Practice Report

Robótica Computacional

Cheuk Kelly Ng Pante (alu0101364544@ull.edu.es)

December 29, 2023

Contents

1 Forward kinematics **1**

2 Inverse kinematics **4**
2.1 Example execution 6

3 Localization **7**
3.1 Localization code 8
3.2 Example execution 9

1 Forward kinematics

Forward kinematics is a branch of robotics and mechanics that deals with the relationship between the movements of the links of a robot and the variables that control them. In other words, forward kinematics is the problem of finding the position and orientation of the end of the robot, given the set of parameters that define the positions and orientations of all the links.

To explain the forward kinematics, the Denavit-Hartenberg (DH) coordinate system will be used. The Denavit-Hartenberg coordinate system is a coordinate system used to model direct and inverse kinematics of articulated robots. The Denavit-Hartenberg coordinate system is based on four parameters associated with each joint. These parameters are:

- d_i: Distance between the z_{i-1} and z_i axes along the x_{i-1} axis.
- θ_i: Angle between the z_{i-1} and z_i axes around the x_{i-1} axis.
- a_i: Distance between the x_{i-1} and x_i axes along the z_{i-1} axis.
- α_i: Angle between the x_{i-1} and x_i axes around the z_{i-1} axis.

The DH parameters can be calculated according to the following table:

	A	**B**	**C**
d_i	O_{i-1}	$Z_{i-1} \cap X_i$	Z_{i-1}
θ_i	X_{i-1}	X_i	Z_{i-1}
a_i	$Z_{i-1} \cap X_i$	O_i	X_i
α_i	Z_{i-1}	Z_i	X_i

Table 1.1: Parameters of DH

For the explanation of forward kinematics, manipulator 3 will be used:

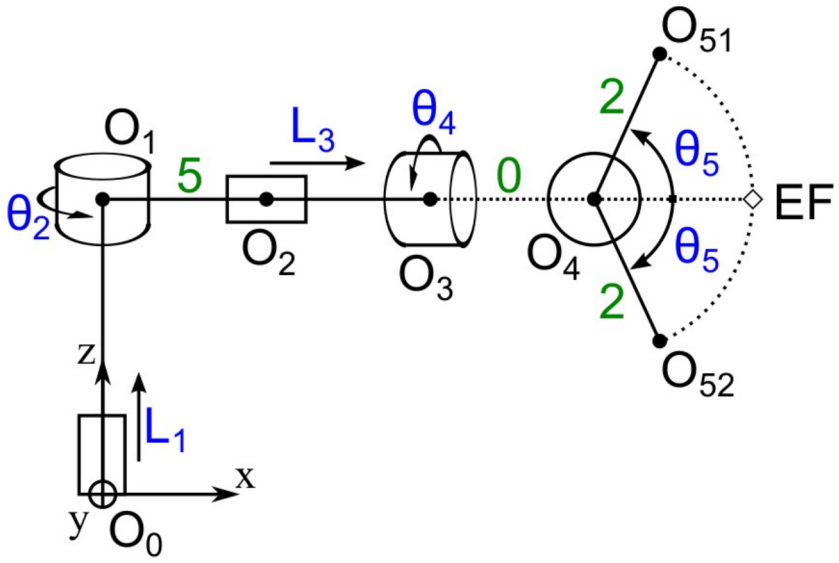

Figure 1.1: Example manipulator

To calculate the forward kinematics, we will first calculate the DH parameters:

```
# Variables de entrada
l1 = p[0]
t2 = p[1]
l3 = p[2]
t4 = p[3]
t5 = p[4]

# Parámetros D-H:
#        1    2     2'    3    4     51     52     EF
d  = [  l1,   0,    0,   l3,   0,     0,     0,    0]
th = [   0,  t2,  -90,    0,  t4, 90-t5, 90+t5,   90]
a  = [   0,   5,    0,    0,   0,     2,     2,    2]
al = [   0, -90,  -90,    0,  90,    90,    90,    0]
```

