Marta Rosa Flores // Daniel Triano Moreno

ampliación de robótica

Memoria de prácticas: Manipuladores

Índice:

[Práctica 1: Planificación cartesiana - cuaternios. 2](#_Toc73264590)

[Apartado 1.1: 2](#_Toc73264591)

[Apartado 1.2: 2](#_Toc73264592)

[Apartado 1.3: 5](#_Toc73264593)

[Práctica 2: Localización basada en EKF 7](#_Toc73264594)

[Apartado 2.1 7](#_Toc73264595)

[Apartado 2.2 8](#_Toc73264596)

[Apartado 2.3 10](#_Toc73264597)

[Práctica 3: Evitar Obstáculos 12](#_Toc73264598)

[Apartado 3.1 12](#_Toc73264599)

[Apartado 3.2 14](#_Toc73264600)

[Apartado 3.3 15](#_Toc73264601)

[Práctica 4: Planificación de caminos (Dijkstra) 17](#_Toc73264602)

[Apartado 4.1 17](#_Toc73264603)

[Práctica 5: Ampliación de caminos (A\*) 25](#_Toc73264604)

[Apartado 5.1 25](#_Toc73264605)

[Apartado 5.2 28](#_Toc73264606)

[Apartado 5.3 28](#_Toc73264607)

[Práctica 6: Navegación Autónoma 30](#_Toc73264608)

[Apartado 6.1 30](#_Toc73264609)

[Apartado 6.2 32](#_Toc73264610)

[Apartado 6.3 33](#_Toc73264611)

# Práctica 1: Planificación cartesiana - cuaternios.

## Apartado 1.1:

**Definir la función de interpolación por cuaternios basada en el método de Taylor qr=qpinter(q1,q2,t) que realice el cálculo del cuaternio intermedio entre q1 (inicial) y q2 (final). El valor t debe cumplir 0≤t≤1, de modo que [q1]=qpinter(q1,q2,0) y [q2]=qpinter(q1,q2,1).**

Se procede a la implementación del método de Taylor. Para ello partimos de las localizaciones iniciales y finales de los puntos P1 y P2 tal y como se muestra en la Figura X.

Se puede alcanzar P2 partiendo de P1 y viceversa ejecutando una traslación y una rotación del sistema, trataremos de implementar dichos conceptos de forma matemática.

Se define la traslación del sistema en función del tiempo desde el P1 hacia P2, tal y como se observa en la ecuación X.

Definimos la rotación del sistema partiendo de la orientación 1 hacia la orientación 2 en base a la ecuación X:

Donde el valor de viene definido por la siguiente ecuación:

**Para calcular el eje de giro de nuestro cuaternio, se calcula q para** , tal y como se muestra en la ecuación X.

Conociendo estas ecuaciones, se procede a su implementación en MATLAB tal y como se aprecia en la Figura 1.

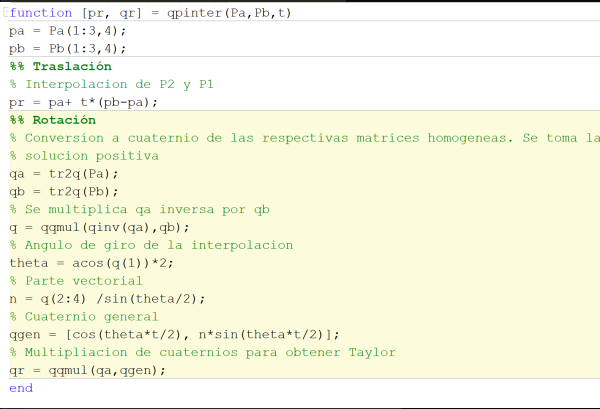


Figura : Función qpinter.

En primer lugar, se leen las orientaciones de ambos puntos, y se realiza la interpolación entre ambos puntos, utilizando la ecuación X, a continuación, se convierten a cuaternios ambas orientaciones, tomando la solución positiva por defecto. Seguidamente, se multiplican los cuaternios de cada punto y se calculan tanto el ángulo de giro como la parte vectorial.

Finalmente, se calcula el cuaternio general en función del tiempo y se multiplica por el cuaternio del punto de partida para obtener el cuaternio Taylor.

Se hace uso de la función qinv que calcula la inversa de un cuaternio tal y como se expone a continuación en la Figura 2.

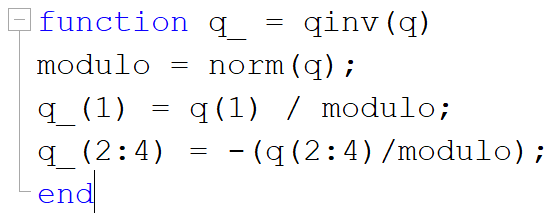
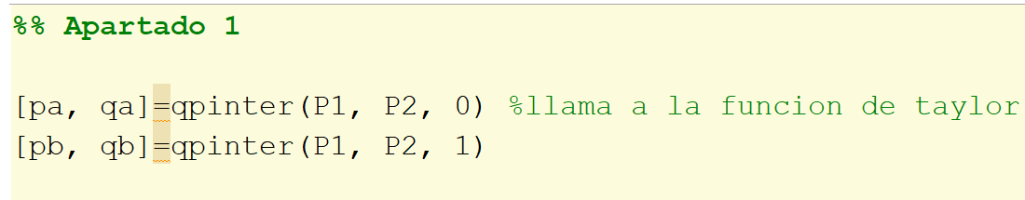


Figura - Función qinv que invierte un cuaternio

Se implementa la función qpinter en el script de la práctica y se ejecuta tal y como se observa en la Figura 3.



Figura

El resultado es el observado en la Figura 4.

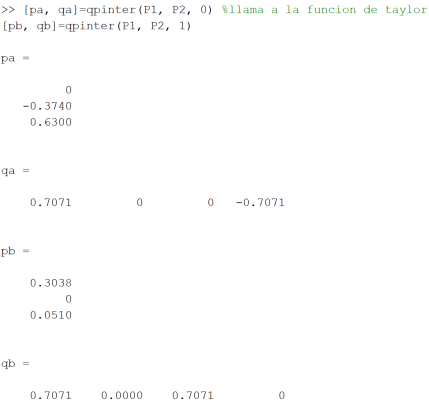


Figura : Resultado tras ejecutar la función Taylor.

## Apartado 1.2:

**Simular la navegación punto a punto del robot móvil a una velocidad de 1.2 m/s**

Se ha continuado con el código del primer apartado y se le han añadido los bloques que se detallan a continuación.

Se implementa el bloque del modelo cinemático inverso (MCI), tal y como se observa en la Figura 7.

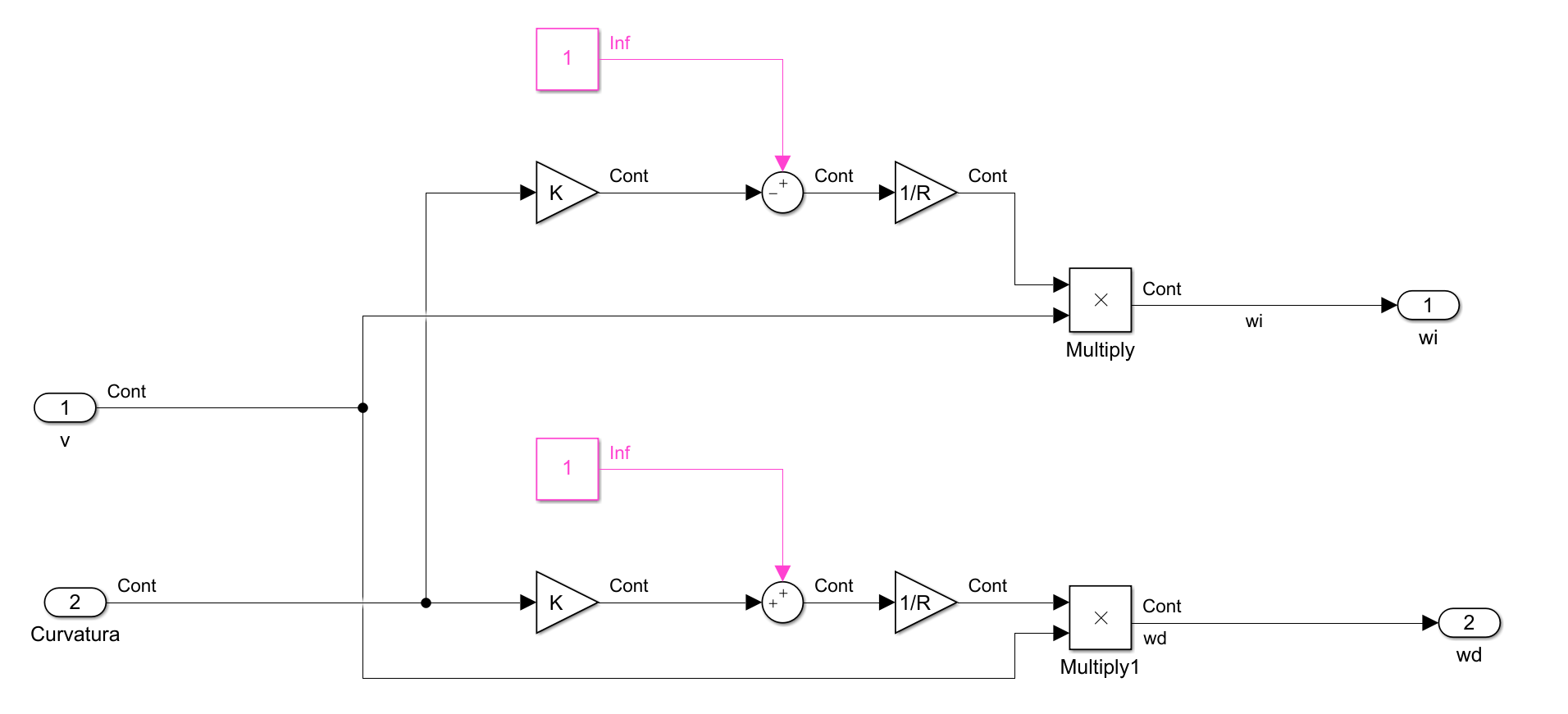


Figura : Bloque MCI.

Se ha realizado implementando las ecuaciones:

|  |  |
| --- | --- |
|  | (9) |
| |  | | --- | |  | | (10) |

Para poder simular la señal GPS se hace uso de la función de Matlab que se observa en la Figura 8, donde tiene como entradas la pose de nuestro robot y una señal de tiempo. Si las entradas son nulas, inicializamos cada una de las salidas a cero. Si han pasado más de 0.3 segundos, vamos a devolver la señal guardada proporcionada a la entrada. Estos datos se leerán únicamente cada 0.3s.

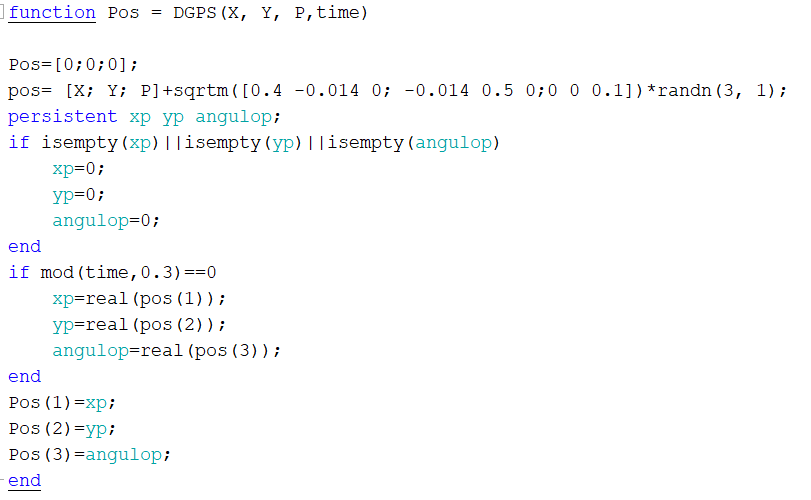


Figura : Función DGPS implementada

Una vez se consiguen leer los datos del GPS, se procede a realizar una función para seleccionar desde que punto partimos y a cuál vamos. Dado que ambos integrantes del proyecto ya trabajamos juntos en la asignatura de Laboratorio de Robótica donde ya implementamos un selector de puntos, nos hemos basado en el código que ya aprendimos para poder desarrollarlo aquí coherentemente.

El selector de puntos se puede observar en la Figura 9. Se ha implementado un LookAHead de 1 metro. Por lo tanto, si estamos a menos de un metro del punto al que queremos llegar pasaremos al siguiente, siguiendo así toda la ruta solicitada.



Figura : Selector de puntos.

Para poder cambiar a coordenadas locales se hace uso de las fórmulas (11) y (12) vistas en clase.

|  |  |
| --- | --- |
| 𝑐𝑜𝑠(ϕ0) × (x − 𝑥0) + 𝑠𝑖𝑛(ϕ0) × (𝑦 − 𝑦0) | (11) |
| |  | | --- | | 𝑦̂ = −𝑠𝑖𝑛(ϕ0) × (x − 𝑥0) + 𝑐𝑜𝑠(ϕ0) × (𝑦 − 𝑦0) | | (12) |

Se implementan dichas funciones en el bloque “locales” que se observa en la Figura 10.

