Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Informe de Prácticas

Robótica Computacional

Grado en Ingeniería Informática

José Ramón Morera Campos

La Laguna, 1 de enero de 2025

Índice general

Índice de Figuras

Capítulo 1

Introducción

En este informe se analizan las cuatro prácticas realizadas a lo largo de la asignatura de Robótica Computacional.

Se estudia el comportamiento de los programas realizados, mostrando ejemplos con diversas configuraciones de parámetros. Adicionalmente, se explican las modificaciones realizadas, así como otras propuestas de mejora.

Finalmente, para cada prácitca se exponen unas breves conclusiones sobre los resultados obtenidos. Dichas conclusiones se exponen en ingles, siguiendo los requisitos del informe.

Capítulo 2

Cinemática Directa

En esta práctica se calcula la cinemática directa para manipuladores tridimensionales. Esto es, a partir de la morfología del robot y de las variables articulares, se calculan las coordenadas de cada articulación.

En el código, se puede modificar la morfología del robot indicando la longitud de los elementos rígidos, así como la existencia de articulaciones de revolución o prismáticas. Al ejecutar el script se indican los valores de las variables articulares.

2.1. Código Implementado

2.1.1. Análisis

El script proporcionado contiene todo el código necesario para calcular la cinemática directa y visualizar los manipuladores. Solamente se deben modificar ciertos parámetros para adaptar el script a la morfología del robot:

- `nvar` Número de variables que se introduciran al ejecutar el programa, correspondientes a las variables articulares.
- `Parámetros de Denavit-Hartenberg` son 4 vectores, uno para cada parámetro. El tamaño del vector se corresponde al número de articulaciones.
- `Orígenes para cada articulación` un vector con las coordenadas homogéneas de cada articulación.
- `Matrices T` son matrices que permiten realizar la transformación de un sistema de coordenadas a otro. Las matrices entre sistemas consecutivos son triviales, y se construyen con sus parámetros de Denavit–Hartenberg. Las matrices que describen la transformación entre sistemas no consecutivos se obtienen multiplicando las matrices de transformación de los sistemas intermedios. Por ejemplo, la matriz T02, que transforma el sistema de coordenadas 0 al 2, se obtiene realizando el producto de las matrices T01 y T12.
- `Coordenadas de cada articulación` se calcula el origen de coordenadas del sistema de cada articulación mediante el producto de la matriz de transformación T0X y el origen de coordenadas oXX.

- `Visualización del robot` se deben modificar las instrucciones de visualización para adaptarlas a la morfología del robot, incluyendo los origenes de coordenadas calculados que se quieren visualizar.

En la figura 2.1 se muestra el fragmento del script que se debe modificar.

```
95  # Introducción de los valores de las articulaciones
96  nvar=2 # Número de variables
97  if len(sys.argv) != nvar+1:
98    sys.exit('El número de articulaciones no es el correcto ('+str(nvar)+')')
99  p=[float(i) for i in sys.argv[1:nvar+1]]
100
101 # Parámetros D-H:
102 #        1    2
103 d  = [   0,   0]
104 th = [p[0],p[1]]
105 a  = [  10,   5]
106 al = [   0,   0]
107
108 # Orígenes para cada articulación
109 o00=[0,0,0,1]
110 o11=[0,0,0,1]
111 o22=[0,0,0,1]
112
113 # Cálculo matrices transformación
114 T01=matriz_T(d[0],th[0],a[0],al[0])
115 T12=matriz_T(d[1],th[1],a[1],al[1])
116 T02=np.dot(T01,T12)
117
118 # Transformación de cada articulación
119 o10 =np.dot(T01, o11).tolist()
120 o20 =np.dot(T02, o22).tolist()
121
122 # Mostrar resultado de la cinemática directa
123 muestra_origenes([o00,o10,o20])
124 muestra_robot   ([o00,o10,o20])
125 input()
126
```

Figura 2.1: Porción del código que se debe modificar para cada manipulador

Al ejecutar el código se deben proporcionar como parámetros los valores de las variables articulares: unidades de longitud en el caso de las articulaciones prismáticas, y grados de rotación en el caso de las articulaciones de revolución.

2.1.2. Complejidad

La complejidad del código depende del número de articulaciones del manipulador. Por cada nueva articulación, solamente es necesario realizar dos productos de matrices de transformación homogénea. Uno para obtener la matriz de transformación y otro para obtener las coordenadas de la articulación. Por tanto, la complejidad es lineal respecto al número de articulaciones.

2.2. Mejoras

En esta práctica no se han implementado mejoras, pero se proponen algunas que podrían ser interesantes.

- `Lectura de JSON` se podría implementar la lectura de la morfología del robot desde un fichero JSON, evitando tener que modificar el script para cada manipulador.
- `Modificación de los parámetros en tiempo real` Sería de gran utilidad poder modificar las variables articulares mientras se visualiza el manipulador, para poder estudiar fácilmente cómo lo afectan.

2.3. Ejemplo

Se emplea el Manipulador 3 de entre los propuestos para visualizar la cinemática directa. Se ha elegido puesto que tiene suficiente complejidad para ilustrar correctamente la práctica, pero sin ser excesiva.

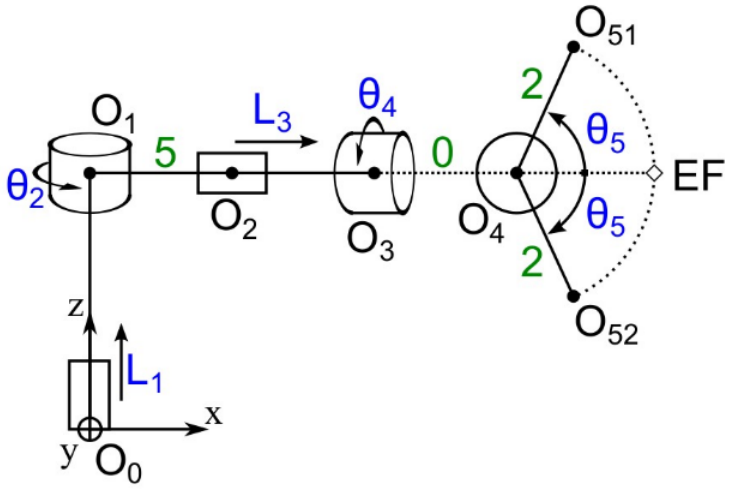

Figura 2.2: Manipulador 3

2.3.1. Código

La figura 2.3 muestra el código modificado para el manipulador 3. Cabe destacar que se necesita emplear un sistema de coordenadas auxiliar entre O1 y O2 para que el sitema sea dextrógiro y compatible Denavit-Hartenberg. También resulta reseñable que para la visualización del manipulador, en las 2 funciones pertinentes, los puntos O51 y O52 se agrupan en un mismo vector, para señalar que son 2 bifurcaciones desde el punto anterior.

2.3.2. Ejecución

En primer lugar, en la figura 2.4 se muestra un ejemplo de ejecución con todas las variables articulares a 0. Observamos que algunos orígenes se solapan, por ejemplo O0 y O1, o O51 y O52.

A continuación, probamos dando valor de 5 a las articulaciones prismáticas y 30º para la articulación O4. Dejamos el resto a 0. De esta forma, como observamos en la figura 2.5 el manipulador se asemeja al diagrama del ejercicio.

En último lugar, aplicamos una rotación de 30º en las articulaciones O1 y O3. Observamos el resultado en la figura 2.7.

De esta forma, hemos comprobado que el código funciona correctamente y que la descripción del manipulador realizada es correcta.

2.4. Conclusions

We have studied the forward kinematics of a 3D manipulator. We have seen that we can characterize any manipulator by its Denavit-Hartenberg parameters, and that it is easy and fast to calculate the coordinates of each joint. We have also seen that the visualization of the manipulator is very useful to understand its behavior and to check that the calculations made to obtain the parameters are correct.

Although the script provided is very useful, it has room for improvements. Also, we still have to manually calculate the Denavit-Hartenberg parameters, which can be a tedious task for complex manipulators.

```
nvar=5 # 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    aux    3      4     51      52      EF
d  = [p[0],    0,    5, p[2],    0,       0,       0,   0]
th = [ -90, p[1],    0,    0, p[3]-90, p[4]-90, -p[4]-90, -90]
a  = [   0,    0,    0,    0,    0,       2,       2,   2]
al = [   0,  -90,    0,    0,  -90,       0,       0,   0]

# Orígenes para cada articulación
o00=[0,0,0,1]
o11=[0,0,0,1]
oauxaux=[0,0,0,1]
o22=[0,0,0,1]
o33=[0,0,0,1]
o44=[0,0,0,1]
o5151=[0,0,0,1]
o5252=[0,0,0,1]
oefef=[0,0,0,1]

# Cálculo matrices transformación
T01=matriz_T(d[0],th[0],a[0],al[0])
T1aux=matriz_T(d[1],th[1],a[1],al[1])
Taux2=matriz_T(d[2],th[2],a[2],al[2])
T23=matriz_T(d[3],th[3],a[3],al[3])
T34=matriz_T(d[4],th[4],a[4],al[4])
T451=matriz_T(d[5],th[5],a[5],al[5])
T452=matriz_T(d[6],th[6],a[6],al[6])
T4ef=matriz_T(d[7],th[7],a[7],al[7])

T0aux=np.dot(T01, T1aux)
T02=np.dot(T0aux,Taux2)
T03=np.dot(T02,T23)
T04=np.dot(T03,T34)
T051=np.dot(T04,T451)
T052=np.dot(T04,T452)
T0ef=np.dot(T04,T4ef)

# Transformación de cada articulación
o10 =np.dot(T01, o11).tolist()
o20 =np.dot(T02, o22).tolist()
o30 =np.dot(T03, o33).tolist()
o40 =np.dot(T04, o44).tolist()
o510 =np.dot(T051, o5151).tolist()
o520 =np.dot(T052, o5252).tolist()
oef0 =np.dot(T0ef, oefef).tolist()

# Mostrar resultado de la cinemática directa
muestra_origenes([o00,o10,o20,o30, o40, [o510, o520], oef0])
muestra_robot   ([o00,o10,o20,o30, o40, [[o510], [o520]]], oef0)
input()
```

