recuperado de http://www.udesantiagovirtual.cl/moodle2/mod/book/view.php?id=24916

[2] Samuel Martín Morales. [2023]. Direct-Kinematics-using-Denavit-Hartenberg. GitHub. Recuperado de https://github.com/Samuelmm15/Direct-Kinematics-using-Denavit-Hartenberg.git

[3] Samuel Martín Morales. [2023]. Inverse-Kinematics. GitHub. Recuperado de https://github.com/Samuelmm15/Inverse-Kinematics.git.

[4] Samuel Martín Morales. [2023]. Robot-Localization. GitHub. Recuperado de https://github.com/Samuelmm15/Robot-Localization.git.

[5] Universidad de La Laguna. [2023]. Cinemática Directa mediante Denavit-Hartenberg. Universidad de La Laguna. https://drive.google.com/file/d/1639F9ebw14zquNaZcDf6o_u-_Tpaksh4/view?usp=sharing

[6] Comunidad Python. [2023]. bpy - Blender Python. Comunidad Python. https://pypi.org/project/bpy/

[7] Escamilla Losoyo José de Jesús. [2020]. Cinemática inversa. Studocu. https://www.studocu.com/es-mx/document/instituto-tecnologico-de-leon/robotica/cinematica-inversa/12281135

[8] Tekniker. [2023]. Localización y planificación de trayectorias. Tekniker. https://www.tekniker.es/es/localizacion-y-planificacion-de-trayectorias

[9] Wikipedia. [2023]. Filtro de partículas. Wikipedia. https://es.wikipedia.org/wiki/Filtro_de_part%C3%ADculas

[10] Azucena Fuentes Ríos. [2011]. LOCALIZACIÓN Y MODELADO SIMULTÁNEOS EN ROS PARA LA PLATAFORMA ROBÓTICA MANFRED. Universidad Carlos III de Madrid. https://e-archivo.uc3m.es/bitstream/handle/10016/13392/pfc_Azucena_Fuentes_Rios.pdf?sequence=1&isAllowed=y