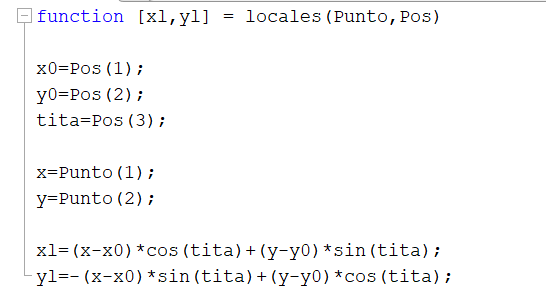


Figura : Función para conseguir las variables locales.

Para obtener la curvatura se utilizará la función atan2, calculando el ángulo correspondiente en cada momento. Tras unir todos los distintos bloques y funciones comentados se confirma la buena implementación en la Figura 11.

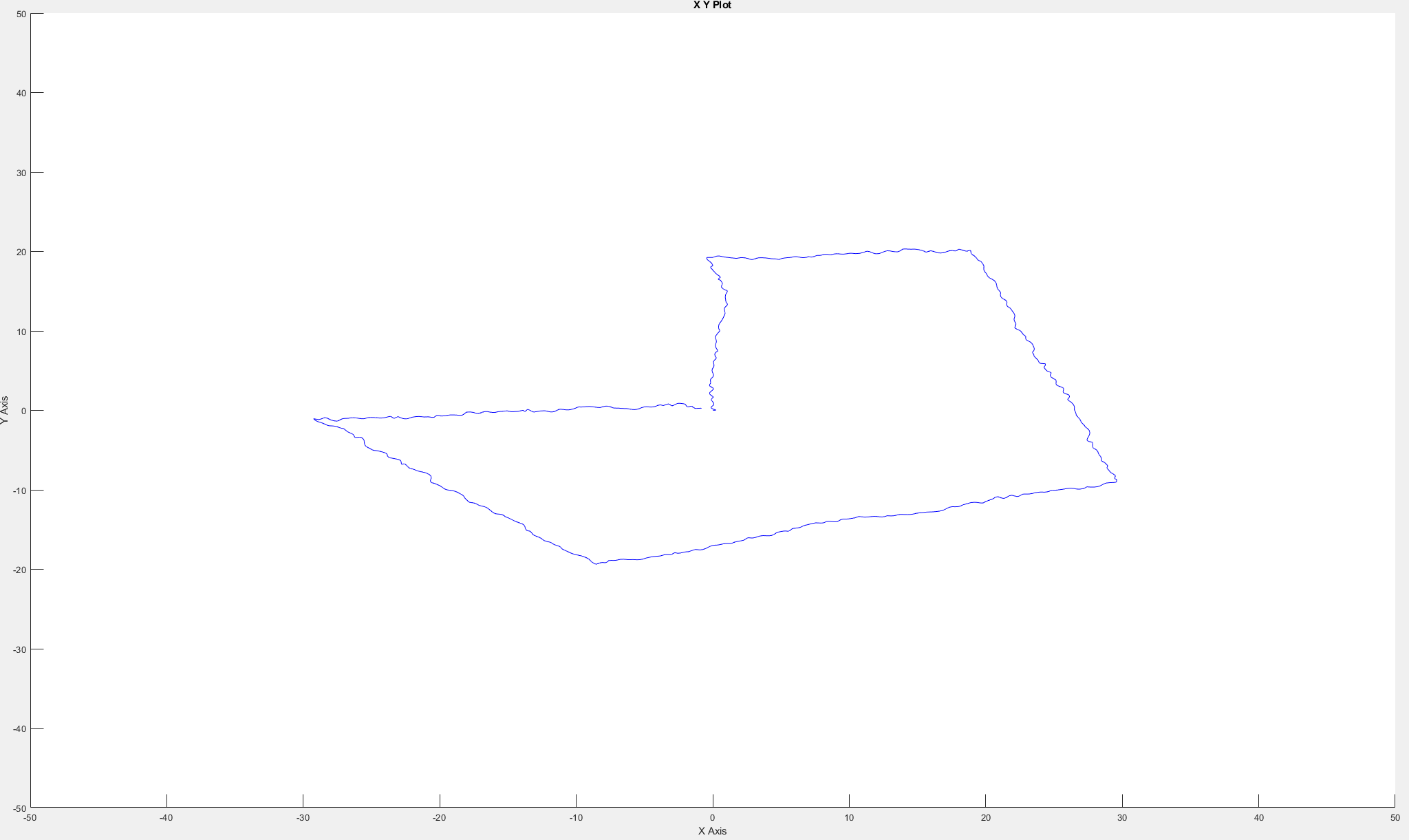


Figura : Resultado de la práctica 1b.

## Apartado 1.3:

**Empleando el método de Persecución Pura se pretende que el robot móvil recorra un pasillo largo a una velocidad de 0.3 m/s.**

El objetivo inicial para poder resolver este ejercicio es poder saber dónde se encuentra nuestro robot respecto al centro, punto medio entre ambas paredes que lo rodean. Para lograrlo, en primer lugar, se va a calcular el ángulo que debe tomar el robot para orientarse, tomando como medida el rango [0, 180]°.

Haciendo uso trigonométrico podemos llegar a la siguiente ecuación (13).

|  |  |
| --- | --- |
|  | (13) |

Haciendo uso de las explicaciones de clase, utilizaremos las coordenadas locales para poder calcular los incrementos en X e Y.

|  |  |
| --- | --- |
|  | (14) |
|  | (15) |

Queda ya pues repetir el proceso hasta llegar al punto deseado, representado en la Figura 12

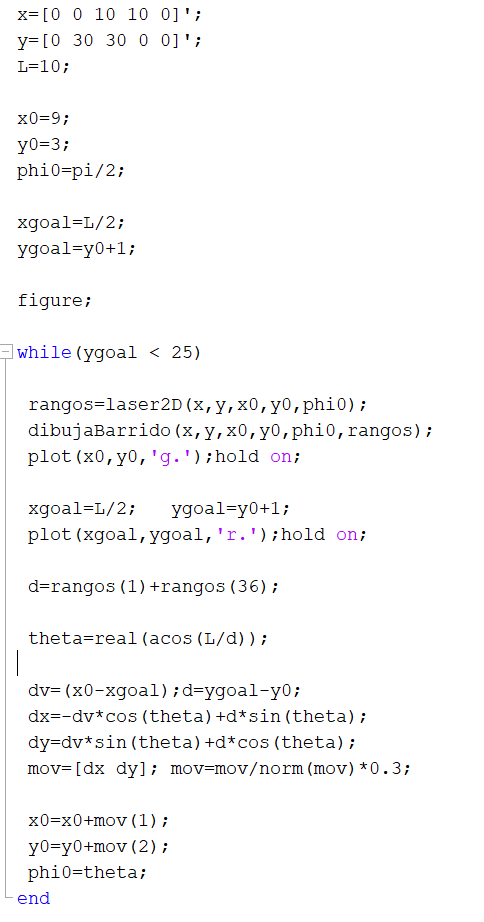


Figura : Código implementado.

El resultado obtenido se detalla en la Figura 13, donde las condiciones iniciales son el punto 3 del eje X, el punto 1 del eje Y, con un ángulo inicial de inclinación de 90°.

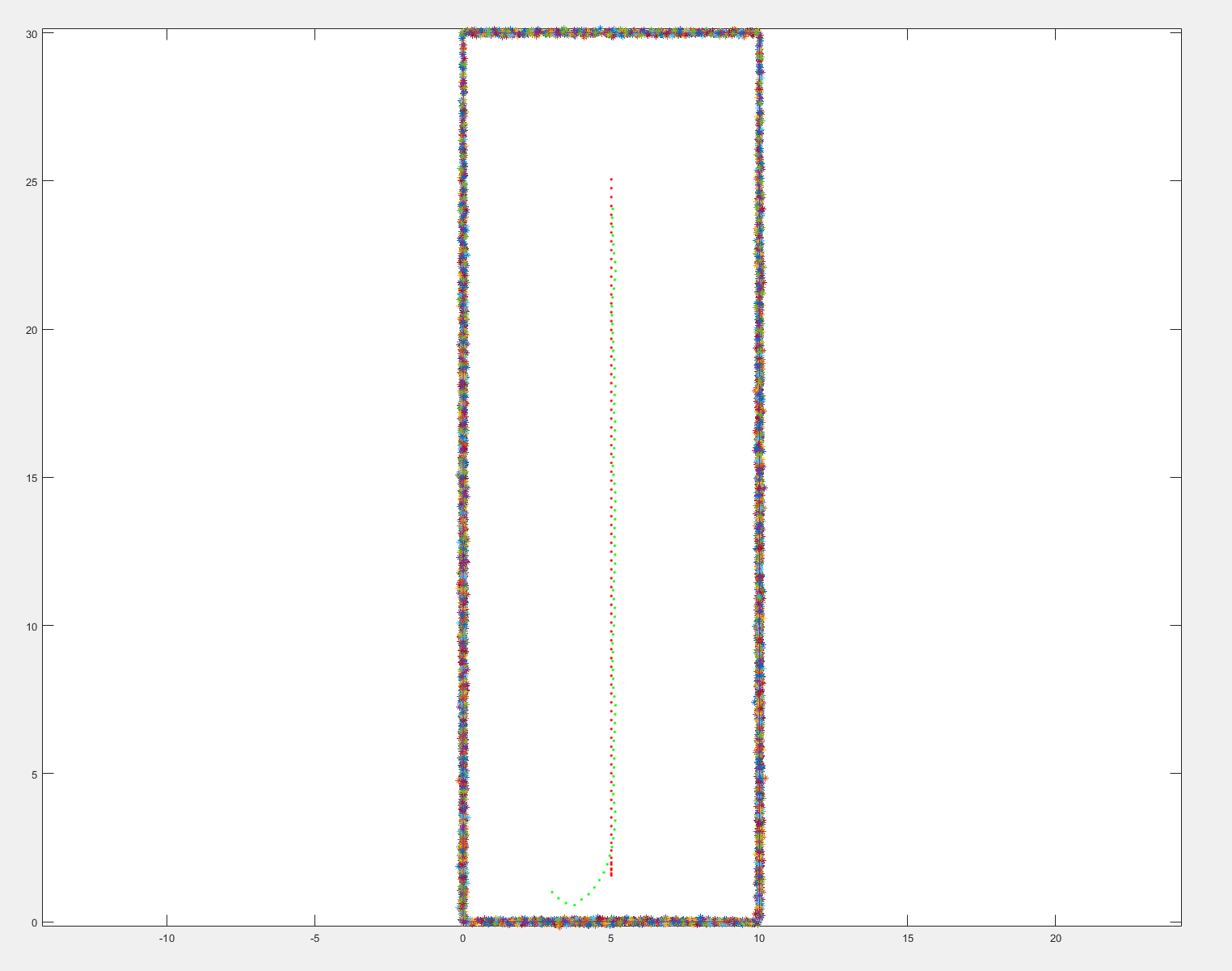


Figura : Resultado obtenido partiendo del punto [3,1] con ángulo inicial pi/2.

# Práctica 2: Localización basada en EKF

## Apartado 2.1

**Completar la fase de predicción (función predict) con las matrices jacobianas correspondientes al modelo de conducción anterior.**

En primer lugar, se carga el mapa a recorrer tal y como se comenta en el enunciado. Si se acude a la función predict se puede observar cómo están definidas las derivadas parciales en función de u[V,G], tal y como muestra la Figura 14.

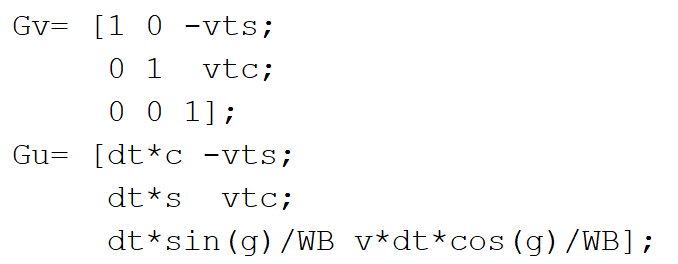


Figura : Matrices derivadas parciales

Tras ejecutar la función **ekfloc(lm,wp)** (ver Figura 15) se analizan los siguientes resultados.

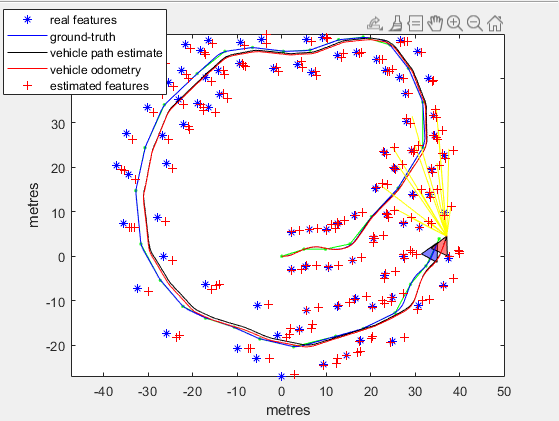


Figura : Resultado con parámetros actuales

Se observa una cierta discrepancia entre la odometría y el camino recorrido, así como las marcas de entorno previstas y las reales.