Figura 2.3: Código para el manipulador 3

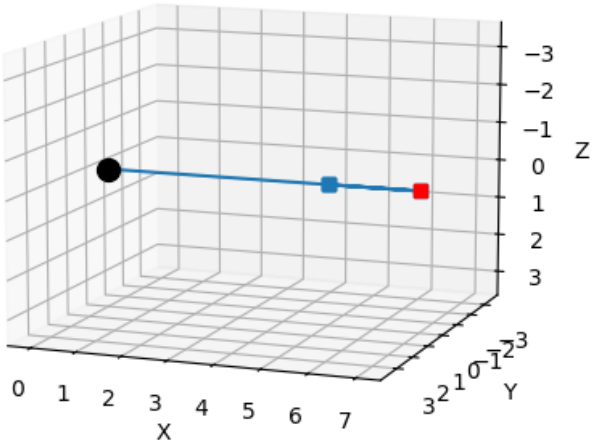

Figura 2.4: Ejecución 1

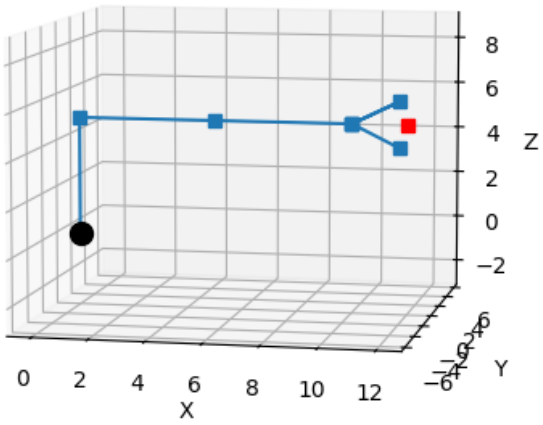

Figura 2.5: Ejecución 2

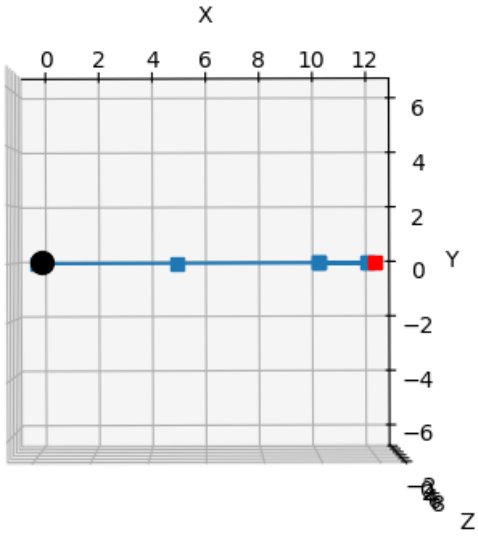

Figura 2.6: Ejecución 2, perspectiva vertial

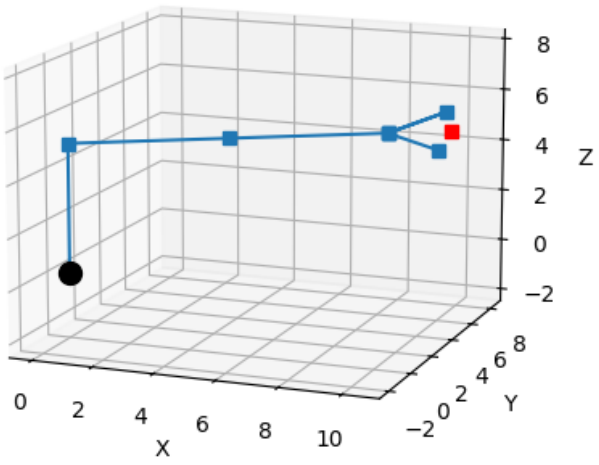

Figura 2.7: Ejecución 3

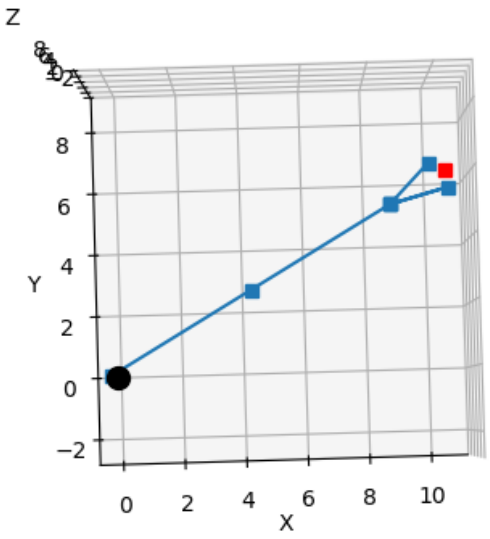

Figura 2.8: Ejecución 3, perspectiva vertial

Capítulo 3

Cinemática Inversa

En esta práctica se calcula la cinemática inversa para manipuladores bidimensionales. Es decir, se obtienen las variables articulares necesarias para alcanzar un punto objetivo en el plano. Para ello, se emplea el algoritmo iterativo *Cyclic Coordinate Descent* (CCD).

Se caracteriza el robot mediante sus articulaciones, prismáticas o de revolución, y sus parámetros de Denavit-Hartenberg theta y a (al encontrarnos en 2D solo hay 2 parámetros). Al ejecutar el script se indica las coordenadas del punto objetivo para el extremo del robot, y se obtienen las variables articulares necesarias para alcanzar dicho punto.

3.1. Código Implementado

3.1.1. Análisis

El script proporcionado contiene muchas de las funciones necesarias, sin embargo, se deben implementar varias funciones para poder calcular la cinemática inversa. Principalmente, se debe implementar la lógica del CCD. Se realiza un cálculo para cada articulación desde el extremo final (EF) hasta el origen del robot, actualizando su valor y recalculando la cinemática directa. Al finalizar, se comprueba la distancia al punto objetivo, si es menor a la tolerancia indicada (umbral de convergencia epsilon), se finaliza. En caso contrario, se itera hasta conseguir una aproximación tolerable, o que se detecte la imposibilidad de convergencia.

- `Detección del tipo de articulación` se distingue entre articulaciones prismáticas y de revolución.
- `Cálculo de las articulaciones prismáticas` Para calcular la distancia que se incrementa o decrementa la articulación, se calcula la distancia entre el extremo del robot y el punto objetivo, proyectada sobre la recta de movimiento de la articulación prismática. Esto se consigue realizando el producto del vector unitario $\vec{u}$, que representa la dirección de la variable prismática, y el vector $\vec{v}$, que une el extremo del robot y el punto objetivo. El vector $\vec{u}$ se obtiene sumando los ángulos de todas las articulaciones previas a la actual, y calculando el seno y coseno del resultado.
- `Cálculo de las articulaciones de revolución` se calcula el ángulo en el que se debe modificar la articulación calculando la diferencia entre el vector $\vec{v}$, que une la articulación de rotación y el punto extremo EF, y el vector $\vec{w}$, que une la

articulación de rotación y el punto objetivo. Para ello, se normalizan ambos vectores y se calcula la diferencia entre sus arcotargentes. En la práctica, se emplea la función *math.atan2* ya que se encarga de realizar las comprobaciones necesarias de cuadrantes.

- `Corrección de ángulo de rotación` se corrige el ángulo calculado para que se encuentre en el rango $[-\pi, \pi]$.
- `Límites` se implementan límites inferiores y superiores para cada articulación. En caso de que el nuevo valor de las variables articulares este por debajo, se emplea el límite inferior, y si está por encima, el superior.

En las figuras 3.1 y 3.2 se muestra el código implementado para el CCD.

```
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)):
    indice_actuador = len(th)-(i+1)
    EF = O[i][len(th)] # Punto final del brazo; Actualizacion con los calculos de cinematica directa de cada iteración

    # cálculo de la cinemática inversa:
    # O[0] son las posiciones de cada origen antes de la primera iteracion
    # alinear O[0][len(th)-(i+2)], O[0][len(th)-(i+1)]
    print("objetivo", [x for x in objetivo])
    o_rotacion = O[0][len(th)-(i+1)]
    print("O-1", [x for x in o_rotacion])

    if (art_types[indice_actuador]): # Articulación prismática
      w = 0
      for x in range(i):
        w += th[x]

      u = [math.cos(w), math.sin(w)]
      v = [objetivo[0]-EF[0], objetivo[1]-EF[1]]
      d = np.dot(u,v)

      print ("d", d);
      new_a = a[len(th)-(i+1)] + d

      # límites
      if (new_a < limit_inf[indice_actuador]):
          new_a = limit_inf[indice_actuador]
      elif (new_a > limit_sup[indice_actuador]):
          new_a = limit_sup[indice_actuador]

      a[len(th)-(i+1)] = new_a
```

Figura 3.1: Código del CCD para articulaciones prismáticas

3.1.2. Complejidad

Para la cinemática inversa, se realiza una aproximación iterativa, en la que en cada iteración se debe calcular cada articulación. Tras actualizar cada articulación, se debe recalcular la cinemática directa. Es decir, para N articulaciones en una iteración se debe calcular N veces la cinemática directa. Como se dijo en el anterior capítulo, la cinemática directa es lineal respecto al número de articulaciones, por lo que la complejidad de la cinemática inversa es cuadrática $\sim O(N^2)$.

```
else:
  # vector del punto de rotacion al punto objetivo
  w =  [a - b for a, b in zip(objetivo, o_rotacion)]
  w_magnitude  = sqrt( w[0]**2+ w[1]**2)
  w = [x/w_magnitude for x in w]

  # vector del punto de rotacion al punto extremo EF
  v = [a - b for a, b in zip(EF, o_rotacion)]
  v_magnitude  = sqrt( v[0]**2+ v[1]**2)
  v = [x/v_magnitude for x in v]

  # arcotangente
  alpha1 = math.atan2(w[1], w[0])
  alpha2 = math.atan2(v[1], v[0])

  delta = alpha1-alpha2
  new_th = delta + th[len(th)-(i+1)]

  # corrección del ángulo (-pi <= delta <= pi)
  while new_th > math.pi:
    new_th -= 2*math.pi
  while new_th < -math.pi:
    new_th += 2*math.pi

  # límites
  if (new_th < limit_inf[indice_actuador]):
    new_th = limit_inf[indice_actuador]
  elif (new_th > limit_sup[indice_actuador]):
    new_th = limit_sup[indice_actuador]

  th[len(th)-(i+1)] = new_th # actualizamos th

# Calculamos cinemática directa
O.append(cin_dir(th,a))
```

Figura 3.2: Código del CCD para articulaciones de revolución

3.1.3. Fé de erratas

El código entregado en prácticas contenía un pequeño pero importante error. En la detección del tipo de articulación, se estaba empleando un índice incorrecto: se recorría en el orden inverso, de origen a extremo. Esto provocaba que se detectaran las articulaciones de forma incorrecta. No se había detectado en la entrega ya que los ejemplos probados eran simétricos en ese sentido. Se ha corregido este problema, y como se puede observar más adelante, el programa funciona correctamente en todos los casos.

3.2. Mejoras

Se ha implementado una sola mejora, la lectura de JSON.

3.2.1. Lectura de JSON

Se leen las características de cada articulación desde un fichero JSON. Cada articulación se caracteriza por su tipo (rotación o prismática), sus parámetros de Denavit-Hartenberg (th y a), y sus límites inferiores y superiores, expresados en unidades de longitud o radianes.

En el fichero, las articulaciones se indican en orden desde el origen hasta el extremo, siendo la primera la más cercana al origen.

En la figura 3.3 se muestra el código implementado para la lectura de JSON. En la figura 3.4 se observa un ejemplo de fichero JSON para un robot con una articulación de rotación, otra prismática, y otra de revolución.

```
# Variables del manipulador

art_types = [] # Tipos de articulación; 1 = prismatica, 0 = rotacion
# valores articulares arbitrarios para la cinemática directa inicial
th=[]
a =[]
# límites superiores e inferiores
limit_sup=[]
limit_inf=[]

# Lectura de fichero
with open("robot.json", "r") as file:
    data = json.load(file)

for articulacion in data:
    if (articulacion['type'] == "rotacion"):
      art_types.append(0)
    elif (articulacion['type'] == "prismatica"):
      art_types.append(1)
    th.append(articulacion['th'])
    a.append(articulacion['a'])
    limit_sup.append(articulacion['limit_sup'])
    limit_inf.append(articulacion['limit_inf'])
```

Figura 3.3: Código para la lectura de JSON

3.2.2. Propuestas

Actualmente solo se consideran las dimensiones iniciales del robot para la visualización. Sería interesante considerar también las coordenadas del punto objetivo para garantizar que se visualiza correctamente.