Figure 1.2: Manipulator 3 DH parameters

As can be seen in figure 1.2, before calculating the DH parameters, some input variables have been assigned, these correspond to the angles of the manipulator joints and these are the values that are introduced when the program is executed. For example, if we run it as follows:

```
$ python3 ./cinematica_directa 5 0 5 90 45
```

the forward kinematics result would be:

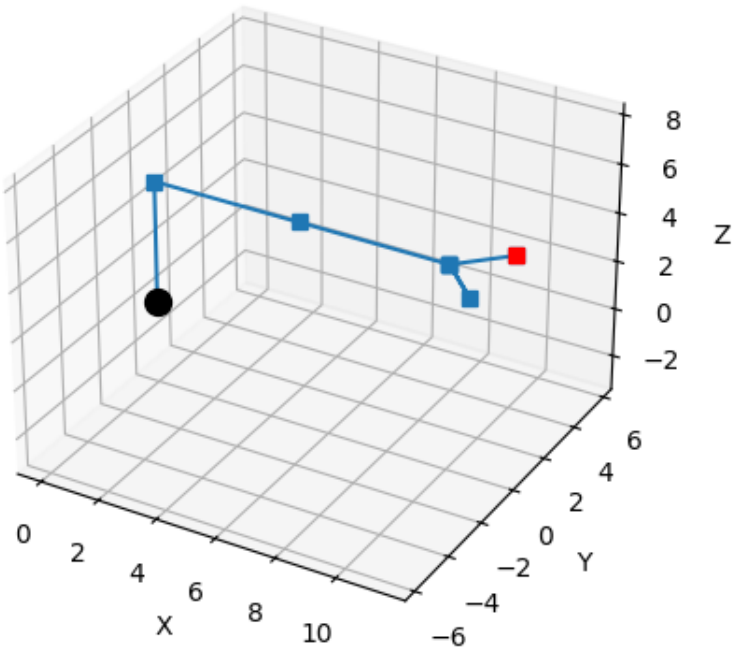

Figure 1.3: Forward kinematics of manipulator 3

Then, the transformation matrix of each joint was calculated for this manipulator:

```
# 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)

T22prima=matriz_T(d[2],th[2],a[2],al[2])
T2prima3=matriz_T(d[3],th[3],a[3],al[3])
T23=np.dot(T22prima,T2prima3)
T03=np.dot(T02,T23)

T34=matriz_T(d[4],th[4],a[4],al[4])
T04=np.dot(T03,T34)

T451=matriz_T(d[5],th[5],a[5],al[5])
T051=np.dot(T04,T451)

T452=matriz_T(d[6],th[6],a[6],al[6])
T052=np.dot(T04,T452)

T4EF=matriz_T(d[7],th[7],a[7],al[7])
T0EF=np.dot(T04,T4EF)
```

Figure 1.4: Matrix transformation of each joint of manipulator 3

Then, the transformation of each joint, specifying the data from the Denavit Hartenberg table specified in figure 1.2:

```
# 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()
```

Figure 1.5: Transformation of each manipulator joint 3

And finally, the result of the forward kinematics of manipulator 3:

```
# Mostrar resultado de la cinemática directa
muestra_origenes([o00, o11, o22, o33, o44, [[o510], [o520]], oEF0])
muestra_robot   ([o00, o10, o20, o30, o40, [[o510], [o520]], oEF0])
input()
```

Figure 1.6: Result of the forward kinematics of manipulator 3

2 Inverse kinematics

Inverse kinematics is a branch of robotics and mechanics that deals with the relationship between the movements of the links of a robot and the variables that control them. In other words, inverse kinematics is the problem of finding the set of parameters that define the positions and orientations of all the links, given the position and orientation of the end of the robot.

Inverse kinematics can be a complex problem due to the presence of physical constraints and limitations of the robot, such as joint movement limits, collisions or singularities.

In the practice of inverse kinematics, the distance that a prismatic joint at point O_i has to be extended in such a way that the end point of the robot O_n is as close as possible to the target position R is to be calculated.