## Apartado 2.2

**Calcular el error cuadrático medio en la distancia y ángulo de orientación a lo largo de la trayectoria, según la ecuación:**

Si queremos analizar cómo afectan los distintos parámetros de ruido, ya sea de control o de observación, debemos acudir al archivo ***configfile,*** donde se modifican los parámetros correspondientes a las distintas sigmas (ver Figura 16).

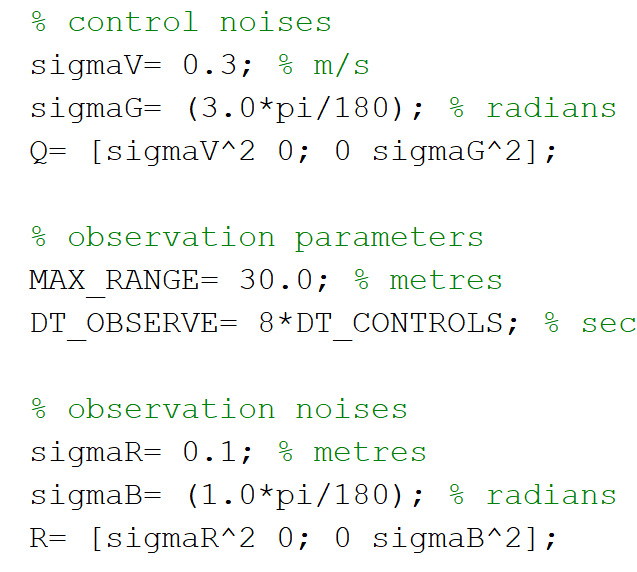


Figura : Parámetros para cada tipo de sigma.

Para ver el error medio tanto en X, en Y como en Theta, se ha implementado la función llamada ***graficaMartaDani*** (ver Figura 17).

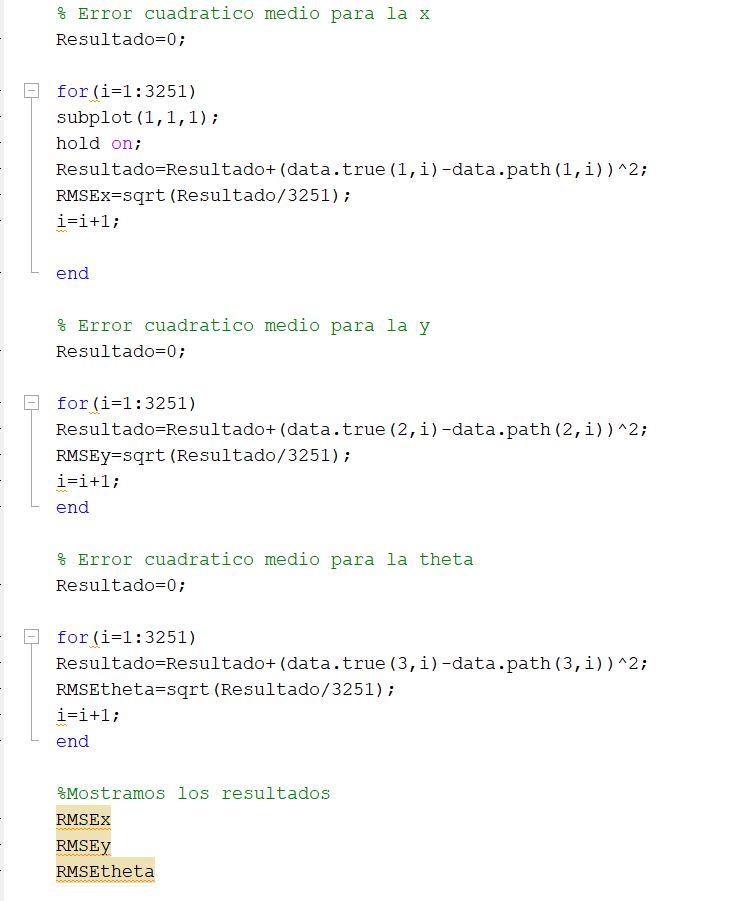


Figura : Código del archivo "GraficaMartaDani".

Donde simplemente cogemos la columna correspondiente a cada variable para poder calcular su error. Se utiliza un bucle for de 3251 iteraciones dado que ese es el tamaño de nuestra función (ver Figura 18).

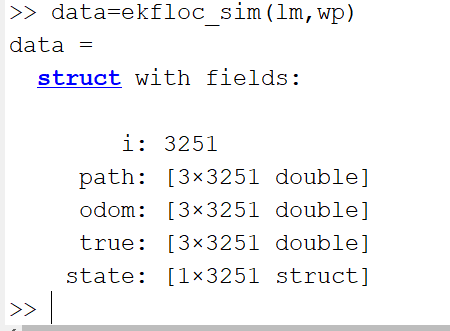


Figura : Estructura obtenida tras ejecutar ekfloc.

Así pues, se observa el error sin modificar ningún parámetro, obteniendo los siguientes resultados ver Figura 19:

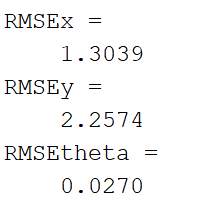


Figura : Errores obtenidos (I)

Se observa un mayor error en Y, y casi nulo en theta.

## Apartado 2.3

**Realizar simulaciones para distintos valores de incertidumbre en el control del vehículo y de la observación (al menos un cambio en cada uno de ellos). Modificar dichas incertidumbres en el fichero de parámetros (configfile). Estudiar su influencia en el resultado calculando el error cuadrático medio en cada caso y comentar las conclusiones.**

Si ahora se aumenta el valor de SigmaV por tres veces el valor anterior, se observa la Figura 20:

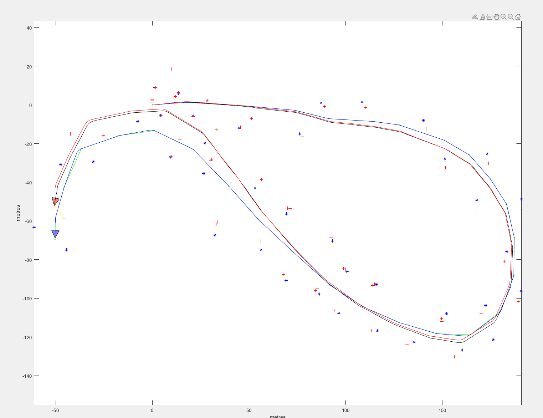


Figura : Resultados tras modificar SigmaV.

Se puede ver a simple vista y sin realizar cálculos que el error ha aumentado, sobre todo en el eje Y donde se dispersan más los puntos. Comprobamos eso al calcularlo (ver la Figura 21).

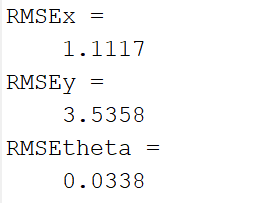


Figura : Errores obtenidos (II)

Confirmamos la previsión, el error en Y ha aumentado al aumentar el ruido de control SigmaV.

Si ahora modificamos SigmaG (ver Figura 22), vemos que no solo aumenta el error en Y si no que también aumenta en X (ver Figura 23).

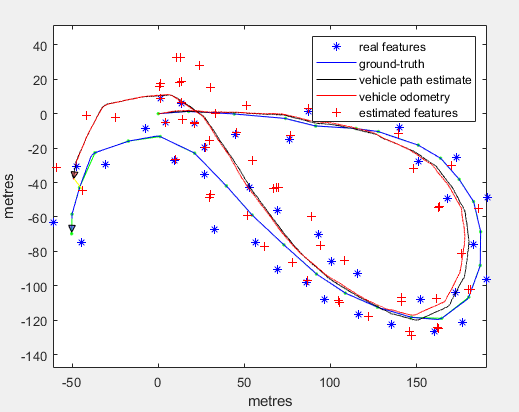
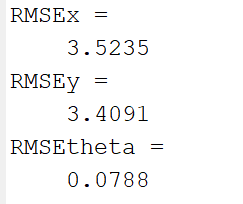
 

Figura : Resultados tras modificar SigmaG.  Figura : Errores obtenidos (III)

A su vez el error en theta duplica con respecto al anterior caso.

Modificamos a continuación los ruidos de observación, pasando de un SigmaR=0.1 a SigmaR=0.5 metros. Se observa la siguiente gráfica (ver Figura 24) donde claramente el error es menor respecto a las anteriores. Todo esto se confirma al ver el error real (Figura 14).

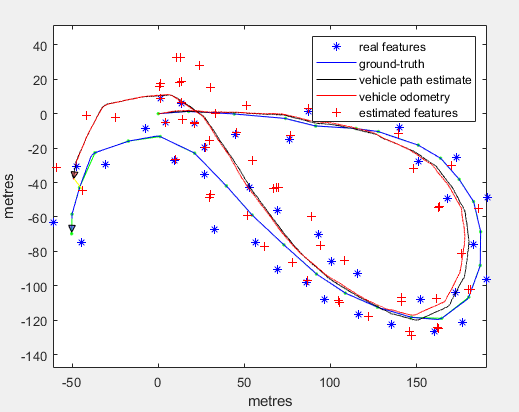
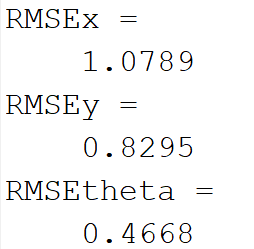
 

Figura : Resultados tras modificar SigmaR. Figura : Errores obtenidos (IV)

Al realizar la última modificación, pasando a una sigmaB tres veces mayor, se observa un aumento del error en theta bastante notable. El error en X y en Y también aumenta, pero en menor proporción. Todo esto se refleja a continuación.

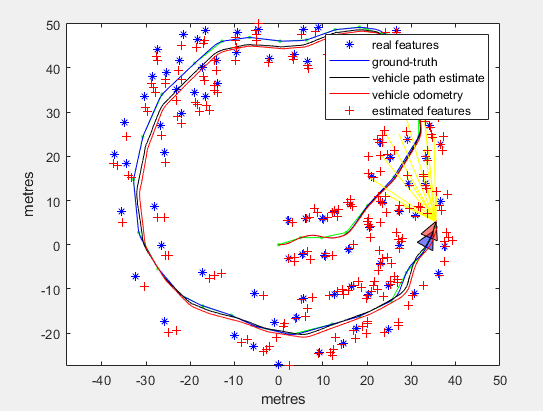
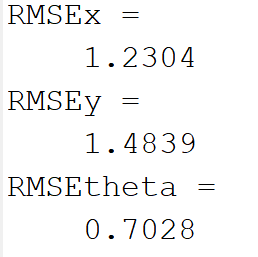
 

Figura : Resultados tras modificar SigmaB Figura : Errores obtenidos (V)

Como conclusión podemos ver que al aumentar el ruido la posición real del robot varía más que el camino que sigue la odometría. Si disminuimos el error se acercará por lo tanto al camino de la odometría, aumentando así la exactitud.

# Práctica 3: Evitar Obstáculos

## Apartado 3.1

**Completar el script denominado plantilla para conseguir la navegación reactiva del robot según el método de campos potenciales.**

Se ha hecho una función que se encarga de detectar los obstáculos del mapa, para ello hemos hecho dos bucles for de una amplitud X e Y, que serán las proporciones del mapa seleccionado. Si se detecta un píxel de valor 0, significará que se ha encontrado un píxel negro y por lo tanto un obstáculo (ver Figura 28).

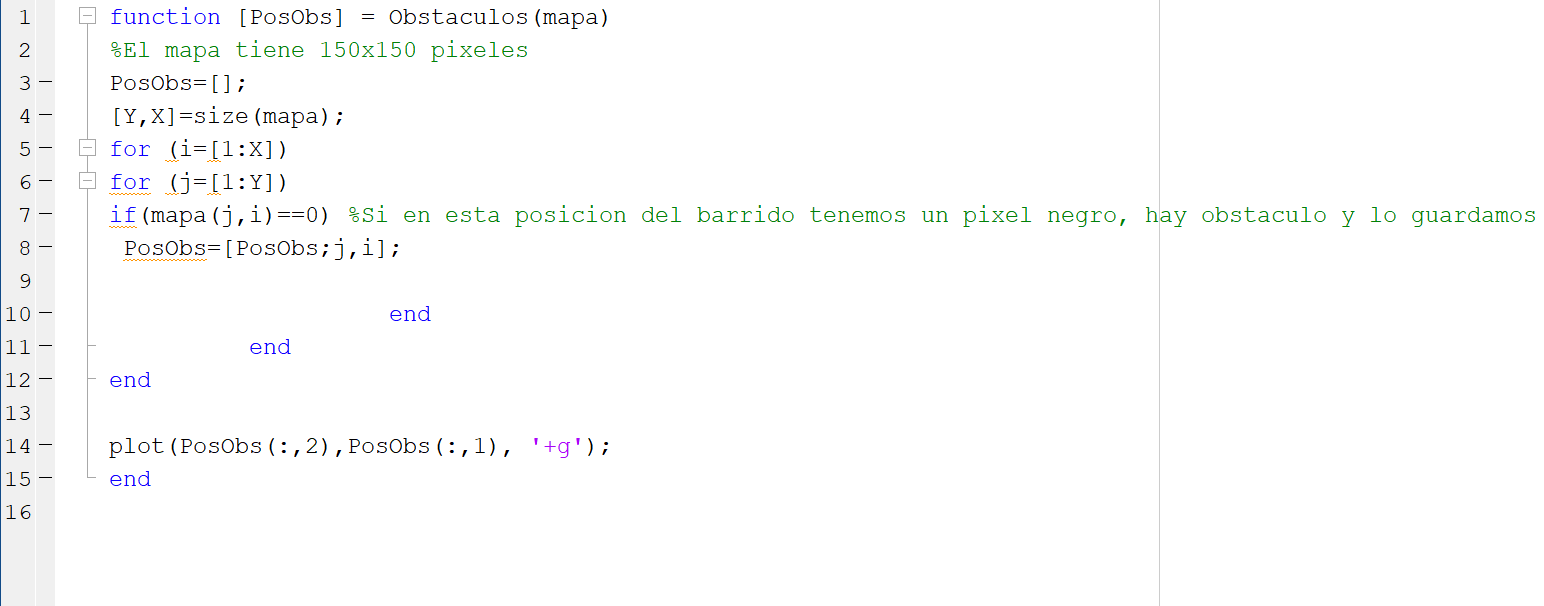


Figura : Función Obstáculos para detectar las discontinuidades en el mapa.