3.3. Ejemplos de Ejecución

En la figura 3.5 se muestra un ejemplo de fichero JSON con las características de un robot de 5 articulaciones.

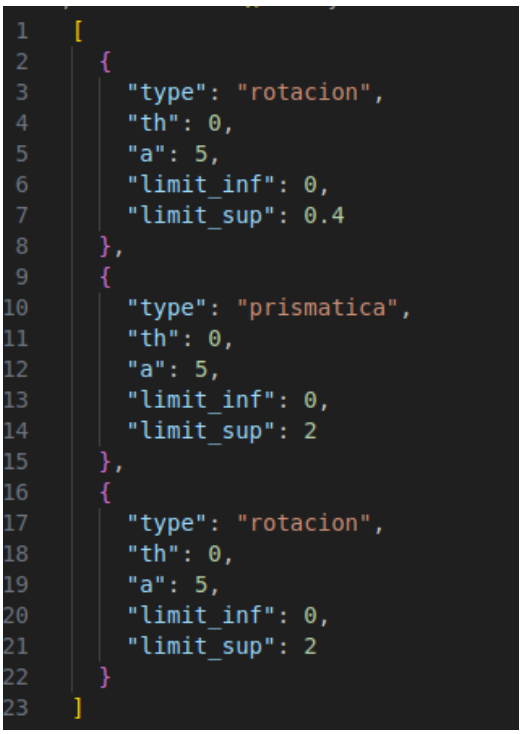

Figura 3.4: Ejemplo de fichero JSON para un robot de 3 articulaciones

3.3.1. Ejecución 1

En la figura 3.6 se muestra la primera iteración del algoritmo sobre el robot descrito considerando como objetivo el punto 10,10. En la figura 3.7 se muestra la última iteración del algoritmo, en la que se ha alcanzado el umbral de convergencia. En este caso, se consigue converger en solo 2 iteraciones.

3.3.2. Ejecución 2

En la figura 3.8 se muestra la primera iteración del algoritmo sobre el robot descrito considerando como objetivo el punto -20,-20. En la figura 3.9 se muestra la última iteración del algoritmo, en la que se ha alcanzado el umbral de convergencia. En este caso, se consigue converger en 5 iteraciones. Se aprovecha este ejemplo para estudiar la influencia del valor del umbral de convergencia en el número de iteraciones necesarias. El epsilon empleado por defecto es de 0.01, cambiándolo a 0.1 se converge en 4 iteraciones, una menos, con una distancia al objetivo de 0.01663 unidades. Se comprueba por tanto empíricamente que un umbral mayor implica menos iteraciones, pero también una menor precisión en el resultado obtenido.

3.3.3. Ejecución 3

Empleando el robot de la figura 3.4 se comprueba que se detectan los casos en los que no se converge, esto es, no existe solución para alcanzar el punto objetivo con el robot dado.

La figura 3.10 muestra la última iteración para el punto objetivo 10,10. Por los límites establecidos, el robot solo puede alcanzar puntos a un radio de 12 unidades, por lo que no puede llegar al punto 10,10. En la figura 3.11 se muestra la salida por consola indicando

que no se ha podido converger, ya que se detecta que la distancia al objetivo ha disminuido menos de epsilon/100.

3.4. Conclusions

We have implemented the inverse kinematics for a 2D manipulator, using the CCD algorithm. The implementation considers both prismatic and revolute joints, for which we admit lower and upper limits. We have studied the effects of different convergence thresholds, seen that a higher threshold implies fewer iterations but also less precision. Also, we have studied the complexity of the inverse kinematics with the CCD algorithm. We have seen that it is quadratic, as it is an iterative process in which we must calculate the direct kinematics for each articulation in each iteration.

We can conclude that the inverse kinematics are generally more complex and expensive than the direct kinematics, as an exact approach is not always valid because for some configurations there may exist multiple or none solutions.