The approach can only be made in the extension direction L_i, which can be calculated as an angle of w defining the rotation with respect to the absolute x-axis. The angle w can be calculated as:

$$\omega = \sum_{j=0}^{i} \theta_j$$

Using the scalar product we can project the vector O_n to R in the direction of extension of the joint to obtain the distance d:

$$\mathbf{d} = \begin{bmatrix} \cos(\omega) \\ \sin(\omega) \end{bmatrix} \cdot (R - \mathrm{ON})$$

Therefore, the value of L_i after each iteration becomes:

$$\mathbf{L_i} + \begin{bmatrix} \cos(\omega) \\ \sin(\omega) \end{bmatrix} \cdot (R - \mathrm{ON}), \quad con\ \omega = \sum_{j=0}^{i} \theta_j$$

To calculate the inverse kinematics in the script in *Python* we will do it with the Cyclic Coordinate Descent (CCD) algorithm. And before we start developing the CCD algorithm, we must first define the joint values for the direct kinematics. For this purpose, lists have been defined with the joint values of each joint of the robot and these joint values have been defined so that the robot does not exceed the limits of the joints, upper and lower limits. And then, with another list, we will differentiate between a rotational joint and a prismatic joint.

```
# Valores articulares arbitrarios para la cinemática directa inicial
th = [0.,0.,0.]                 # Ángulos de las articulaciones
a = [5.,5.,5.]                  # Longitud de las articulaciones
tipo_articulacion = [0, 0, 0] # Tipo de articulación - (0) rotacional, (1) prismática
limite_sup = [45, 90, 90]       # Límite superior de las articulaciones, por encima eje X
limite_inf = [-45, -90, -90]    # Límite inferior de las articulaciones, por debajo eje X
```

Figure 2.1: Lists of limits joints articulations

Then, already differentiating the joint values, in the main loop we compute ω, which for the rotational joint we calculate the vectors V_1 and V_2, which represent the position differences between the points of interest. For vector V_1 is the difference between the robot end point and the target and for vector V_2 is the difference between the robot end point and the point of interest. Once the vectors are calculated, we calculate the angles between the vectors V_1 and V_2 and these angles are added to the variable θ of the rotational joint making the difference between the angles of the vectors V_1 and V_2. And in the case of the prismatic joint it will be the sum of all the θ up to the current one (inclusive) and then the scalar projection of the vector that goes from the end point of the robot to the target on the extension direction of the prismatic joint is calculated.

Afterwards, it is necessary to ensure that the robot does not exceed the limits of the joints, so the current position must be checked to see if it is above or below the limits, and if so, the position will become that of the limit itself, whether it is above or below.

```
while (dist > EPSILON and abs(prev-dist) > EPSILON/100.):
  prev = dist
  O=[cin_dir(th,a)]
  num_articulaciones = len(th)
  for i in range(num_articulaciones):
    # ===== Cálculo de la cinemática inversa (CCD) =====
    j = num_articulaciones - i - 1  # Articulación actual
    if [tipo_articulacion[j] == 0 for j in range(num_articulaciones)][i]: # Articulacion rotatoria
      v1 = np.subtract(objetivo, O[i][j])
      v2 = np.subtract(O[i][-1], O[i][j])

      alpha1 = atan2(v1[1], v1[0])
      alpha2 = atan2(v2[1], v2[0])

      th[j] += alpha1 - alpha2

      while th[j] > pi: th[j] -= 2 * pi
      while th[j] < -pi: th[j] += 2 * pi

      # Aplicamos los límites a theta
      if (th[j] > np.radians(limite_sup[j])):
        th[j] = np.radians(limite_sup[j])
      elif (th[j] < np.radians(limite_inf[j])):
        th[j] = np.radians(limite_inf[j])
    elif [tipo_articulacion[j] == 1 for j in range(num_articulaciones)][i]: # Articulacion prismatica
      w = np.sum(th[:j + 1])
      d = np.dot([np.cos(w), np.sin(w)], np.subtract(objetivo, O[i][-1]))
      a[j] += d

      # Comprobamos los límites
      if (a[j] > limite_sup[j]):
        a[j] = limite_sup[j]
      elif (a[j] < limite_inf[j]):
        a[j] = limite_inf[j]
    else:
      print("Tipo de articulacion no reconocido, tiene que ser rotatoria (0) o prismatica (1)")
      exit(1)

    O.append(cin_dir(th,a))
```