Mediante las siguientes ecuaciones, proporcionadas en las diapositivas de clase, se han implementado dos funciones, para calcular la fuerza de atracción y de repulsión respectivamente (ver ecuaciones 16 y 17).

|  |  |  |
| --- | --- | --- |
|  | (16) | () |
| = | (17) | () |

Dado que la fuerza de repulsión vale cero si no se cumple la condición, debemos emplear un condicionante IF, como se detalla en la Figura 29.

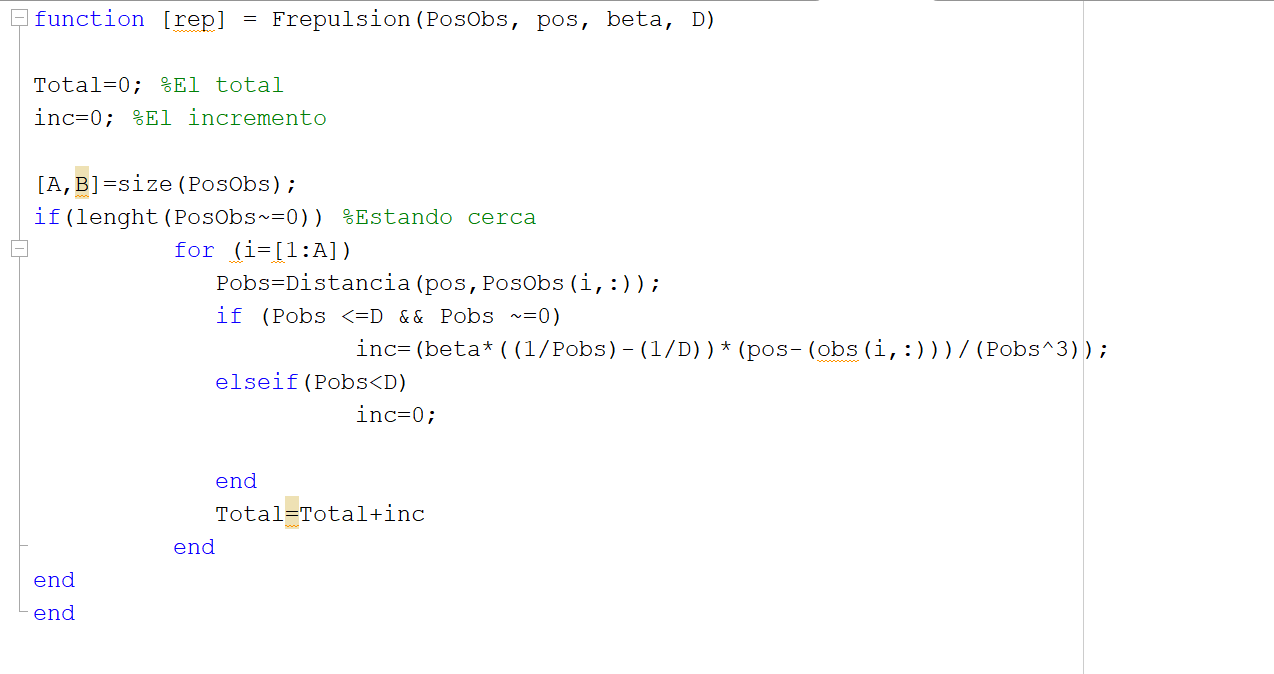


Figura : Configuración de las fuerzas de repulsión.

Se implementa una sencilla función llamada Distancia que calcula la distancia que hay entre dos puntos, como se puede observar en la Figura 30:

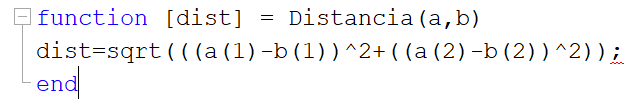


Figura : Función distancia.

Al ejecutar el programa, observamos que, gracias al cálculo de las fuerzas de repulsión, el robot es capaz de esquivar los obstáculos y alejarse de ellos, consiguiendo llegar desde el punto inicial al final sin choca, tal y como se aprecia en la Figura 31.

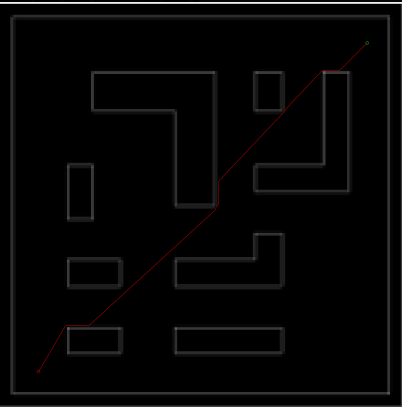


Figura : Recorrido seguido por el robot hasta llegar al punto final.

## Apartado 3.2

**Modificar adecuadamente el programa de Matlab anterior para permitir poder elegir cualquier origen y destino sobre el mapa, empleando para ello la función ginput().**

Dado que en este apartado se pide poder elegir de manera manual la posición de inicio y de final. Esto se ha realizado mediante una función que muestra en pantalla un mensaje para que el usuario elija un punto de inicio y otro de destino.

Al ejecutarse, el usuario tendrá la opción de seleccionar manualmente ambos puntos sobre el mapa, y al hacer clic se guardarán como las posiciones de inicio y destino. Esto se puede ver desarrollado en la Figura 32:

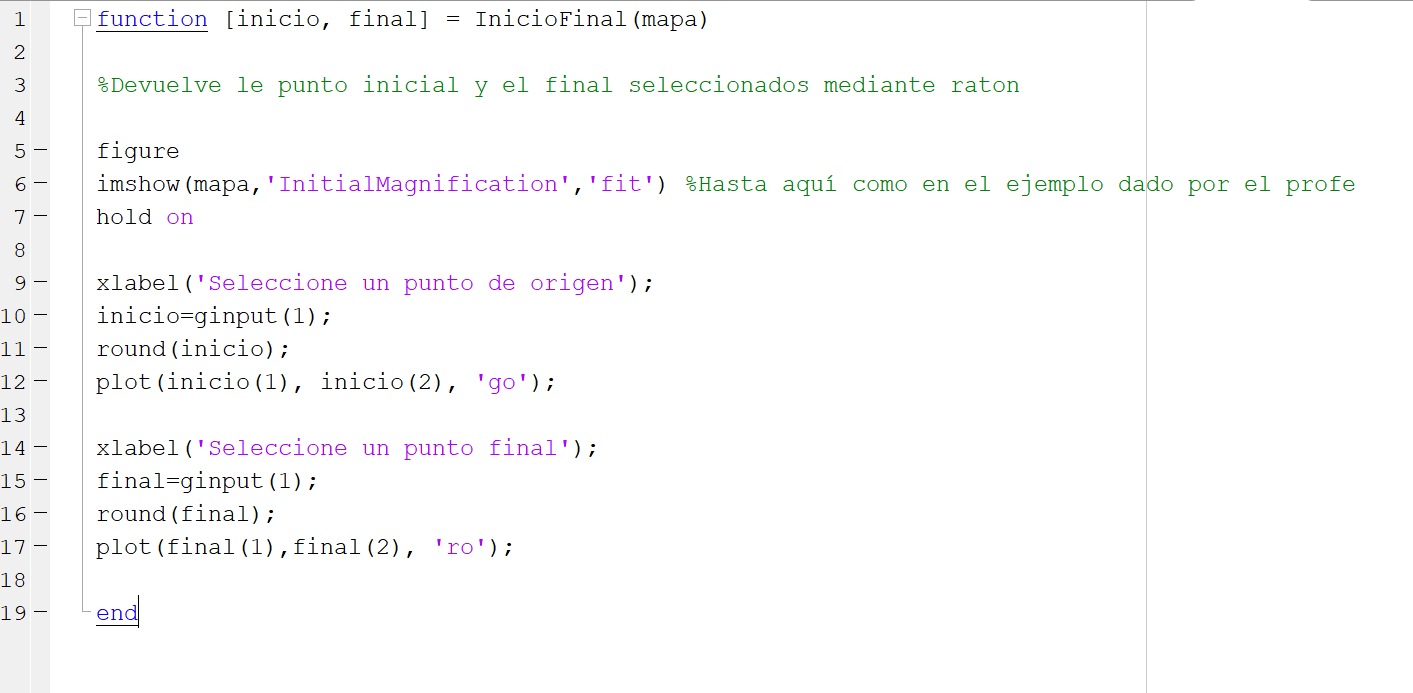


Figura : Código para introducir los puntos de origen y destino manualmente.

En la Figura 33 se observa el robot esquivando los obstáculos para alcanzar el punto de destino, tras haberlo elegido de forma manual.

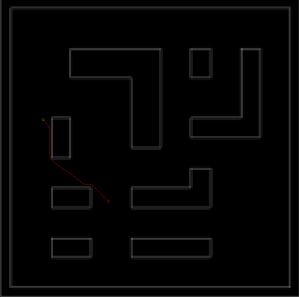


Figura : El robot realizando una ruta para llegar al destino indicado.

## Apartado 3.3

**Realizar varios experimentos con diferentes orígenes y destinos. Cambiar los parámetros del método, como α y β, o cualquier otro, para tratar de mejorar la situación. Poner el destino y origen en una situación de mínimo local (situación de trampa local), ¿es posible que el método pueda encontrar la solución sin modificarlo y cambiando solo los parámetros anteriormente mencionados?**

Este método de navegación presenta un inconveniente, y este es que alcanza situaciones en las que se encuentra un mínimo local que provoca que las fuerzas de repulsión bloqueen el movimiento en todas las direcciones.

Otro de los casos en los que se encuentra un mínimo local es en situaciones como la dada en la Figura 34, en la que el robot no encuentra la forma de llegar al destino rodeando el obstáculo sin alejarse del punto de destino, por lo que se detiene.

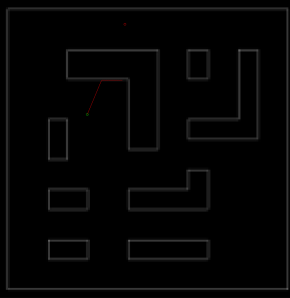


Figura : Se observa un mínimo local del que el robot no logra salir.

Al probar a modificar los parámetros, nos damos cuenta de que, al aumentar el valor de beta, aunque el robot debería acercarse menos a los obstáculos, no hay un cambio sustancial en el resultado obtenido.

Al cambiar el valor de alfa, igual que en el caso anterior, no hay un gran cambio, aunque se suaviza la trayectoria, alejándose ligeramente más de los obstáculos.

Estos dos parámetros por sí mismos no consiguen solucionar el problema de los mínimos locales.

# Práctica 4: Planificación de caminos (Dijkstra)

## Apartado 4.1

**Implementar el algoritmo de Dijkstra como una función de Matlab que devuelva el coste y la ruta óptima a partir de un origen y un destino pasados como parámetros, además del mapa topológico o grafo, que se le pasará a la función como una matriz NxN, que almacena el coste de llegar del nodo n1, como fila, al nodo n2, como columna. La función se debe implementar de forma que la llamada Dijkstra devuelva el coste de llegar desde el nodo origen al nodo destino, y un vector con la lista de nodos que componen la ruta (incluidos los nodos inicial y final).**

El algoritmo de Dijkstra es conocido como algoritmo de caminos mínimo porque consiste en encontrar el camino más optimo posible teniendo en cuenta los costes de desplazamiento.

Así pues, se van a realizar una serie de funciones que, trabajando de manera coordinada, consigan hallar tanto el coste total del desplazamiento, así como la ruta a seguir.

El mapa topológico seguido, el cual se detalla en la presentación es la siguiente (ver Figura 35).

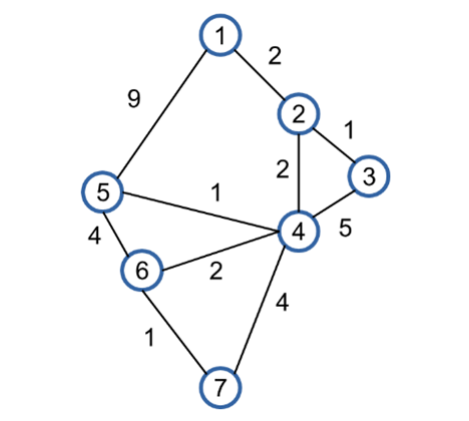


Figura : Mapa topológico de costes y nudos.