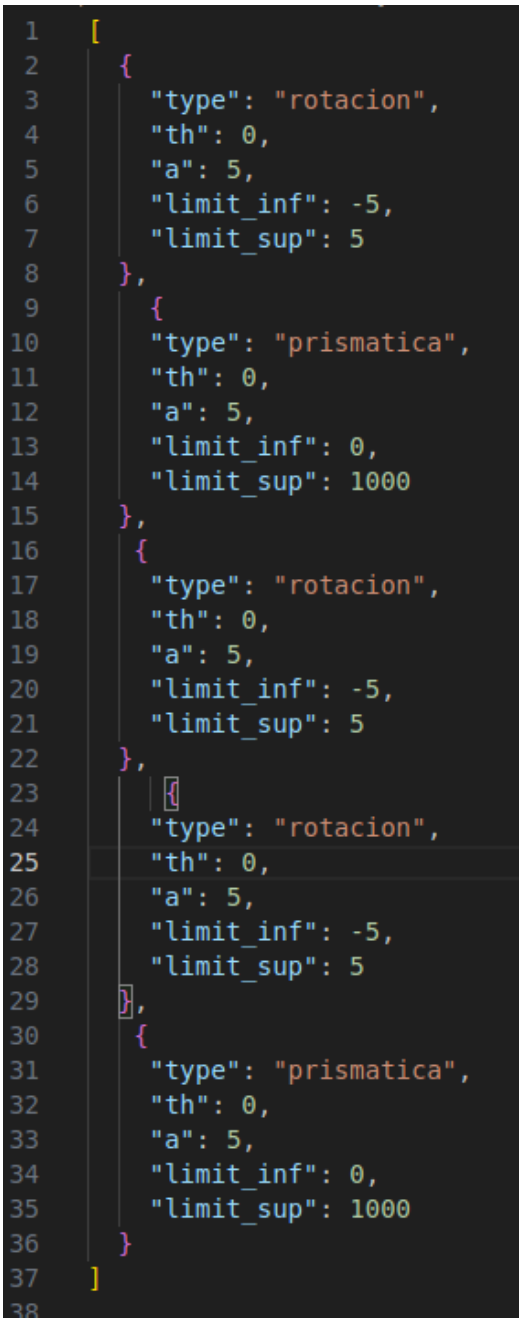

Figura 3.5: Ejemplo de fichero JSON para un robot de 5 articulaciones

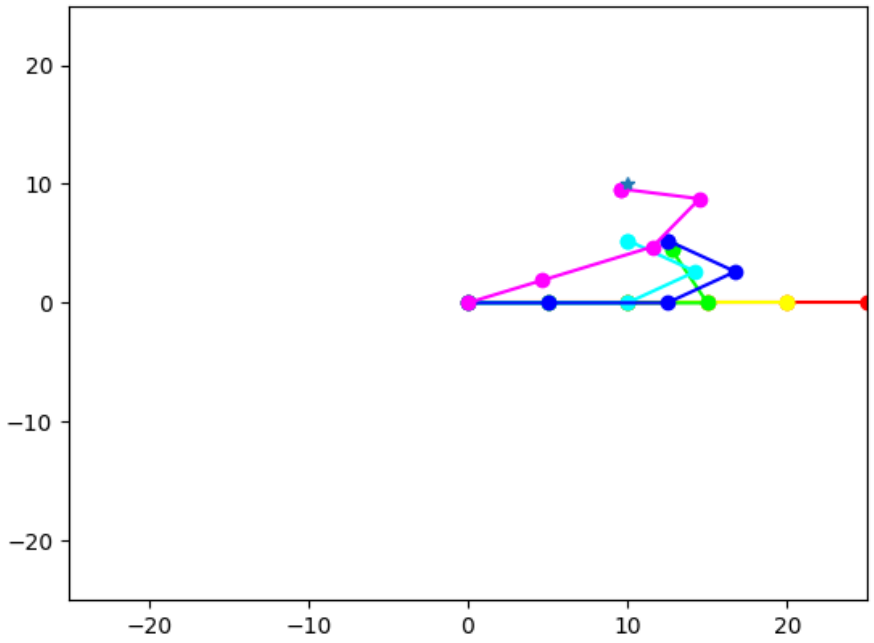

Figura 3.6: Primera iteración para objetivo 10,10

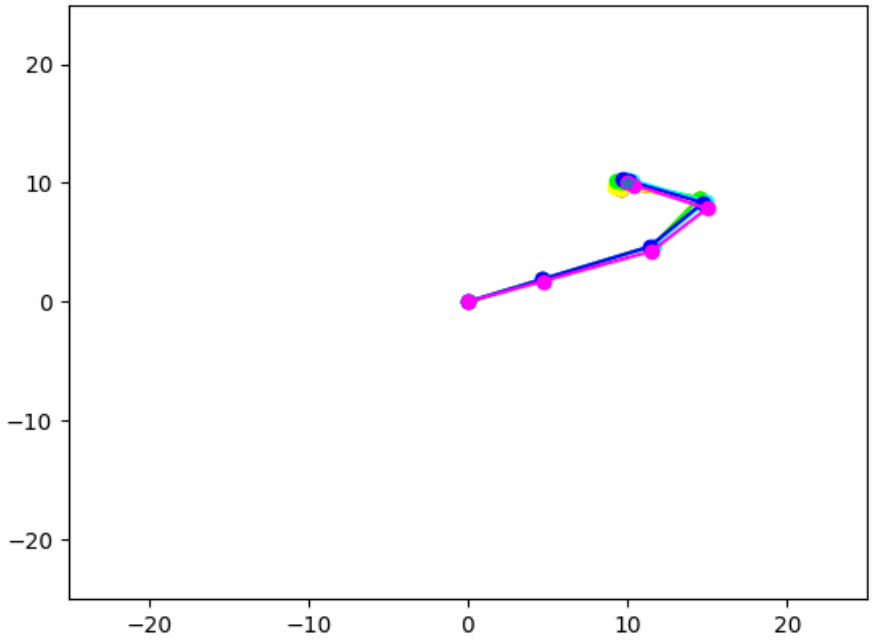

Figura 3.7: Última iteración para objetivo 10,10

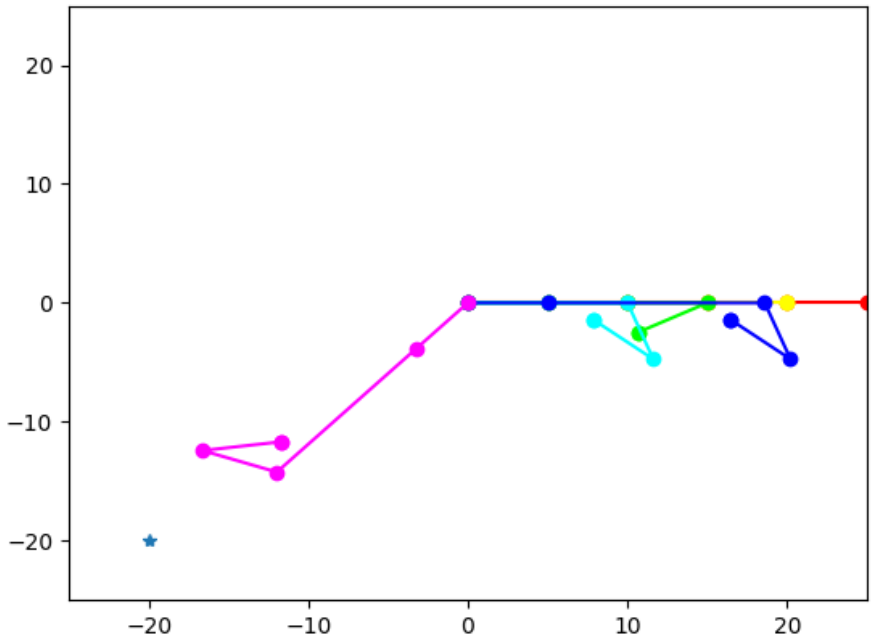

Figura 3.8: Primera iteración para objetivo -20,-20

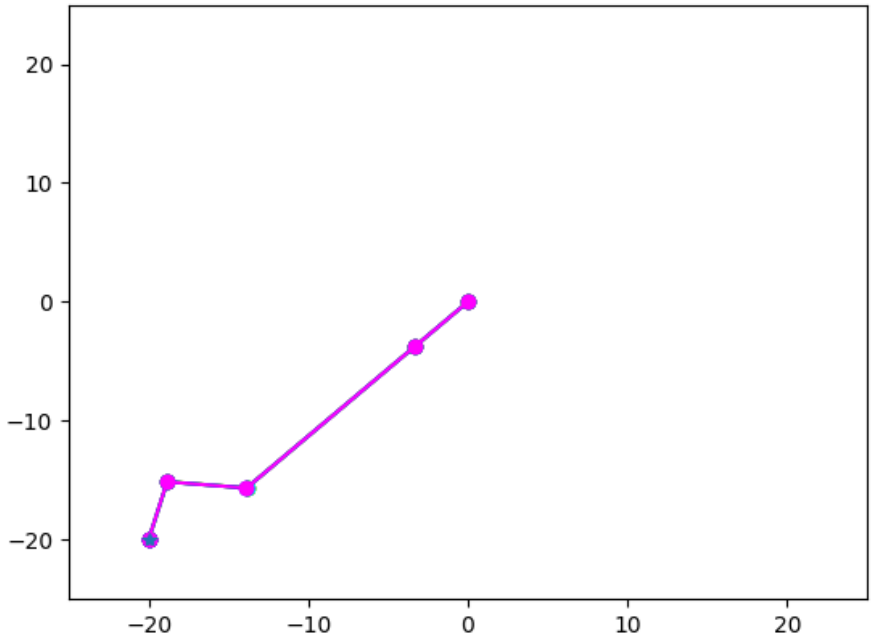

Figura 3.9: Última iteración para objetivo -20,-20

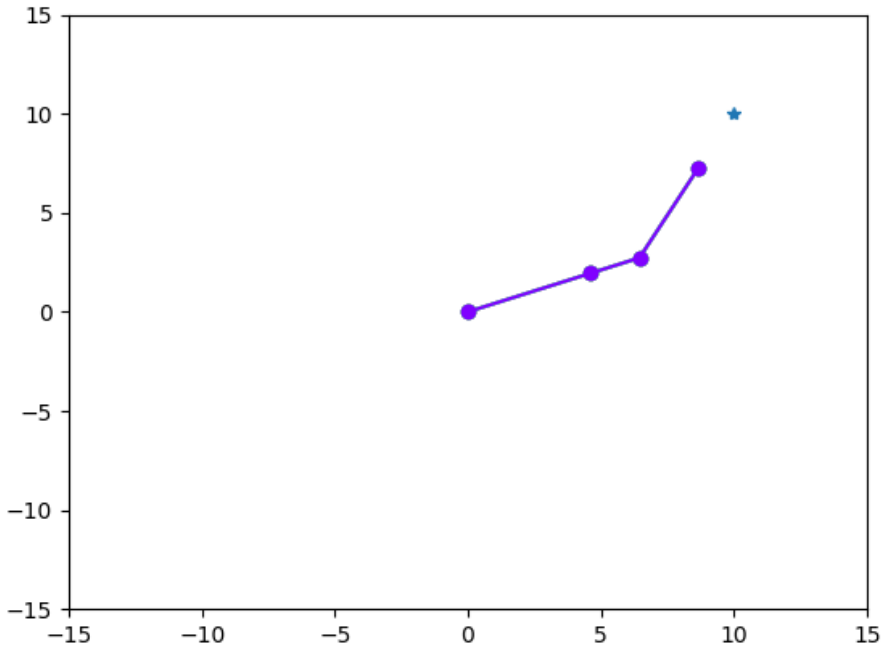

Figura 3.10: Última iteración para objetivo 10,10

```
- Iteracion 5:
Origenes de coordenadas:
(O0)0    = [0, 0]
(O1)0    = [4.605, 1.947]
(O2)0    = [6.447, 2.726]
(O3)0    = [8.642, 7.219]

Distancia al objetivo = 3.09524

No hay convergencia tras 6 iteraciones.
- Umbral de convergencia epsilon: 0.01
- Distancia al objetivo:          3.09524
- Valores finales de las articulaciones:
  theta1 = 0.4
  theta2 = 0
  theta3 = 0.716
  L1     = 5
  L2     = 2
  L3     = 5
(base) jr@jr-All-Series:~/Downloads$
```

Figura 3.11: Salida por consola cuando no converge

Capítulo 4

Localización

En esta práctica se aborda el problema de localización de un robot móvil. Se considera un robot con un sistema sensor, que tiene un cierto error de medición, y unas balizas, que son los puntos respecto a los que se toman las mediciones. En este caso, las balizas son a la vez los objetivos, es decir, describen la ruta que debe seguir el robot. Para esta simulación se calcula la predicción o modelo de la posición del robot, en base al movimiento que se realiza en los actuadores. No obstante, este movimiento tiene un cierto ruido, por lo que la posición real del robot dista de la posición ideal o estimada. Por tanto, se deben emplear las mediciones para corregir el modelo y ajustarlo a la realidad.

4.1. Código Implementado

El script proporcionado contiene muchas de las funciones necesarias, sin embargo, se debe implementar código para poder calcular la localización.

4.1.1. Análisis

Se implementan las siguientes características:

- `Función de localización`: Dadas las balizas, el robot ideal, el robot real y las mediciones, se toma un punto de búsqueda a partir del cual se explora un área cuadrada comprendida entre -radio y radio, usando un incremento indicado como parámetro. Se comprueban todas las posiciones posibles dentro de ese área, y se calcula la probabilidad de cada una en función de las mediciones del sensor y las posiciones de las balizas. Por último, se devuelve la posición con mayor probabilidad. Esta función se puede observar en la figura 4.1.

- `Localización inicial`: Para comenzar la localización, se debe determinar la posición en la que comienza el robot. 4.3.

- `Ajuste de la posición`: Mientras el robot se mueve, se debe verificar si la posición predicha es correcta, para ello se comprueba la probabilidad de las mediciones de los sensores respecto a la posición predicha. Se establece un umbral de error, que al ser superado causa un ajuste de la posición del modelo. Para ello, se emplea la función de localización descrita, tomando un radio de $2 * error\_mediciones$. 4.4.

En la figura 4.5 se observan algunos de los parámetros que se han indicado del script, más adelante se experimentará con ellos.

```python
# Centro = vector [x, y]
def localizacion(balizas, real, ideal, centro, radio, limite_incremento, incremento_angulo, mostrar=0):
  # Buscar la localizaci@n m@s probable del robot, a partir de su sistema
  # sensorial, dentro de una regi@n cuadrada de centro "centro" y lado "2*radio".
  mejor_pose = ideal.pose()
  error_mejor_pose = 1000 # Valor arbitrario alto

  # B@squeda en "pir@mide", cada iteraci@n elige una cuadr@cula
  incremento = radio
  while incremento > limite_incremento:
    imagen = []
    print("Radio: ", radio, "Incremento: ", incremento)
    # Buscar mejores coordenadas
    for y in np.arange(-radio, radio, incremento):
      imagen.append([])
      for x in np.arange(-radio, radio, incremento):
        ideal.set(centro[0] + x + radio/2, centro[1] + y + radio/2, ideal.orientation)
        error_actual = real.measurement_prob(ideal.sense(balizas), balizas)
        imagen[-1].append(error_actual)

        if error_actual < error_mejor_pose:
          error_mejor_pose = error_actual
          mejor_pose = ideal.pose()

    if mostrar:
      #plt.ion() # modo interactivo
      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(mejor_pose[0],mejor_pose[1],'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()

    # Actualizamos la "pir@mide"
    centro = [mejor_pose[0], mejor_pose[1]]
    radio /= 2
    incremento = radio
```

Figura 4.1: Primera parte de la función de localización

4.1.2. Complejidad

Los cálculos del algoritmo de localización se deben realizar al inicio, y luego cuando se supera el umbral de error. Por tanto, el rendimiento del programa dependerá del ruido en el movimiento, así como del ruido de los sensores, y principalmente, del umbral de error considerado. Un mayor umbral resulta en menos cálculos, mientras que un umbral menor requiere una mayor precisión, y por tanto, más cálculos. Se debe tener cuidado, puesto que un umbral demasiado bajo puede causar cálculos excesivos debido a los ruidos de sensores.

Respecto a la función de localización, su complejidad depende del radio y el incremento considerado. Un menor incremento realiza un barrido más exhaustivo del área, pero requiere más cálculos. Por tanto, es importante elegir un valor que consiga un balance entre precisión y rendimiento.

4.2. Mejoras

En adición al comportamiento básico, se han implementado varias mejoras:

```
102
103     # Buscar mejor orientaci@n
104     mejor_orientacion = ideal.orientation
105     for angulo in np.arange(-pi, pi, incremento_angulo):
106       ideal.set(mejor_pose[0], mejor_pose[1], angulo)
107       error_actual = real.measurement_prob(ideal.sense(balizas), balizas)
108
109       if error_actual < error_mejor_pose:
110         error_mejor_pose = error_actual
111         mejor_pose = ideal.pose()
112
113     print("Error mejor pose: ", error_mejor_pose)
114     return mejor_pose
115
```

Figura 4.2: Fin de la función de localización

```
172 ####################
173 # Exploracion inicial#
174 ####################
175 # Se toma un radio seg@n las coordenadas m@ximas y m@nimas de balizas + margen
176
177 # Obtener las componentes por separado
178 valores_x, valores_y = zip(*objetivos)
179
180 # Calcular los m@nimos y m@ximos
181 min_x, max_x = min(valores_x), max(valores_x)
182 min_y, max_y = min(valores_y), max(valores_y)
183
184 centro_inicial = [(min_x + max_x) / 2, (min_y + max_y) / 2]
185 # Calculamos el radio como el m@ximo entre la distancia vertical u horizontal
186 radio_inicial = max(abs(max_x - min_x), abs(max_y - min_y))/2 + MARGEN
187 # radio_inicial = sqrt((max_x - min_x)**2 + (max_y - min_y)**2) / 2 + MARGEN
188
189 inicial = localizacion(objetivos, real, ideal, centro_inicial, radio_inicial, LIMITE_INCREMENTO, INCREMENTO_ANGULO, 1)
190 ideal.set(*inicial)
191
```

Figura 4.3: Localización inical

4.2.1. Ajuste de la orientación

Además del ajuste de la posición, se implementa un cálculo similar para detectar la orientación (rotación) más probable. Se realiza un barrido entre $-\pi$ y π con un incremento dependiente de un parámetro, y se calcula la probabilidad de cada orientación en función de las mediciones y la posición previamente predicha. El código correspondiente se aprecia en la figura 4.2.