Figure 2.2: Inverse kinematics code with the CCD algorithm

2.1 Example execution

An example of executing inverse kinematics script would be to move the robot to the point x = 5, y = 5:

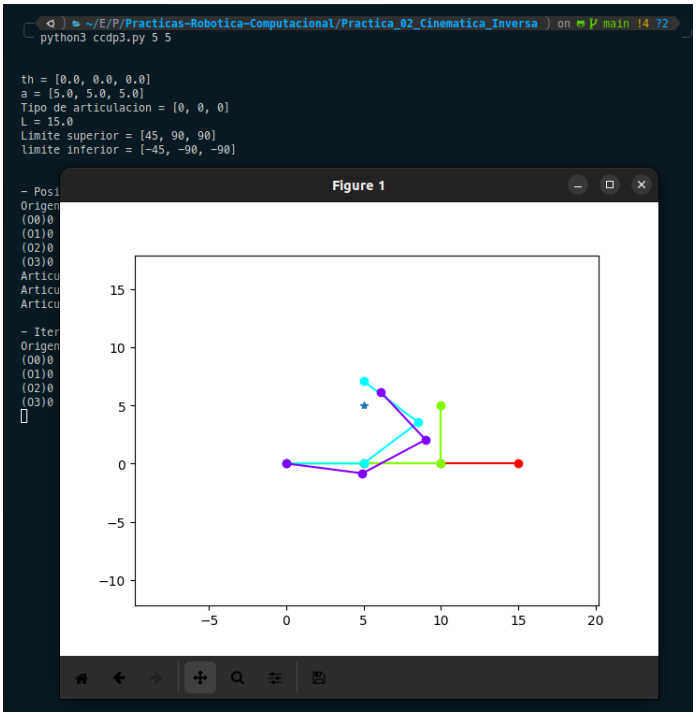

Figure 2.3: Example of the execution of inverse kinematics in the first iteration