Si se representa dicho mapa en forma matricial, se consigue la siguiente matriz (ver Figura 36):

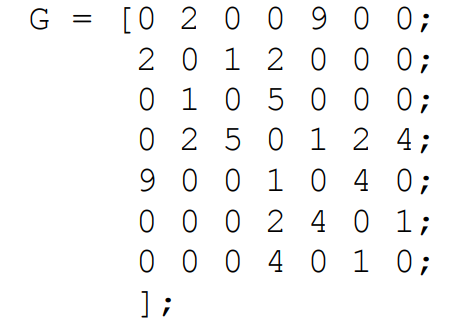


Figura : Matriz topológica de costes.

Mediante la manipulación de dicha matriz, es como se conseguirá el camino mínimo deseado.

Se implementa la función global *Dijkstra* (ver Figura 37) que a su vez está compuesta por otras subfunciones.



Figura : Función Dijkstra implementada.

Antes de proceder a la explicación de dicha función y dado que utiliza otras subfunciones, se va a detallar en primer lugar el significado de cada subfunción y después el código general anteriormente mostrado.

La función Default (ver Figura 38), tiene como entrada la matriz topológica de costes. En primer lugar, lee y almacena el tamaño de la matriz topológica de costes y se comprueba que la matriz es simétrica, es decir, tiene el mismo número de columnas que de filas. En caso contrario significaría que la matriz topológica de costes no proporciona los costes asociados a todos los caminos posibles.

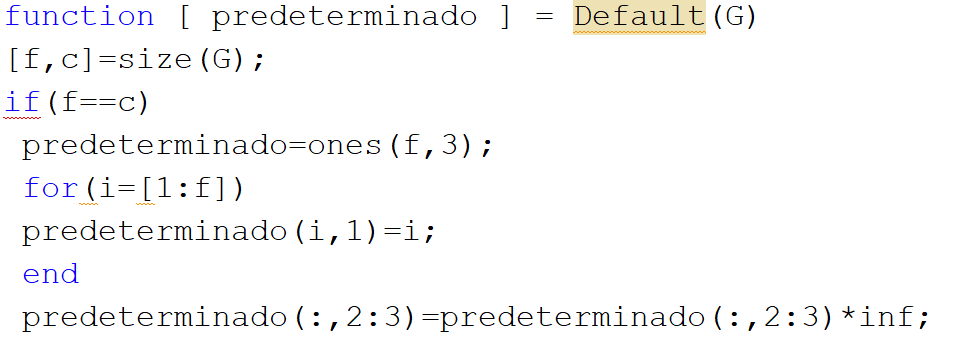


Figura : Función Default.

En función del tamaño de la matriz simétrica se crea una matriz con tantas filas como nodos hay. Además, dicha matriz tiene tres columnas. La primera de ellas está enumerada del 1 hasta f, dado que f es el tamaño de la matriz que estamos analizando, pues recordemos que es simétrica. Dicha primera columna representará pues todos los nodos presentes en nuestro sistema. La segunda y la tercera, en la primera iteración se inicializarán en infinito, dado que no se sabe en dicha primera iteración si por ejemplo del punto uno se puede ir al punto 4 y su coste asociado. La tercera representa el punto al que se desplaza y la segunda el coste referido a desplazarse desde el punto marcado por la primera columna, hacia el nodo marcado por el de la tercera.

La función *ReorganizarMatriz* (ver Figura 39) tiene como entradas la matriz de infinitos previamente creada, el punto de origen dado al inicio, el de partida, la posición actual en la que se encuentra, el coste asociado a desplazarse al nodo siguiente y el nodo siguiente al que se quiere desplazar.

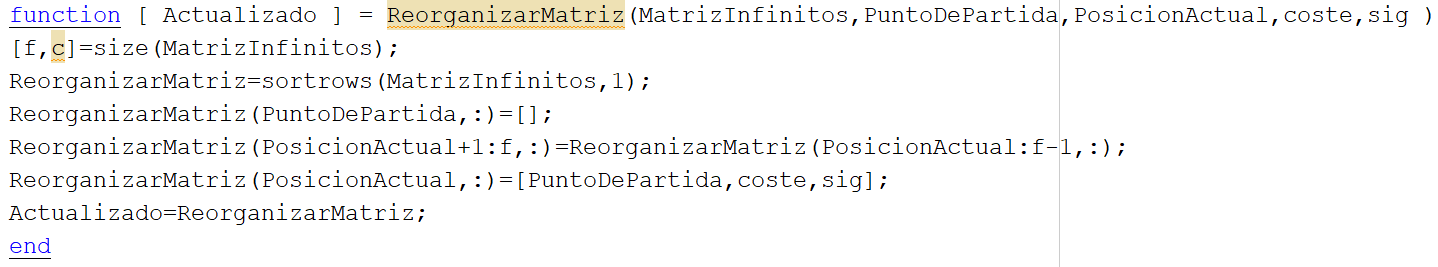


Figura : Función ReorganizarMatriz.

Se calcula nuevamente en esta función el tamaño de la matriz de infinitos, y se ordena toda la matriz en función de la primera columna. Se elimina el punto de partida de dicha matriz, pues no tiene lógica calcular el coste de desplazarte hacia donde ya estás.

Se reordena la matriz pues no nos interesa tener una fila solo de ceros, por lo tanto, si hemos partido, por ejemplo, del nodo cuatro eliminaremos la fila cuatro y la fila cinco pasará a estar en la fila cuatro mediante transformaciones elementales de matrices y así sucesivamente.

La función *Alrededores* (ver Figura 40) tiene como entradas la matriz topológica de costes, la primera posición de la matriz de infinitos donde se está trabajando y la propia matriz de infinitos. Como se ha detallado anteriormente, la segunda columna hace referencia al coste de desplazamientos y será esta la variable entorno a la cual se reorganizará todo. Mediante un bucle for se irán añadiendo los puntos a los que se puede desplazar partiendo del origen y posteriormente se reorganizará la matriz en función de los costes recién calculados.

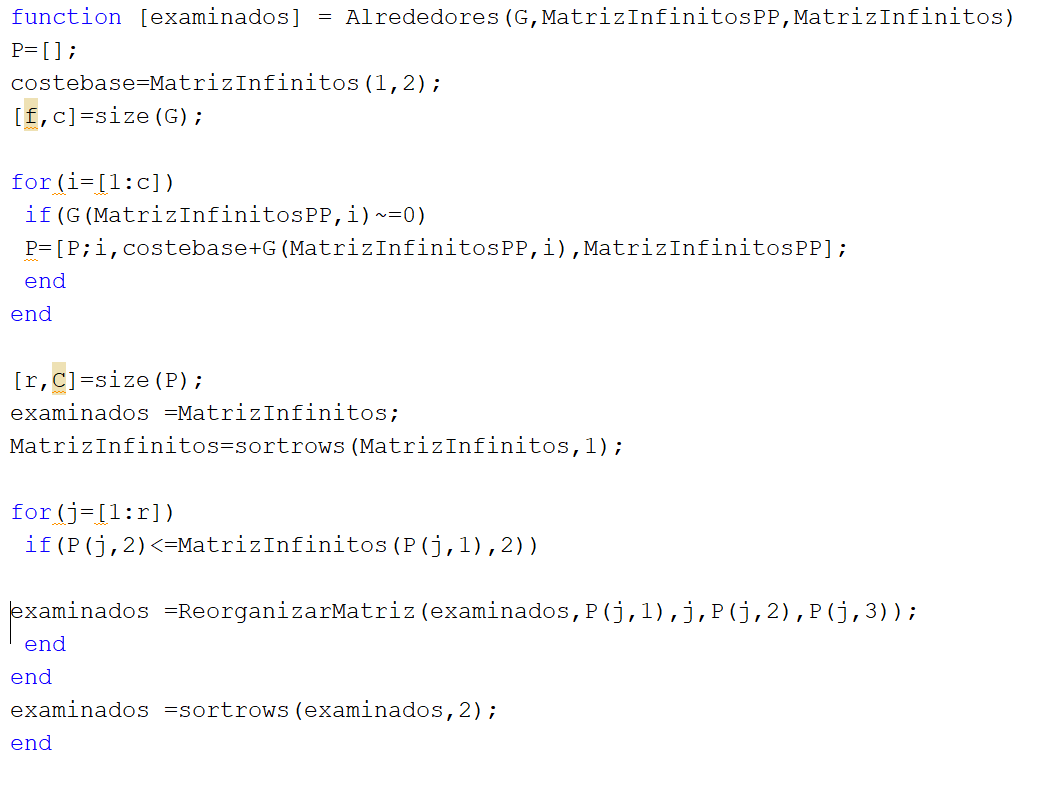


Figura : Función Alrededores.

La función *EliminarCamino* (ver Figura 41) tiene como entradas la matriz de infinitos y la ruta almacenada hasta el momento. El procedimiento a seguir es muy sencillo vamos a eliminar de la ruta seguida los puntos por los que ya ha pasado poniéndolos a valor infinito.

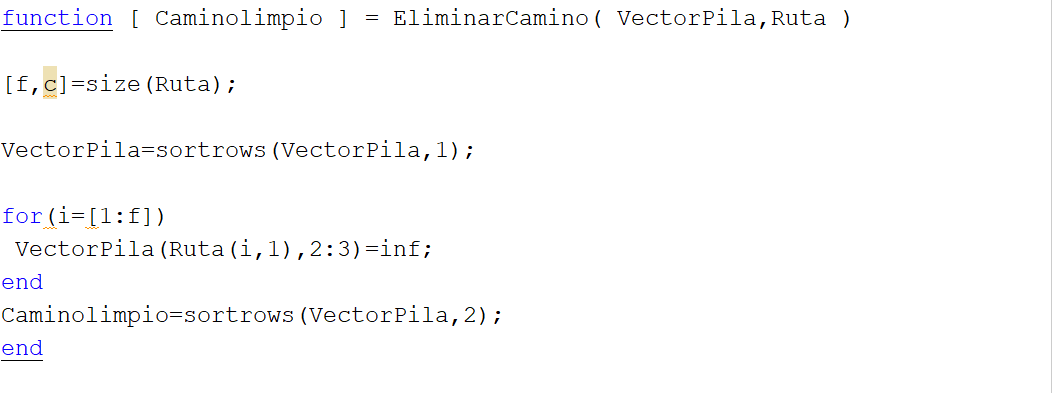


Figura : Función EliminarCamino.

Por último, la función OptimizarRuta (ver Figura 42) tiene como entradas la ruta seguida hasta el momento, el punto de origen y el destino, de forma que devolverá el menor coste entre dos puntos. Se hará un chequeo entre el punto actual y el siguiente de manera secuencial.

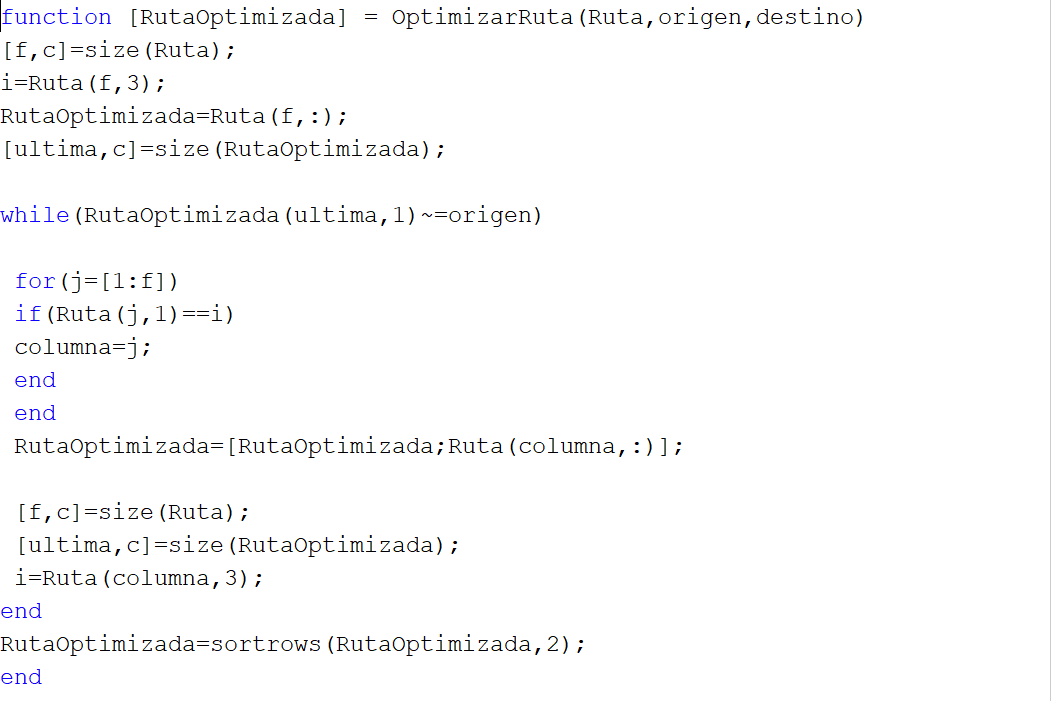


Figura : Función optimizar ruta.