4.2.2. Uso de incremento piramidal

En lugar del comportamiento descrito previamente, en el que se emplea un incremento constante para el barrido del área a estudiar, se emplea un algoritmo de barrido piramidal, en el que en cada iteración se divide el área en 4 cuadrantes, estudiándose el centro de cada uno, y seleccionando el cuadrante con mayor probabilidad. Este proceso se repite hasta que los cuadrantes tienen un tamaño inferior a un límite establecido. El código correspondiente se muestra en la figura 4.1. Cabe destacar que esta forma es mucho más eficiente para estudiar el área, puesto que se evitan cálculos innecesarios. Sin embargo, se puede perder un poco de precisión ya que es posible que se seleccione un cuadrante erróneo si el punto se sitúa en el extremo entre dos, debido a los ruidos.

```python
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())

    # Comparar baliza real e ideal; si la diferencia es grande, localizar
    diferencia_mediciones = real.measurement_prob(ideal.sense(objetivos), objetivos)
    print("Medicion: ", diferencia_mediciones)
    # Si la diferencia es grande, localizar
    if (diferencia_mediciones > UMBRAl_DIF_TRAYECTORIA):
      # radio de localizaci\ufffdn = 2 * diferencia
      ideal.set(*localizacion(objetivos, real, ideal, [ideal.x, ideal.y], 2*diferencia_mediciones, LIMITE_INCREMENTO, INCREMENTO_ANGULO)) # actualizar pose

    espacio += v
    tiempo  += 1
```

Figura 4.4: Corrección de la posición

```python
# Parametros
UMBRAl_DIF_TRAYECTORIA = 0.1 # Diferencia entre mediciones ideales y reales a partir de la que se corrige la pose ideal
LIMITE_INCREMENTO = 0.05 # Precisi\ufffdn al estimar la localizaci\ufffdn
INCREMENTO_ANGULO = 0.01 # incremento al probar la mejor orientacion
MARGEN = 1 # Borde que se a\ufffdade en la localizaci\ufffdn inical sobre el area de las balizas
# RADIO = 1 # Constante arbitraria

# Definici\ufffdn del robot:
P_INICIAL = [0.,4.,0.] # Pose inicial (posicion y orientacion)
V_LINEAL  = .7         # Velocidad lineal    (m/s)
V_ANGULAR = 140.       # Velocidad angular   (\ufffd/s)
FPS       = 10.        # Resolucion temporal (fps)

HOLONOMICO = 1
GIROPARADO = 0
LONGITUD   = .2
```

Figura 4.5: Parámetros de la localización

4.2.3. Cálculo de un radio inicial en función de las balizas y margen

La alternativa inicial consiste en establecer un radio arbitrario para el primer barrido. En su lugar, se decide implementar un cálculo del cuadrado que contiene a las balizas, mediante la selección de los mínimos y máximos de las componentes de sus coordenadas. A este área se le suma un cierto margen en el que puede encontrarse el robot. De esta forma se consigue una aproximación más precisa en función de la geometría de las balizas.

4.2.4. Propuestas

En adición a las mejoras implementadas, se proponen las siguientes:

- `Lectura de los datos de configuración desde un JSON`: Actualmente los parámetros se encuentran en el propio script, lo que dificulta su modificación. Se propone implementar un archivo JSON en el que se almacenen los parámetros, de forma que se puedan modificar sin necesidad de acceder al código.
- `Visualización de múltiples trayectorias en una misma ejecución`: Sería valioso poder comparar varias trayectorias, ya sea de distintas ejecuciones con los mismos parámetros, debido a los ruidos, o para comparar distintas configuraciones.

4.3. Ejemplos de ejecución

4.3.1. Ejemplo 1

Se estudia la ejecución para un caso de 4 balizas, con un límite de incremento para la localización de 0.05 y un umbral para la corrección de trayectoria de 0.1. El resto de parámetros se mantienen en sus valores por defecto.

En primer lugar analizamos la localización inicial. En la figura 4.6 se observa la primera iteración. Se distinguen los 4 cuadrantes y se muestra en tonos morados el que tiene mayor probabilidad. Además, se observa la posición real del robot con un diamante verde, y la posición estimada, que corresponde con el centro del cuadrante, con un diamante violeta. A continuación, en la figura 4.7 se muestra la última iteración, en la que se observa que la posición estimada está bastante cerca de la real, (4.0625, -0.0650) vs (4.0, 0.0) respectivamente. Además observamos que en esta iteración la probabilidad de los centros de los cuadrantes es menor que la de probabilidad de la iteración previa, por lo que la posición estimada se mantiene sin cambios.

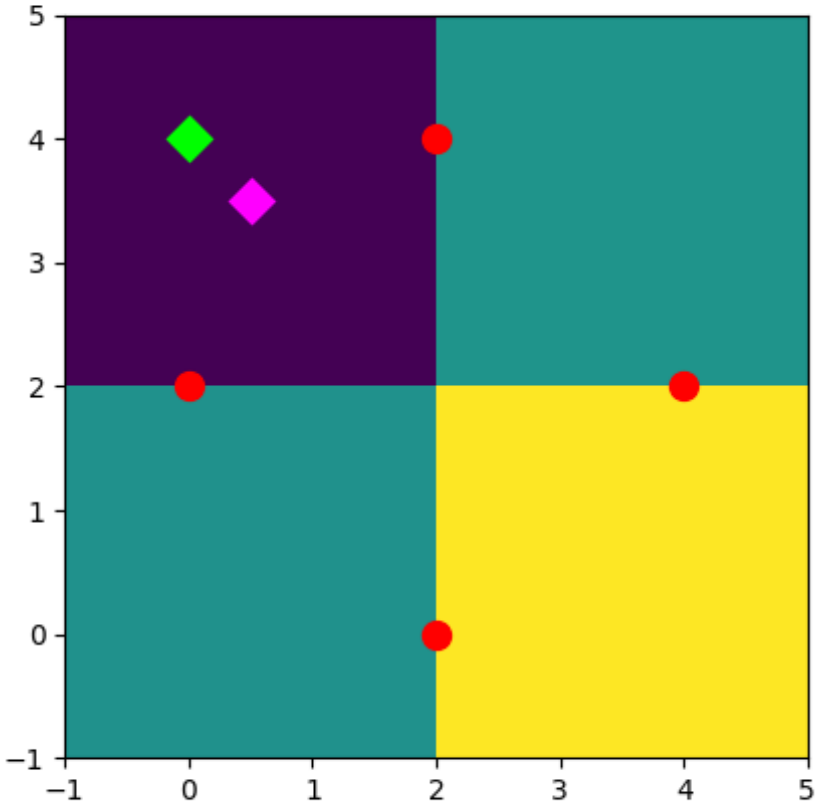

Figura 4.6: Primera iteración de la localización inicial

En la figura 4.8 se observa la evolución de la posición real del robot en rojo, así como las correcciones de la posición ideal, en verde. Vemos como cuando la trayectoria predicha se separa lo suficiente de la real, se realiza una corrección.

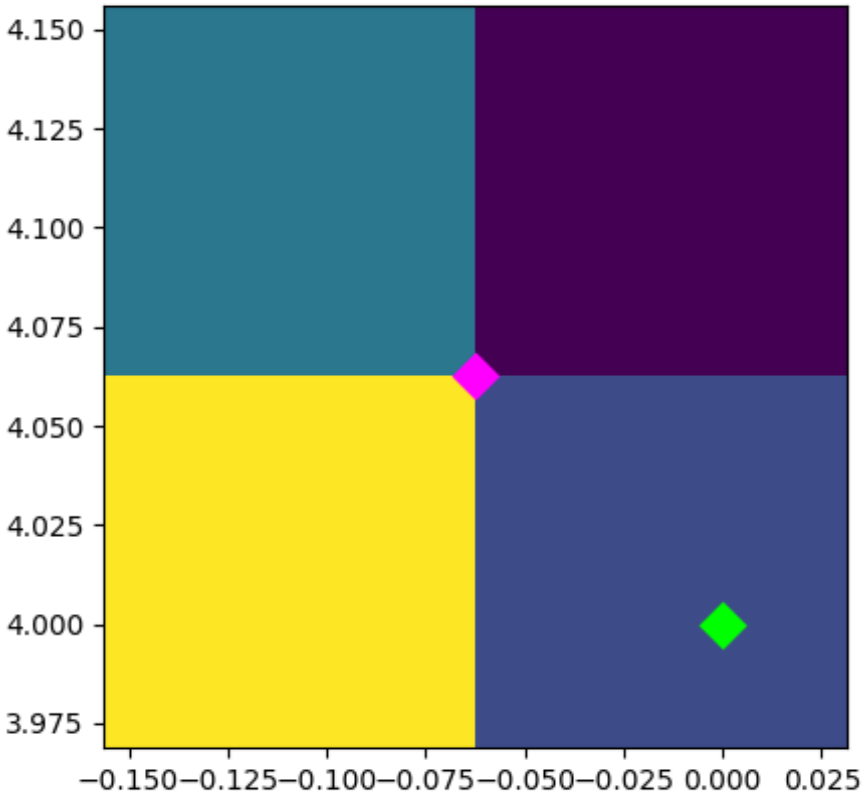

Figura 4.7: Última iteración de la localización inicial

4.3.2. Ejemplo 2

Mantenemos los valores del experimento anterior, pero disminuimos el umbral a 0.01, lo que debería aumentar la precisión de la trayectoria ideal. En la figura 4.9 observamos el resultado de ejecución. Cabe destacar que se han realizado giros alrededor de todas las balizas, incluso varios en el caso de la baliza izquierda. Esto es debido a que la trayectoria predicha se ha quedado a una distancia mayor al umbral de las balizas, por lo que se ha tenido que ajustar la trayectoria real para aseguarse de que pasa por las balizas. Además, observamos que la trayectoria ideal, en verde, se reajusta más veces que en el ejemplo 1.

4.3.3. Ejemplo 3

Ahora consideramos la configuración del ejemplo 2, pero aumentamos la velocidad angular a 240º por segundo (por defecto estaba a 140º/s). En la figura 4.10 se observa la trayectoria del robot. Constatamos que los giros son más cerrados.

4.3.4. Ejemplo 4

En este caso mantenemos los valores del ejemplo 2, pero reducimos la velocidad angular a 40º por segundo. En la figura 4.11 se observa la trayectoria del robot. Vemos que ahora los giros son más abiertos, y en consecuencia, el robot tarda más en realizar el recorrido y requiere computar más posiciones.