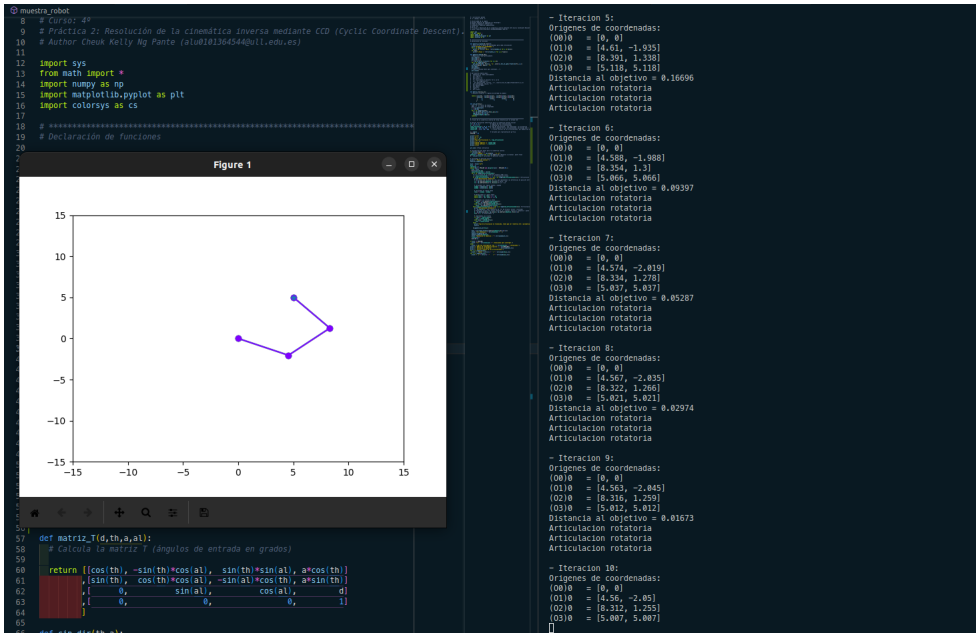

Figure 2.4: Example of inverse kinematics execution in the last iteration

3 Localization

Localization is the process of estimating the position of a robot in an unknown environment. Localization is a fundamental problem in mobile robotics, as most mobile robotics tasks require the robot to know its position in the environment. Localization is a difficult problem because the robot does not know its initial position and the environment may be unknown and dynamic. In addition, the robot's sensors may be noisy and the robot may have inaccurate movement.

In the practice of localization, the most probable position of the robot has to be calculated from its sensory system within a square region of centre "centre" and side $2 \cdot$ radius.

To calculate the location in the script in *Python* is to complete the function *localizacion* which receives as parameters the beacons, the actual location of the robot, the ideal location of the robot, the centre of the region and the radius of the region and a show parameter to display the location graph. This function aims to find the most probable location of a robot within a square region, using its sensory system and a set of reference beacons.

The localization process is performed by comparing the measurements of the real robot with the measurements of the ideal robot at different points within the square region. The ideal robot is moved to each point and the error between the measurements of the ideal robot and the measurements of the real robot is calculated. The point with the smallest error is considered the most likely location of the real robot.

The code uses a *for* loop to iterate over the points within the square region. It uses the function *np.arange* to generate a sequence of values within the range of the radius of the region. This function returns an array of values that are used to calculate the 'x' and 'y' coordinates of the current point.

Within the loop, the position of the ideal robot is updated to the current point and the error between the measurements of the ideal robot and the measurements of the real robot is calculated using the function *real.measurement_prob(ideal.sense(balizas), balizas)*. The error is stored in an image matrix.

The code also keeps track of the minimum error found so far and the point corresponding to that minimum error. If an error smaller than the current minimum error is found, the minimum error is updated and the current point is saved as the best point.

Once the loop is completed, the ideal robot is placed at the best point found, which is considered the most likely location of the actual robot. actual robot. Finally, the best point and the minimum error are printed.

Once the function *localizacion* has been developed, it has been invoked before iterating over the list of target points and also at the end of the loop to relocate the robot to calculate the best position for the ideal robot based on the position of the real robot.

3.1 Localization code

```python
def localizacion(balizas, real, ideal, centro, radio, mostrar=0):
    imagen = []
    min_error = sys.maxsize
    mejor_punto = []
    incremento = 0.05

    for i in np.arange(-radio, radio, incremento):
        imagen.append([])
        for j in np.arange(-radio, radio, incremento):
            x = centro[0] + j
            y = centro[1] + i
            ideal.set(x, y, ideal.orientation)
            error = real.measurement_prob(ideal.sense(balizas), balizas)
            imagen[-1].append(error)
            if (error < min_error):
                min_error = error
                mejor_punto = [x, y]
    ideal.set(mejor_punto[0], mejor_punto[1], real.orientation)
    print("Mejor:", mejor_punto, min_error)

    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(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()
```

Figure 3.1: Localization code

```python
localizacion(objetivos, real, ideal, ideal.pose(), 5, mostrar=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())

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

        espacio += v
        tiempo += 1
```