Así pues, tal y como se muestra en la Figura 37, se inicializarán las variables ruta y camino. Antes de entrar a realizar cualquier operación se analizará previamente si hay algún nodo incompleto o inalcanzable. Mientras no estemos en un punto aislado y no hayamos llegado al destino, se irá completando los distintos posibles caminos haciendo uso de las funciones nombradas anteriormente. Una vez se tengan todos los posibles caminos, se optimizará la ruta y se obtendrá tanto el coste total como la ruta empleada.

Se confirma la robustez del sistema mediante una serie de comprobaciones donde se obtienen los resultados esperados (ver Figura 43).

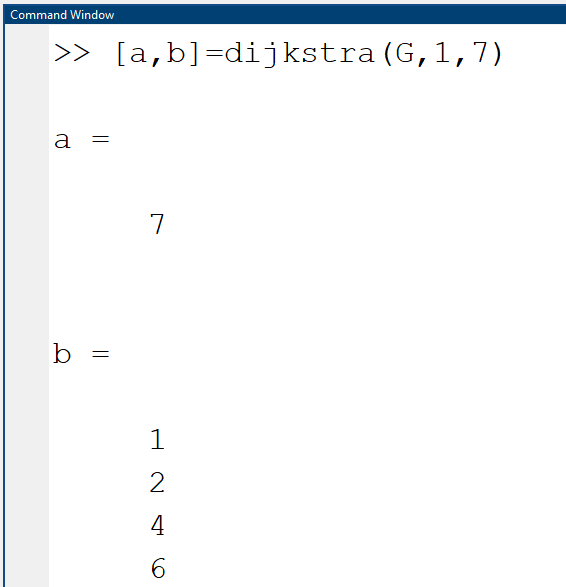


Figura : Resultados obtenidos partiendo del nodo 1 hacia el 7.

Se calculan los resultados (ver Figura 44, Figura 45 y Figura 46) para los siguientes nodos de partida y llegada:

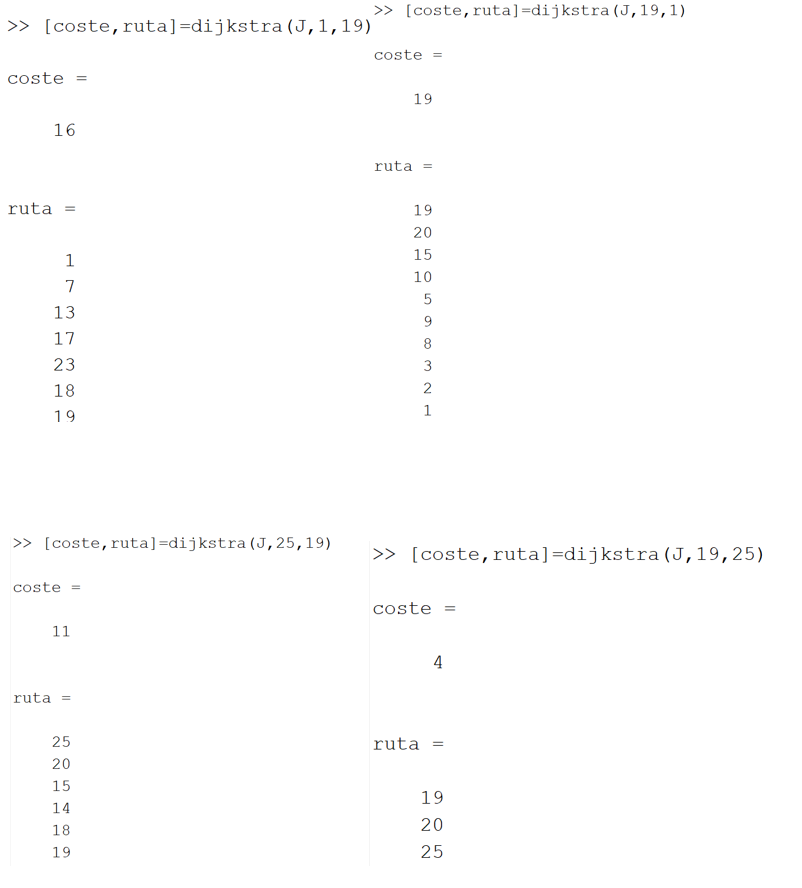


Figura : Resultados obtenidos (1).

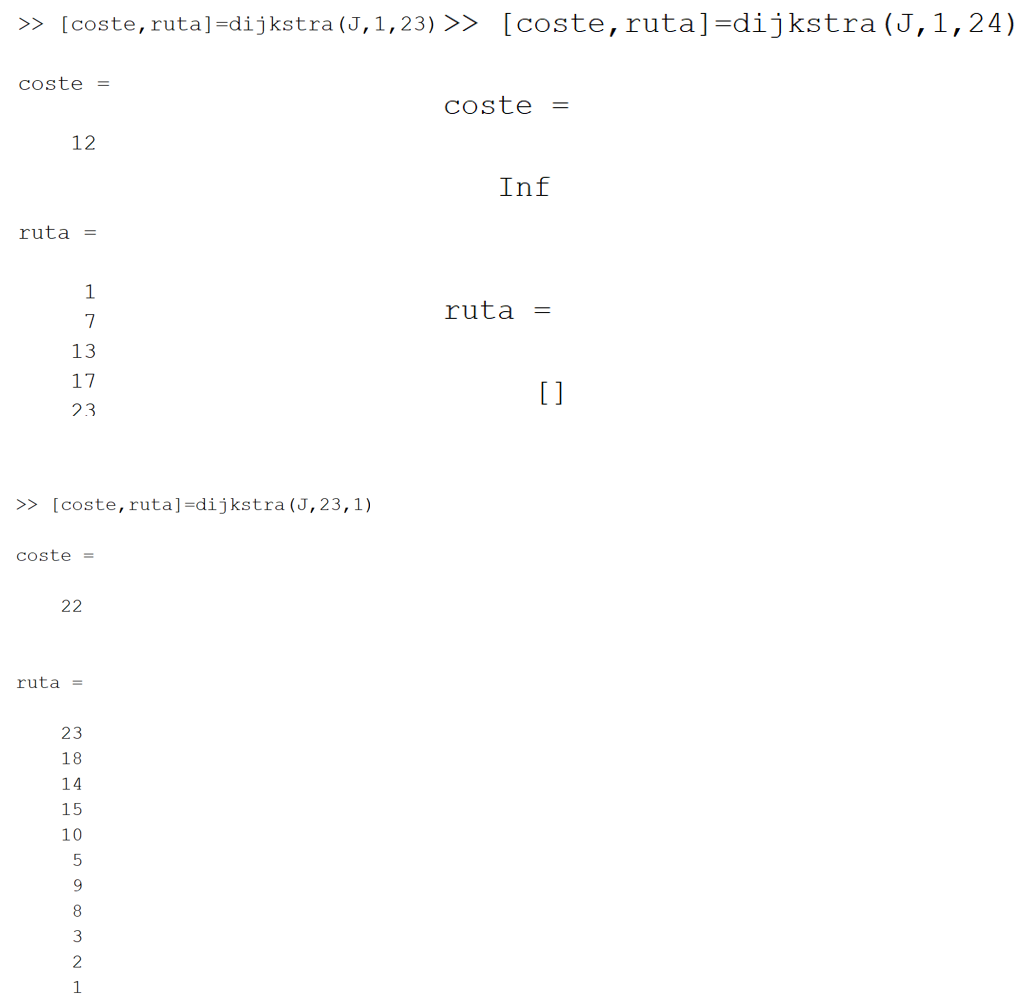


Figura : Resultados obtenidos (2).

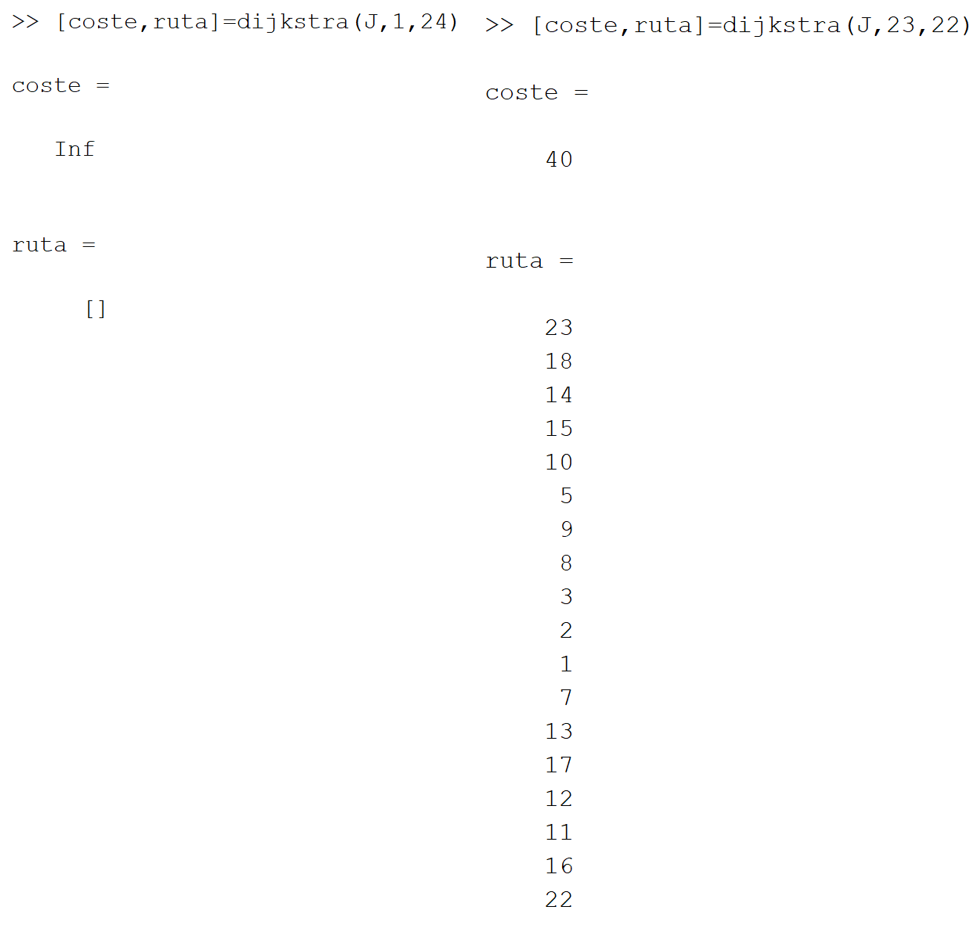


Figura : Resultados obtenidos (3).

Tal y como se puede observar en la Figura 46, dado que el nodo 24 no es alcanzable desde ningún otro nodo, el coste es infinito y no hay ruta mediante la cual se puede alcanzar dicho nodo. En cualquier otro caso, el sistema actúa correctamente según lo esperado.

# Práctica 5: Ampliación de caminos (A\*)

## Apartado 5.1

**Implementar el algoritmo A\* en Matlab mediante una función que devuelva el coste y la ruta óptima a partir de un origen y un destino**

A diferencia de la Práctica 4, este algoritmo de Dijkstra\* se basa en la optimización del cálculo computacional resultando en un algoritmo más eficiente.

Se va a utilizar la misma topología con la cual se ha trabajado anteriormente (ver Figura 35).

Para entender la explicación del código se recomienda ver: Figura 47, Figura 48 y Figura 49 paralelamente al texto que se expone a continuación.

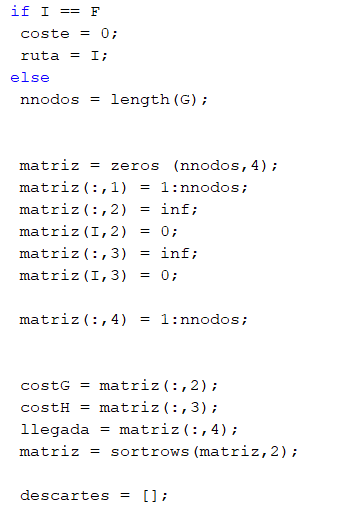


Figura : Análisis iterativo del código

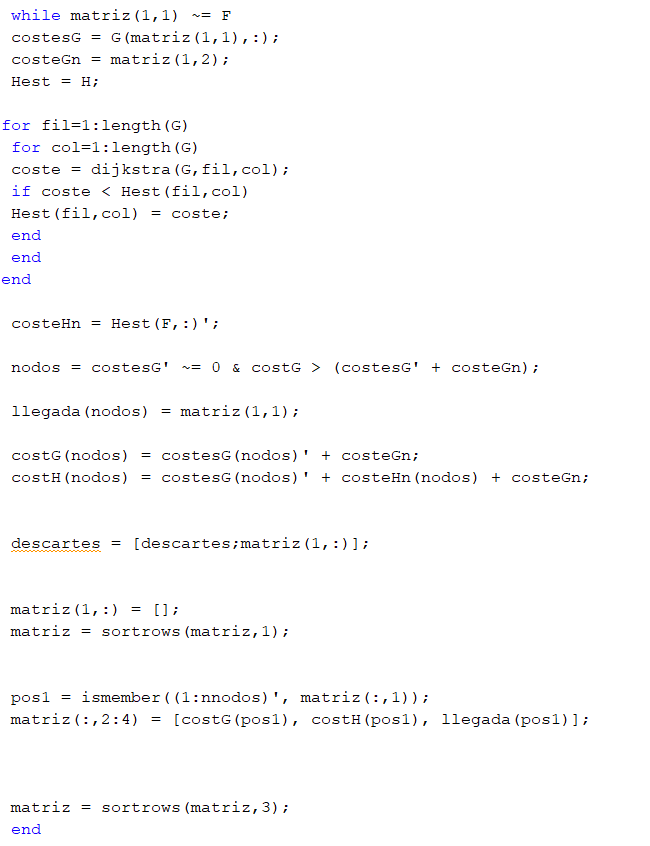


Figura : Inicialización del código. Incluye la posterior mejora.