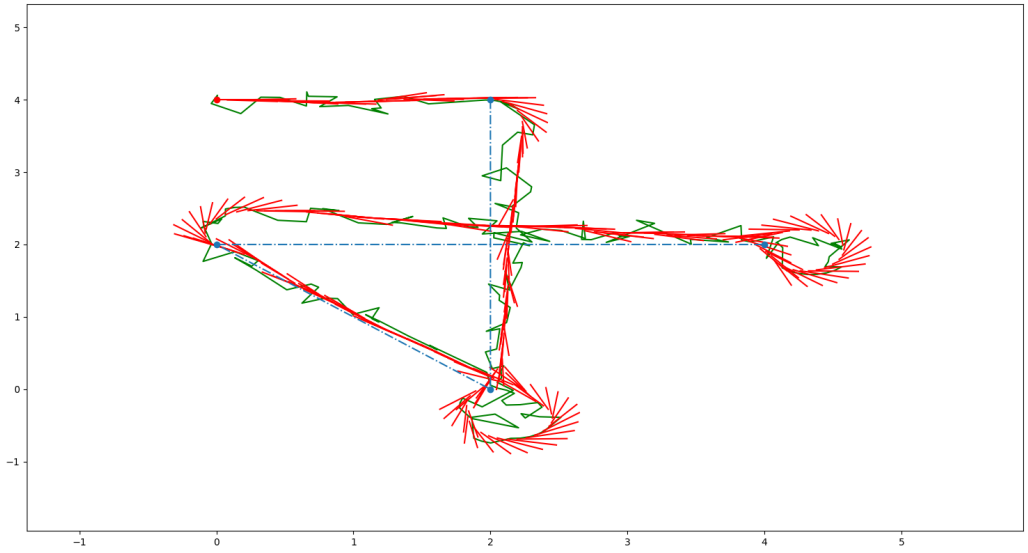

Figura 4.8: Trayectoria del ejemplo 1

4.4. Conclusions

We have implemented an algorithm for modelling the position and orientation (the pose) of a mobile robot, considering the predicted pose and making adjustments based on sensor measurements regarding the position of beacons. We have made improvements such as using a variable increment for the search area through a pyramidal approach.

Also, we have considered the complexity of the algorithm, which depends on the noise in the movement and sensors, as well as the threshold for the error. A lower threshold requires more calculations, so it is important to find a balance between precision and performance. This has been proved experimentally. Additionally, we have also tested the impact of the angular velocity on the robot's trajectory, seing that a higher angular velocity results in tighter turns whereas a lower angular velocity results in wider turns and usually a longer trajectory.

Although the algorithm has performed well on the experiments, it has a relatively high computational cost, and a more efficient approach will be discussed in the next chapter, the particle filter.

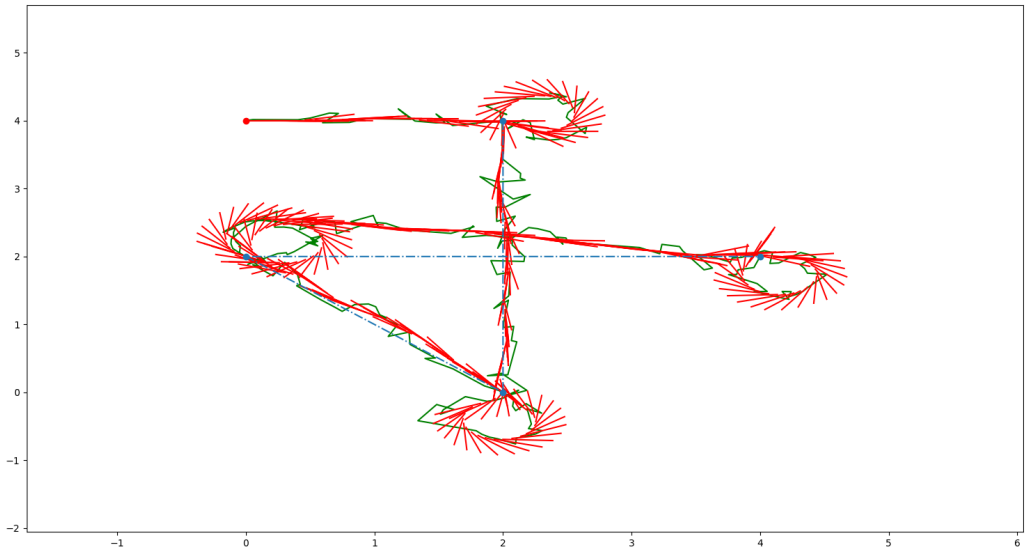

Figura 4.9: Trayectoria del ejemplo 2

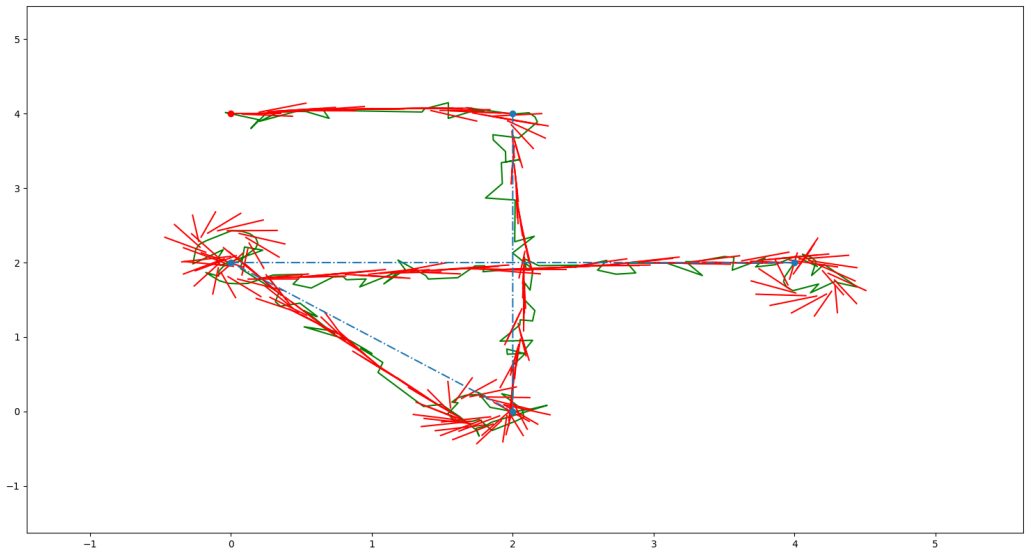

Figura 4.10: Trayectoria del ejemplo 3

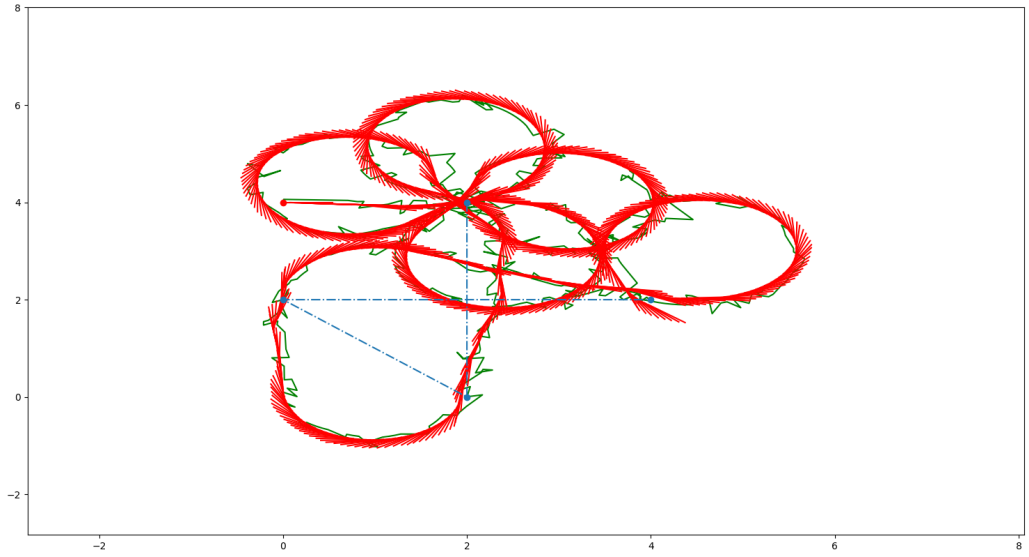

Figura 4.11: Trayectoria del ejemplo 4

Capítulo 5

Filtro de Partículas

En esta práctica se aborda el filtro de partículas. Se trata de un algoritmo eficiente para calcular la localización de un robot móvil que dispone de sensores de distancia.

5.1. Código Implementado

El algoritmo se basa en la generación de una serie de partículas empleando como centro la posición ideal (posición estimada) y siguiendo una distribución de probabilidad sobre un radio indicado como parámetro. Sobre esta distribución inicial, iterativamente se aplica el movimiento del robot y se actualiza la probabilidad de cada partícula, tras lo que se realiza un remuestreo, en el que se escogen las partículas con mayor probabilidad.

5.1.1. Análisis

El código proporcionado contiene muchas de las funciones necesarias. Para el correcto funcionamiento se implementan las siguientes funciones:

- `genera_filtro`: Se inicializa el filtro de partículas con el tamaño indicado, para ello se crea un array de robots, siendo cada uno una partícula. Se inicializa cada partícula con un ruido arbitrario, y se asigna una posición siguiendo una distribución uniforme en el área descrita por el centro y el radio indicados. Adicionalmente, la orientación se establece añadiendo un ruido gaussiano a la orientación ideal. Por último, se establece la probabilidad de cada partícula en base a las mediciones y las balizas.

- `dispersion`: Considera la dispersión espacial de las partículas, para ello, se retorna un array conteniendo los valores mínimos y máximos de las componentes x e y de las posiciones de todas las partículas.

- `peso_medio`: Se calcula el peso medio empleando la suma de todos los pesos para normalizar el de cada partícula.

Todas estas funciones se observan en las figuras 5.1, 5.2 y 5.3.

Además, en el programa principal se codifican los siguientes comportamientos:

- `inicialización del filtro`: se crea el filtro empleando la función *genera_filtro*, usando el número de partículas inicial indicado como parámetro. Se toma además como centro la posición inical del robot, y se emplea un radio 1. A continuación

```python
def genera_filtro(num_particulas, balizas, real, centro=[2,2], radio=3):
  # Inicialización de un filtro de tamaño 'num_particulas', cuyas partículas
  # imitan a la muestra dada y se distribuyen aleatoriamente sobre un área dada.

  # Creamos tantos robots como partículas y establecemos el ruido
  filtro = [robot() for particula in range(num_particulas)]
  for particula in filtro:
    particula.set_noise(0.01,0.01,0.01)

  # Colocamos las partículas en una posición aleatoria con el centro y radio. Además se orientan.
  for particula in filtro:
    x = random.uniform(centro[0] - radio, centro[0] + radio)
    y = random.uniform(centro[1] - radio, centro[1] + radio)
    particula.set(x,y, real.orientation + random.gauss(0,0.1))

  # El peso de cada particula corresponde a las mediciones (mayor peso = menos error/diferencia entre mediciones)
  for particula in filtro:
    particula.weight = particula.measurement_prob(real.sense(balizas), balizas)

  return filtro
```

Figura 5.1: Función *genera_filtro*

```python
def dispersion(filtro):
  # Dispersion espacial del filtro de particulas
  valores_x = []
  valores_y = []
  # Obtener las componentes por separado
  for particula in filtro:
    valores_x.append(particula.x)
    valores_y.append(particula.y)

  # Calcular los mínimos y máximos
  min_x, max_x = min(valores_x), max(valores_x)
  min_y, max_y = min(valores_y), max(valores_y)
  return [min_x, max_x, min_y, max_y]
```

Figura 5.2: Función *dispersion*

se establece como posición inicial la partícula más probable, y se inicializan la trayectoria ideal y real. El código de esta sección se muestra en la figura 5.4.