Figure 3.2: Target localization code

3.2 Example execution

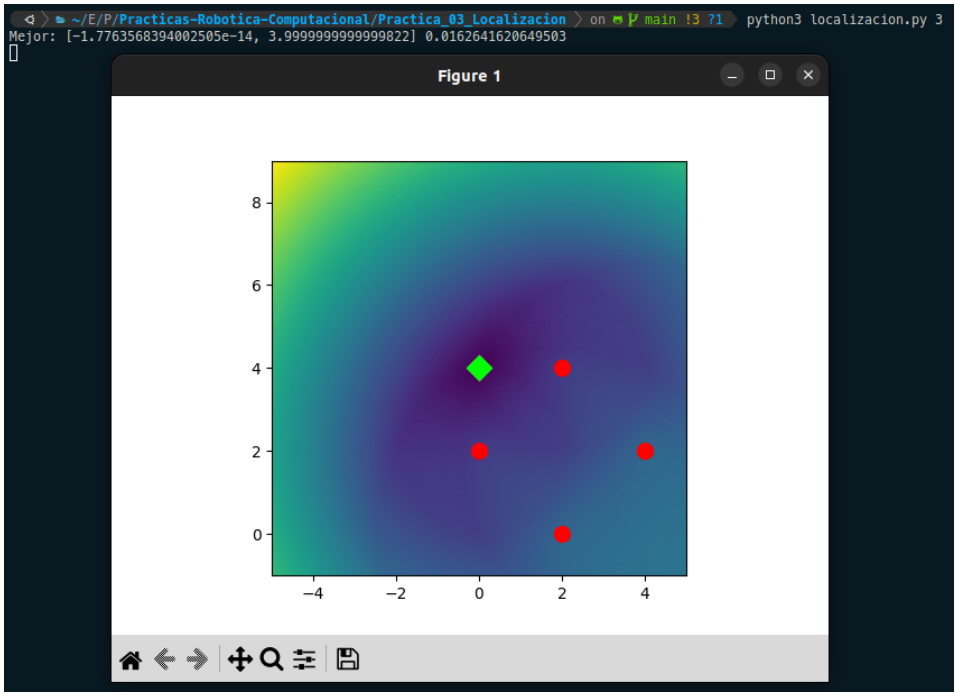

Figure 3.3: Example of localization execution

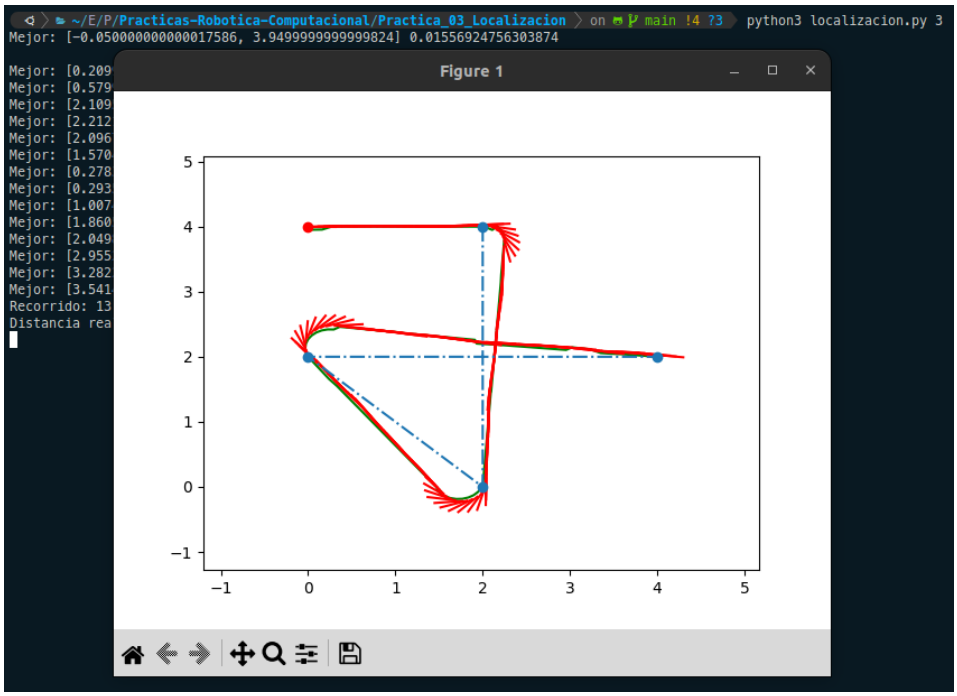

Figure 3.4: Example of localization execution