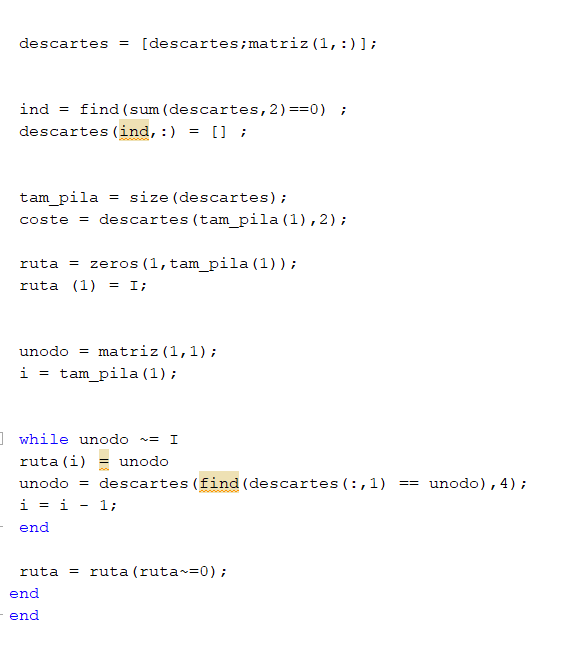


Figura : Parte final del código, se ordenan los resultados.

En primer lugar, se comprueba que el punto de partida y el destino no sean iguales. En dicho caso, el coste de llegar a sí mismo es nulo y la ruta será el punto propio.

Leemos el número de nodos que deben coincidir con el tamaño de la matriz topológica de costes, pues recordemos que estamos trabajando con una matriz simétrica.

Se procede a la implementación de una matriz donde cada columna tiene un propósito distinto. Se tendrá pues, una columna representando la posición del nodo, otra representando los costes de la matriz G, otra igual con H y en último lugar una representación de los nodos por los cuales se ha pasado ya.

Tal y como se ha realizado en la Practica 4, en la primera columna se van a introducir los n nodos al igual que en la última. En la columna dos y tres, dado que representan costes se inicializan en infinito como hemos procedido anteriormente.

Se ordenarán en función de sus respectivos costes sabiendo que el coste inicial de partida, tanto para G como para H son nulos. Se ordenarán ambos costes para que el origen igualado a cero se sitúe en la primera fila.

Se genera un vector auxiliar vacío. Mientras no estemos en el punto de llegada, se van a almacenar los costes de todos los posibles puntos de llegada respecto al punto actual. Se almacena el coste real actual del punto. Así mismo, se almacena el coste heurístico de llegada al punto H.

Si cumple la condición de costes, el nodo actual pasará a la cuarta columna representando al nodo anterior, actualizando así la matriz.

Una vez actualizada la matriz en función del nodo, se actualiza también los costes tanto de G como de H en el nuevo punto, gracias a que cumplirá siempre la condición dado que se han inicializado a infinito, tal y como vimos en clase. Puesto que la primera iteración ya está realizada, se almacenará la trayectoria en el vector auxiliar y se eliminará de la matriz de trabajo.

Se analizará por lo tanto los nuevos costes del nuevo punto entrando así en un procedimiento cíclico.

Se trabajará a continuación con los datos recopilados. En primer lugar, se desecharán aquellos puntos donde el coste total sea cero y se actualiza por lo tanto el nuevo tamaño de la matriz auxiliar. Se inserta en la posición uno el propio nodo de inicio conocido por ser un valor de entrada. Se hace lo mismo con el nodo destino. Por último, se termina de montar el resultado de la ruta desechando todos los ceros que nos encontramos en el camino.

Así pues, los resultados se verifican a continuación:

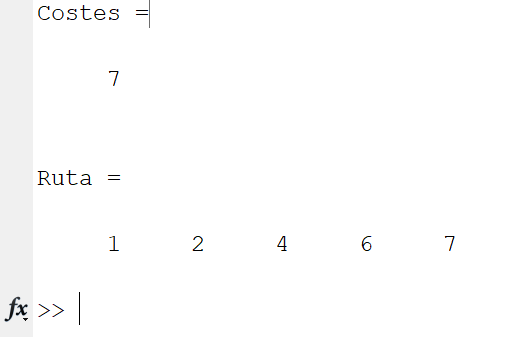


Figura : Resultados de Dijkstra\* con nudo de entrada 1 y llegada 7.

Se observa el resultado esperado, confirmando la correcta realización del ejercicio.

## Apartado 5.2

**Comprobar el resultado del algoritmo para los siguientes nodos inicial y final:**



Si se ejecuta la función Dijkstra implementada en el primer apartado se puede apreciar como el resultado no es el óptimo (ver Figura 51).

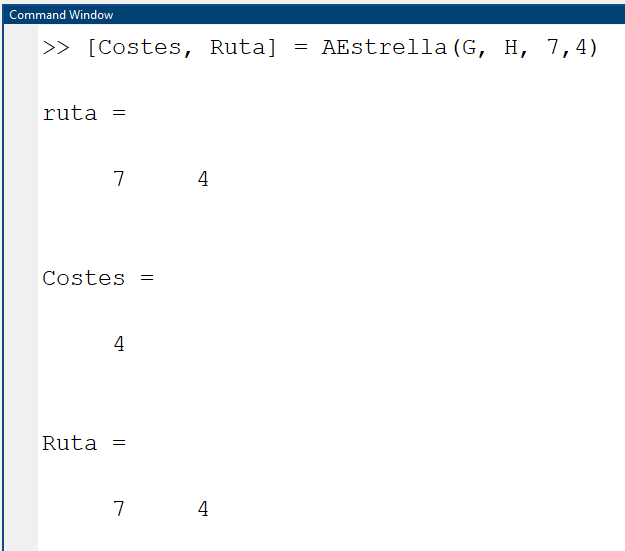


Figura : Resultado de Dijkstra\* con entrada 7 y destino 4.

Esto tiene un razonamiento lógico, pues este algoritmo se basa en ciertas estimaciones de la heurística para calcular el coste hacia el destino, pero en este caso hay muy poca distancia entre el inicio y la llegada por lo tanto la estimación no es totalmente precisa como sí era en el apartado uno, llegando pues a la conclusión de que este algoritmo será más eficiente en trayectorias de cierta longitud.

## Apartado 5.3

**Proponga una heurística admisible para que el resultado anterior si se corresponda al camino óptimo entre ambos nodos.**

Para dar una solución de continuidad a esta problemática, se propone implementar el código de Dijkstra proporcionado en la Práctica 4 para mejorar la matriz heurística entre dos puntos dados. Así pues, a razón de una mayor carga computacional se logra un camino adecuado garantizado, mayor que el de Dijkstra\* de la Práctica 5, pero menor que el de Dijkstra planteado en la Práctica 4, logrando así un término medio entre los dos algoritmos.

El código planteado es el que se observa en la Figura 52, donde como ya se ha dicho, se calculará la matriz heurística mediante el algoritmo de Dijkstra y se confirmará que es el adecuado (ver Figura 53).

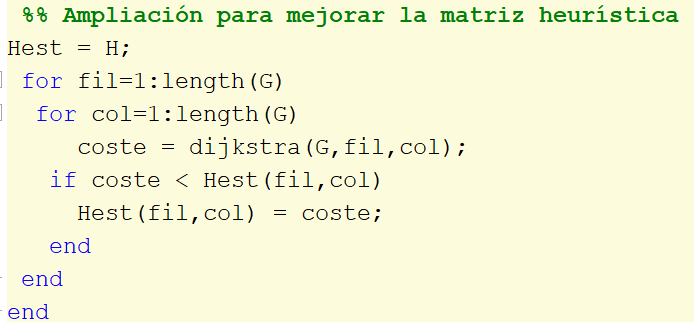


Figura : Corrección de código planteado.

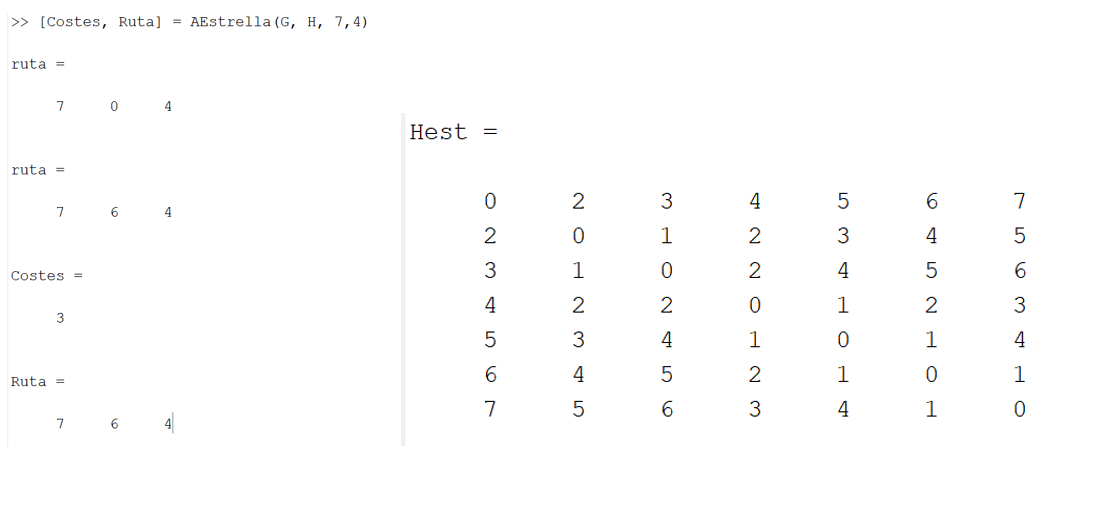


Figura :Problema solventado y matriz heurística corregida.

# Práctica 6: Navegación Autónoma

## Apartado 6.1

El objetivo principal de esta práctica es implementar tanto la planificación global de caminos, ya sea Dijkstra como Dijkstra\* con el método de campos potenciales visto en la Práctica número tres.

**Implementar un programa de Matlab que utilice la planificación de caminos de Dijkstra (práctica 4) y evite obstáculos con el método de campos potenciales (práctica 3). El programa debe preguntar los nodos de inicio y destino, y representar gráficamente el mapa con la trayectoria del robot. Debe indicar con un mensaje si se ha podido llegar al destino.**

En primer lugar, se va a leer y dibujar cada uno de los nodos que se nos proporciona. Para ello haremos uso del código mostrado en la Figura 54. Véase que se ha puesto un condicionante que, si estamos ante el nodo 10 en adelante, desplazamos el dibujo un poco para poder centrarlo.

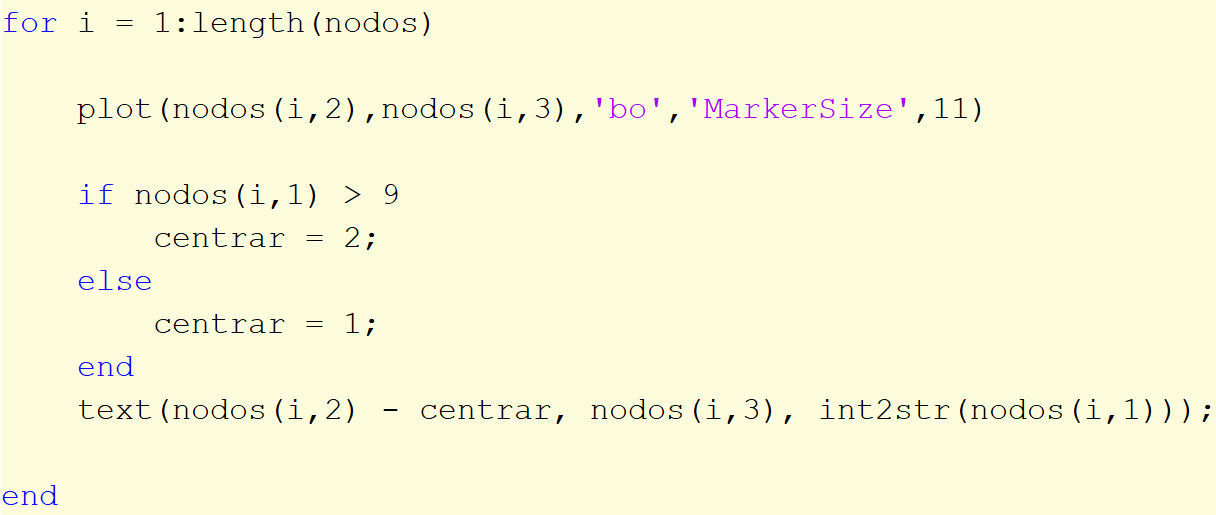


Figura : Código para dibujar los nodos.