- `actualización de la trayectoria y las probabilidades`: se mueven todas las partículas del filtro con el vector de movimiento calculado para dirigirse hacia el objetivo. A continuación, se actualizan los pesos con las nuevas probabilidades, y se normalizan empleando la función *peso_medio*. Finalmente se actualiza la trayectoria real e ideal. El código de este fragmento así como del remuestreo se encuentra en la figura 5.5.
- `remuestreo`: se emplea la función ya definida resample", que toma como parámetros el filtro y el número de partículas deseado.

```python
def peso_medio(filtro):
  # Sumar todos los pesos
  peso_total = sum(particula.weight for particula in filtro)

  # Dividir cada peso entre el total
  for particula in filtro:
    particula.weight /= peso_total

  return filtro
```

Figura 5.3: Función *peso_medio*

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

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

#inicialización del filtro de partículas y de la trayectoria
################################################
filtro = genera_filtro(N_INICIAL, objetivos, real, centro=[P_INICIAL[0], P_INICIAL[1]], radio=1)
pose = hipotesis(filtro) # Cogemos la partícula más probable
trayectoria = [pose]    # Iniciamos la trayectoria como la partícula más probable

trayectreal = [real.pose()]
```

Figura 5.4: Inicialización del filtro

5.1.2. Complejidad

El costo computacional del algoritmo depende principalmente del número de partículas que se mantienen en el filtro, el cuál es uno de los principales parámetros a ajustar. En general, el algoritmo es lineal en el número de partículas, ya que en cada iteración se actualiza la posición y probabilidad de cada partícula.

5.2. Mejoras

No se han implementado mejoras, pero se proponen las siguientes:

- `Número variable de partículas`: Emplear un enfoque adaptativo, en el que el número de partículas se ajuste en función de la precisión del modelo (la probabilidad de que la posición predicha sea correcta según las mediciones).
- `Lectura de datos de configuración desde un JSON`: al igual que en prácticas anteriores, se podría emplear un fichero de configuración para evitar tener que modificar el código cuando se quieren alterar parámetros como el número de partículas, el radio de dispersión, etc.
- `Visualización de probabilidades`: resultaría práctico que las partículas se mostraran con un color que indicara su probabilidad.
- `Mejoras del remuestreo`: ahora mismo se implementa un remuestreo básico, sería interesante considerar aproximaciones más inteligentes, como por ejemplo, remuestreo estratificado, en el que se busca mejorar la representatividad de las partículas escogidas.

5.3. Ejemplos de ejecución

Se han realizado varios experimentos sobre la trayectoria con 3 balizas.

5.3.1. Ejemplo 1

Realizamos una ejecución para comprobar el funcionamiento con los valores por defecto, destacando que el tamaño inicial del filtro es de 2000 partículas, y el tamaño el resto de las iteraciones es de 50.

Observamos en la figura 5.6 el inicio del experimento, en el que se observa la posición inicial del robot y las partículas generadas. En la figura 5.7 se muestra la segunda iteración, en la que se observa la reducción del número de partículas. En la figura 5.8 se muestra el progreso del experimento, destacando la existencia de tres grupos de partículas, unas por encima de la posición real, otras simétricamente por debajo, y el resto en la posición real. Esto se debe a la ambigüedad de las mediciones con la configuración de las balizas. En la figura 5.9 se muestra la convergencia del experimento, en la que se observa que las partículas se concentran en la posición real del robot. Por último, en la figura 5.10 se muestra la trayectoria del robot, en la que se observa que el filtro de partículas ha sido capaz de seguir la trayectoria real del robot.

5.3.2. Ejemplo 2

Se repite el experimento anterior, manteniendo el número de partículas inicial, pero con un tamaño de 100 del filtro de partículas en ejecución. En la figura 5.11 se observa que en las primeras iteraciones ya no existe la ambigüedad del ejemplo previo, sino que se ve un grupo cercano de partículas. En la figura 5.12 se observa la convergencia del experimento, que ha tenido lugar más rápido que en el anterior ejemplo. Por último, en la figura 5.13 se muestra la trayectoria del robot, en la que se observa que el filtro de partículas ha sido capaz de seguir la trayectoria real del robot con mayor precisión, no obstante, cabe destacar que el coste computacional ha sido mayor, el experimento ha tardado más tiempo.

5.3.3. Ejemplo 3

A continuación se repite el experimento pero para un filtro con tamaño 10. En la figura 5.14 se observa la trayectoria del experimento. Destaca que la precisión es mucho menor a los experimentos anteriores, y mientras que se converge rápidamente, se pierde la trayectoria real del robot en varias ocasiones.

5.4. Conclusions

We have implemented a simple particle filter and proved its functionality with some examples. We have studied the computational complexity of the algorithm and seen that a higher particle count leads to improved precision, but also increases the computational cost. It seems that 50 particles is a good balance between precision and performance. It is also worth noting that this algorithm is a fast approach for localization of a moving robot, being faster than the approach described in chapter 4.

```
tiempo  = 0.
espacio = 0.
for punto in objetivos:
  while distancia(trayectoria[-1],punto) > EPSILON and len(trayectoria) <= 1000:

    #seleccionar pose (usando la partícula más probable)
    pose = hipotesis(filtro)

    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
      real.move(w,v)
    else:
      real.move_triciclo(w,v,LONGITUD)


    # Seleccionar hipótesis de localización y actualizar la trayectoria

    # Movemos todas las partículas
    for particula in filtro:
      if HOLONOMICO:
        if GIROPARADO and abs(w) > .01: v = 0
        particula.move(w, v)
      else:
        particula.move_triciclo(w, v, LONGITUD)

    # Recalcular pesos
    for part in filtro:
      part.weight = part.measurement_prob(real.sense(objetivos), objetivos)
    filtro = peso_medio(filtro)

    trayectoria.append(pose)
    trayectreal.append(real.pose())
    mostrar(objetivos,trayectoria,trayectreal,filtro)

    # remuestreo
    filtro = resample(filtro, N_PARTIC)

    espacio += v
    tiempo  += 1

if len(trayectoria) > 1000:
  print ("<< ! >> Puede que no se haya alcanzado la posicion final.")
print ("Recorrido: "+str(round(espacio,3))+"m / "+str(tiempo/FPS)+"s" )
print ("Error medio de la trayectoria: "+str(round(sum(\
    [distancia(trayectoria[i],trayectreal[i])\
    for i in range(len(trayectoria))])/tiempo,3))+"m" )
input()
```

Figura 5.5: Actualización del filtro y remuestreo

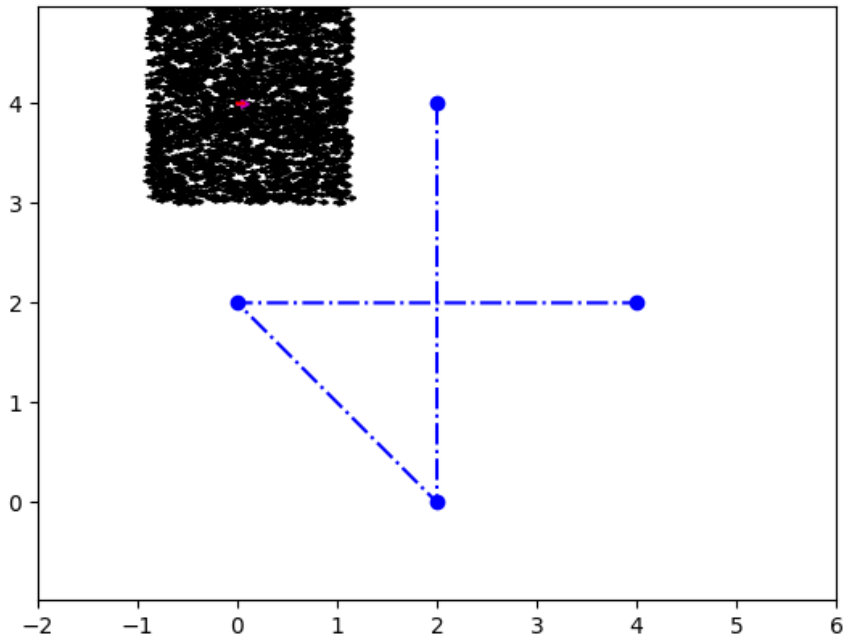

Figura 5.6: Inicio del ejemplo 1

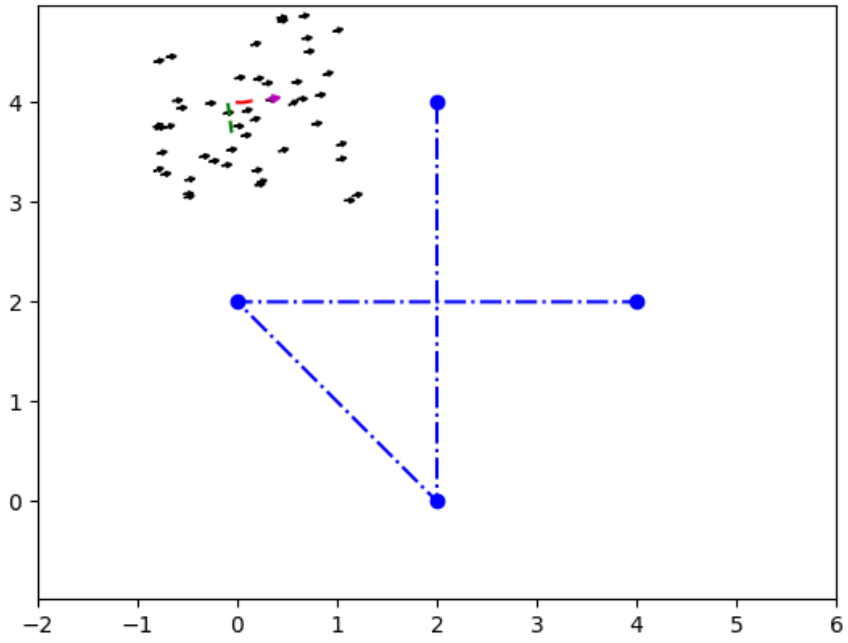

Figura 5.7: Segunda iteración del ejemplo 1

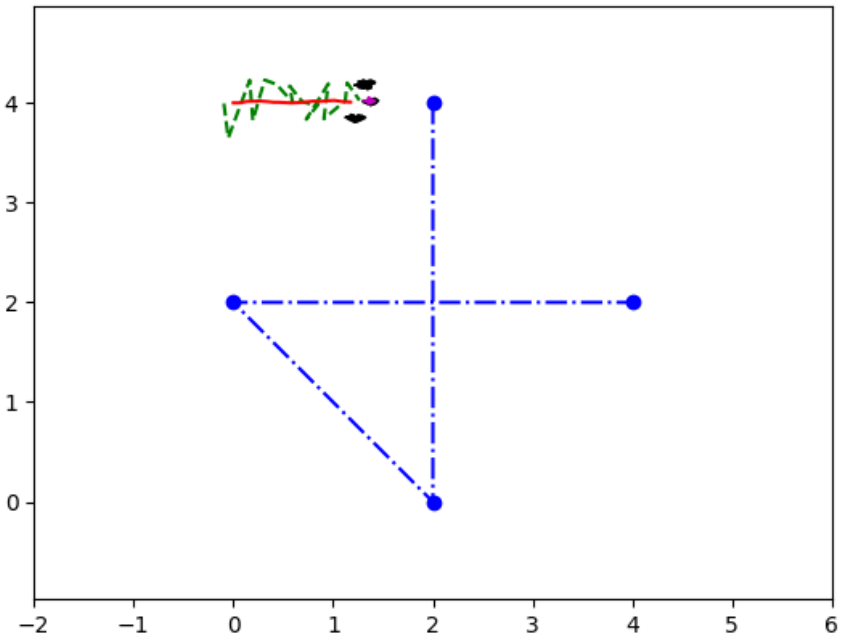

Figura 5.8: Progreso del ejemplo 1

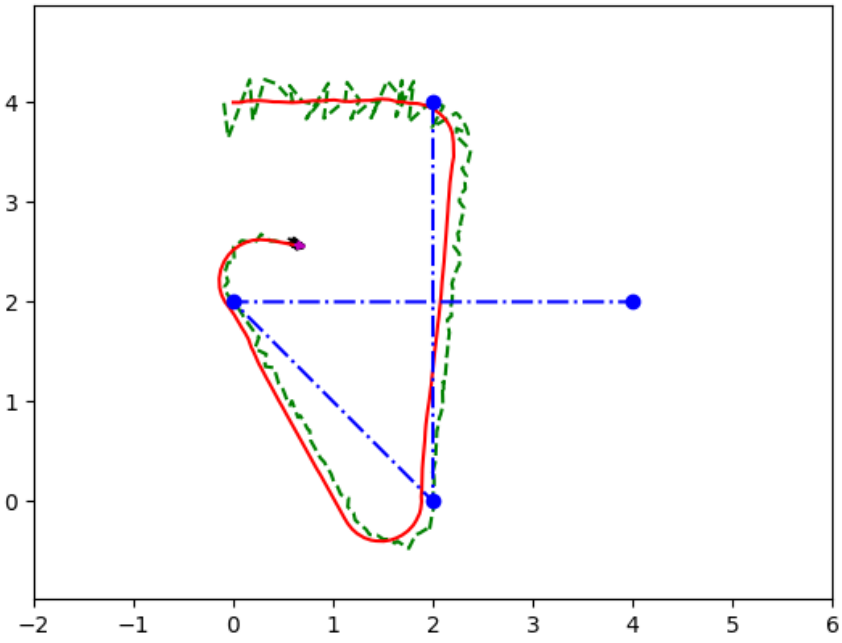

Figura 5.9: Convergencia del ejemplo 1

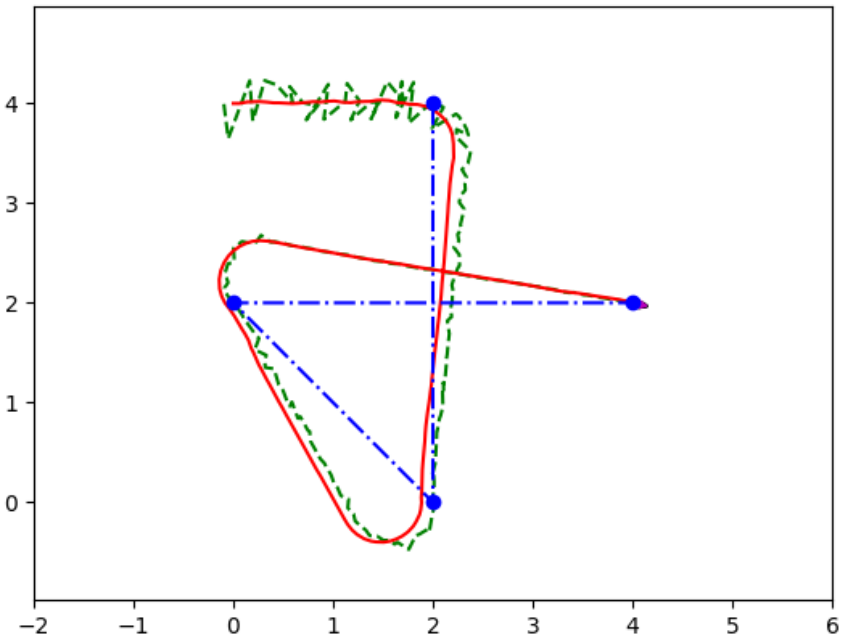

Figura 5.10: Trayectoria del ejemplo 1

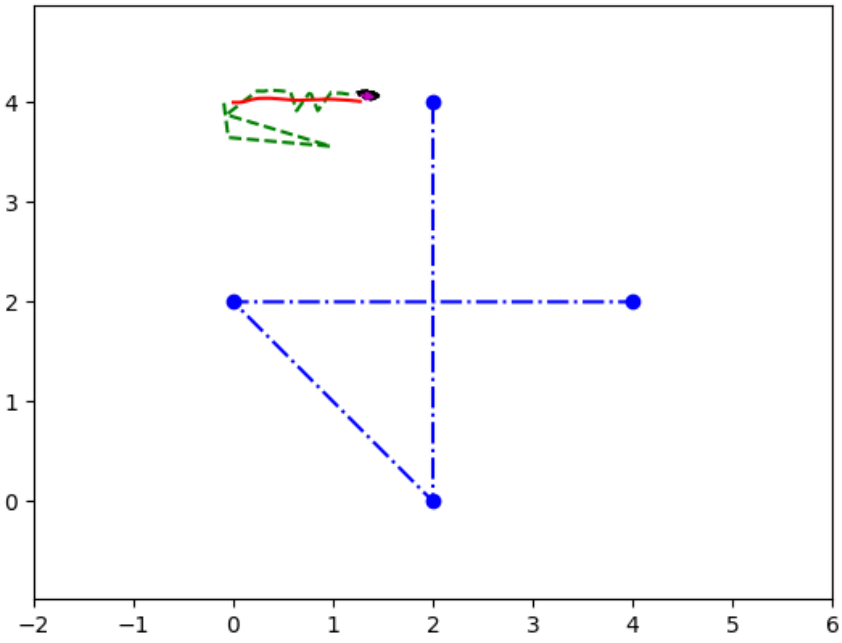

Figura 5.11: Inicio del ejemplo 2

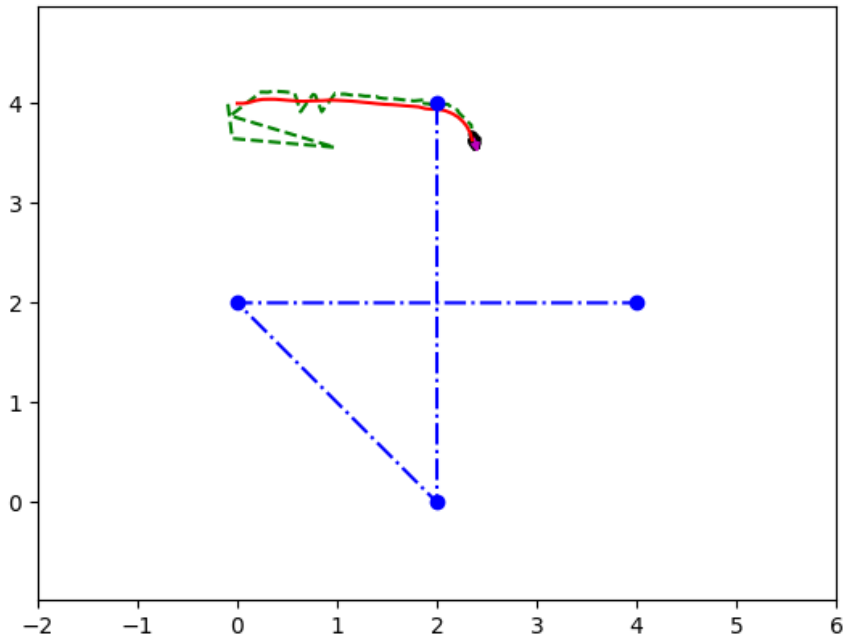

Figura 5.12: Convergencia del ejemplo 2

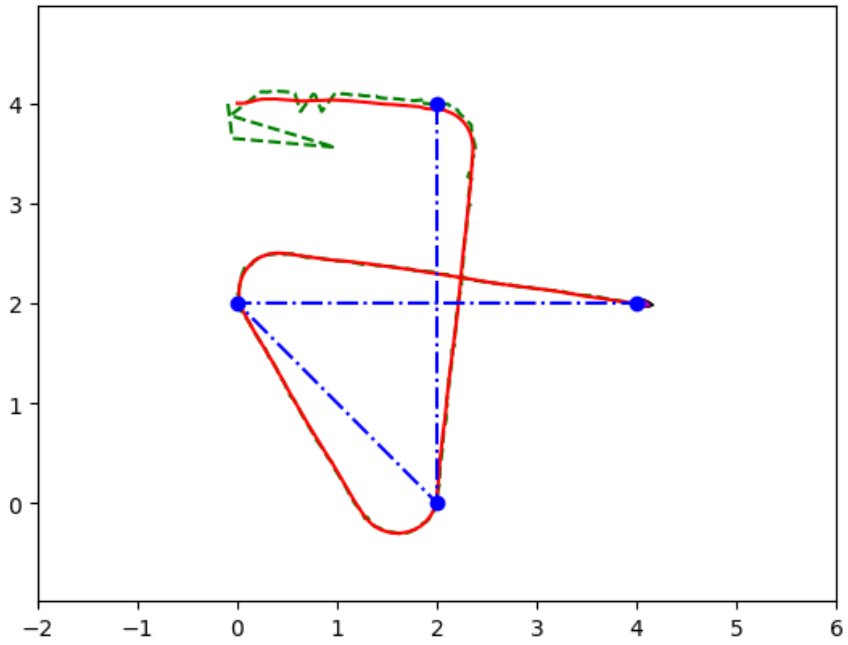

Figura 5.13: Trayectoria del ejemplo 2

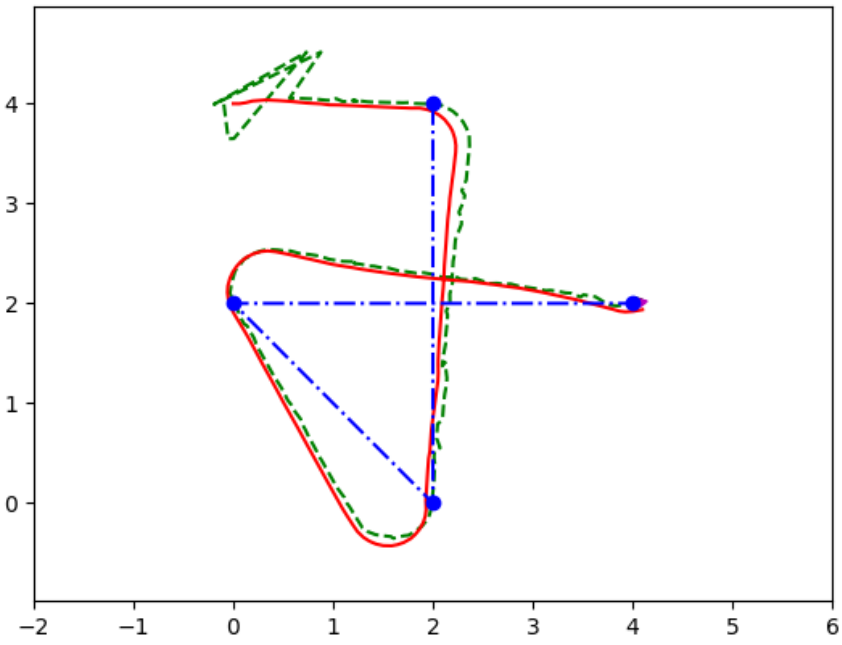

Figura 5.14: Trayectoria del ejemplo 3