Una vez está dibujado, se procede a los cálculos correspondientes. Se calcula el camino óptimo mediante la función Dijkstra, realizada anteriormente (ver Figura 55).

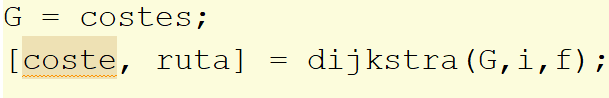


Figura : Llamada a la función Dijkstra.

Dado que el camino a seguir ya es conocido, se va a implementar un bucle for que vaya desde un nodo A hacia un nodo B y posteriormente de B hacia C, siendo A,B,C… los nodos indicados en la ruta mediante Dijkstra. Esto se representa en la Figura 56.

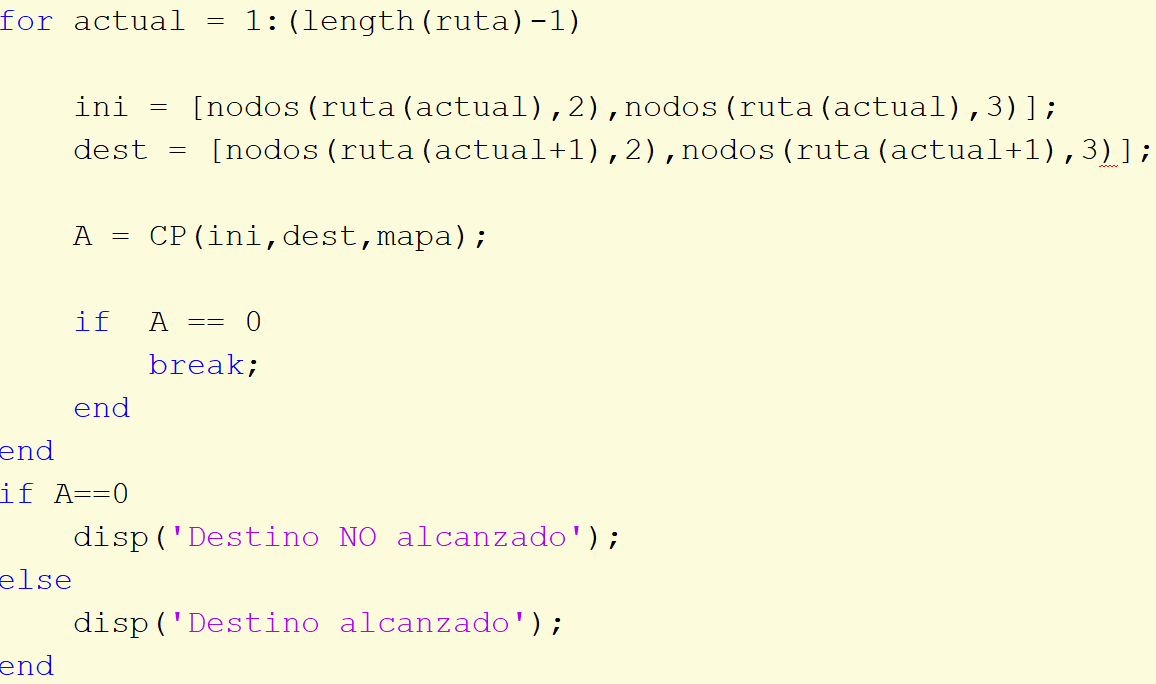


Figura : Campos potenciales en bucle.

Obsérvese como también se implementa un mensaje en caso de error, tal y como se solicita en las indicaciones.

Así pues, al ejecutar el código nos encontramos con un resultado satisfactorio (ver Figura 57 ).

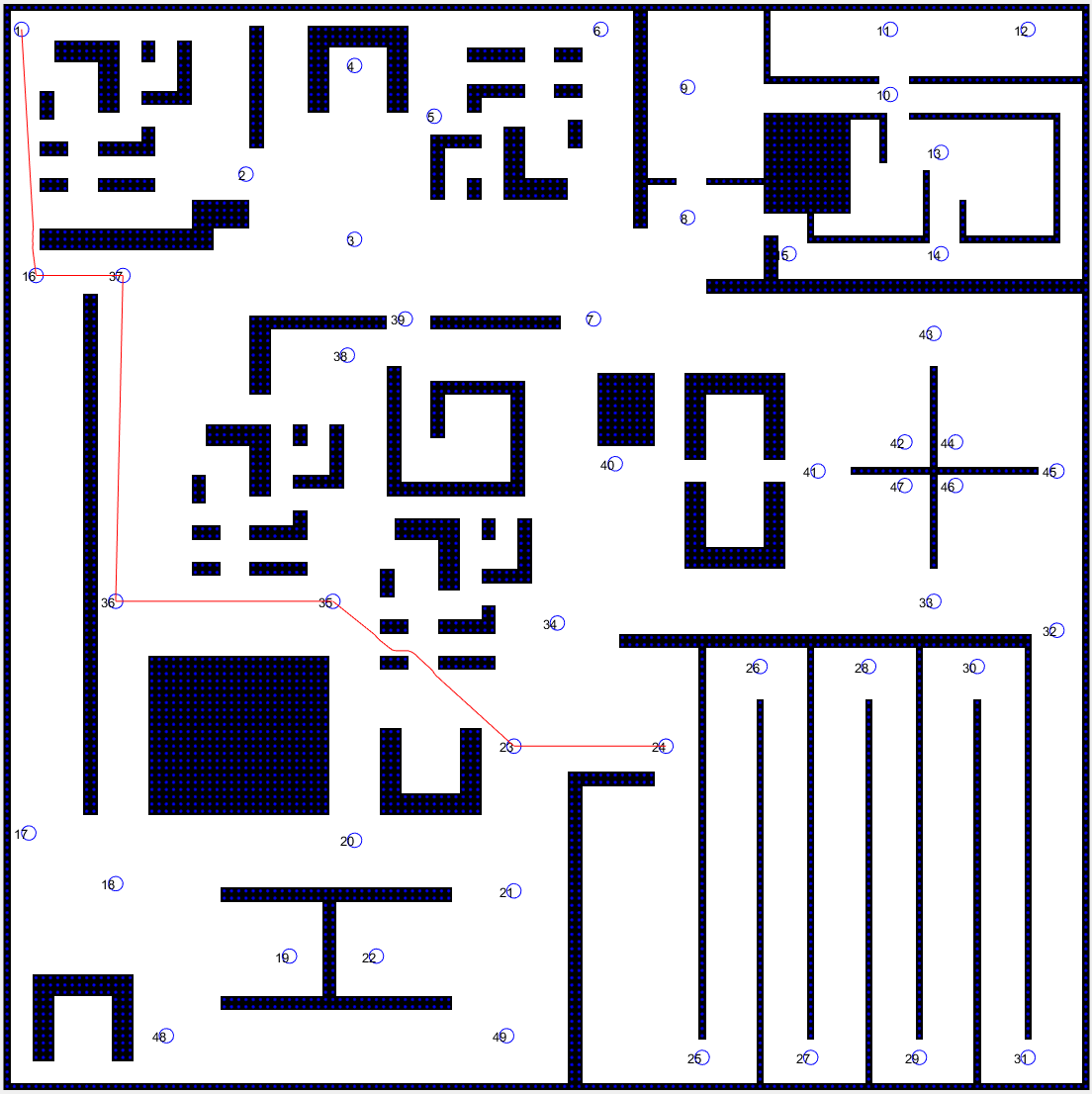


Figura : Resultado obtenido en el apartado 1.

## Apartado 6.2

**Realizar la trayectoria partiendo del nodo 1 hacia un destino en la parte derecha superior del mapa. Intentar atravesar la parte inferior derecha del mapa entre los nodos 24 y 32. ¿Es posible alcanzar el objetivo en esas situaciones?, ¿a qué se debe el problema para completar dichos caminos? Proponer una mejora a la navegación autónoma implementada para resolver esta situación.**

Se introduce como nodo de partida el nodo 1 y como destino el nodo 31 y se observa el siguiente resultado (ver Figura 58).

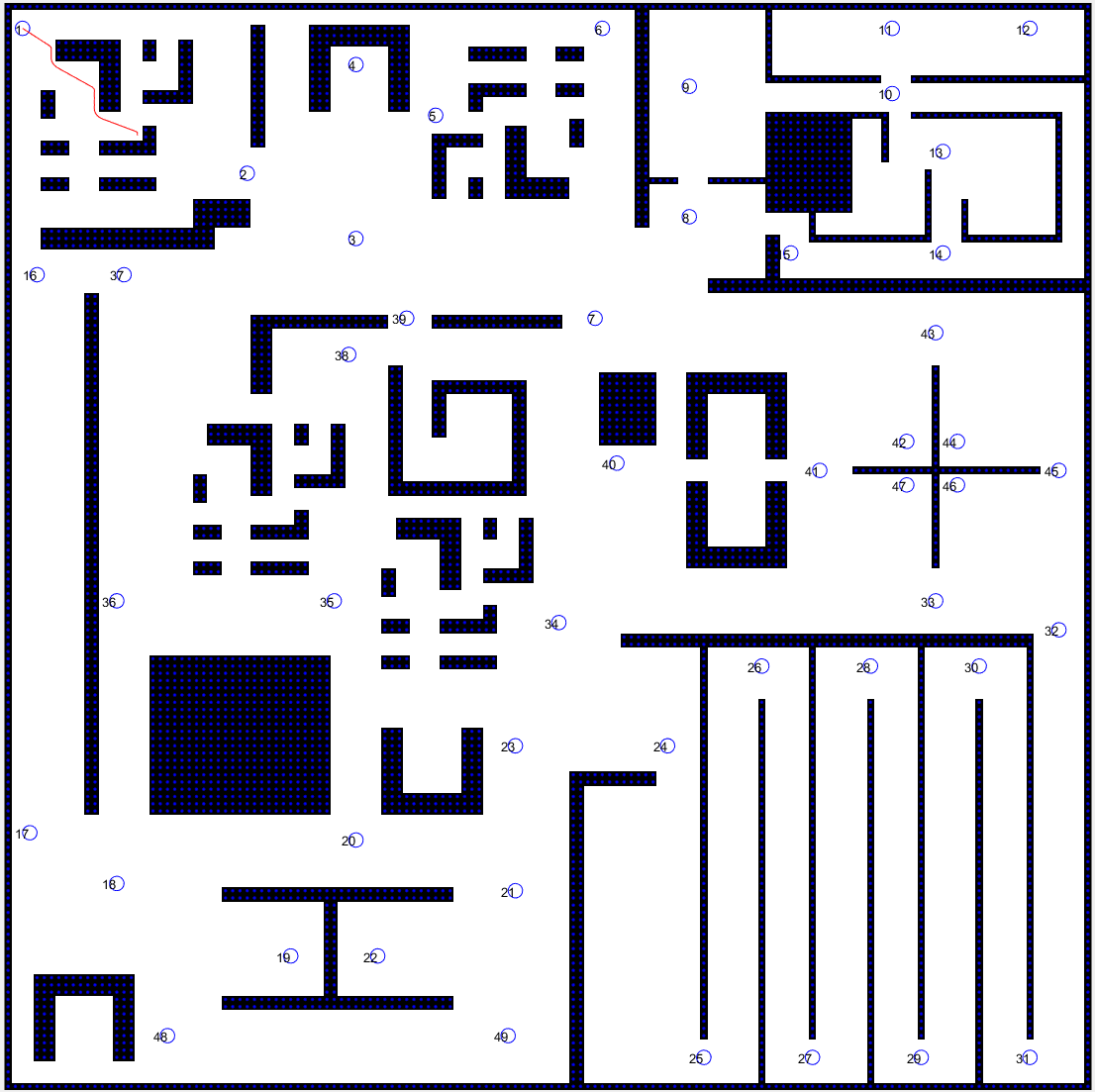


Figura : Resultado fallido con nodo de entrada 1 y nodo de salida 31.

Vemos que no llega al destino. Como ya se vio en la práctica 3 esto es debido a que el sistema ha entrado en una trampa local, impidiendo el desplazamiento como ya se indicó anteriormente.

Si se introdujeran más nodos alrededor de los puntos débiles del sistema se podría solucionar este problema, tal y como se ha hecho, por ejemplo, en la cruz de la derecha, involucrando a los nodos 33, 41, 42, 43, 44, 45, 46 y 47. Así pues, en lugar de ir del nodo 47 al 42, por ejemplo, pasaría por el nodo intermedio 41.

## Apartado 6.3

**Si se desea utilizar el algoritmo A\* para realizar una planificación de caminos más eficiente desde el punto de vista computacional, es necesario definir una heurística. Se propone definir una heurística consistente y reemplazar el algoritmo de Dijkstra por el A\* para la planificación de caminos.**

Se muestra la Figura 59 para construir la matriz heurística. Tal y como se observa, tiene como entrada los nodos proporcionados en el enunciado y la matriz de adyacencia.

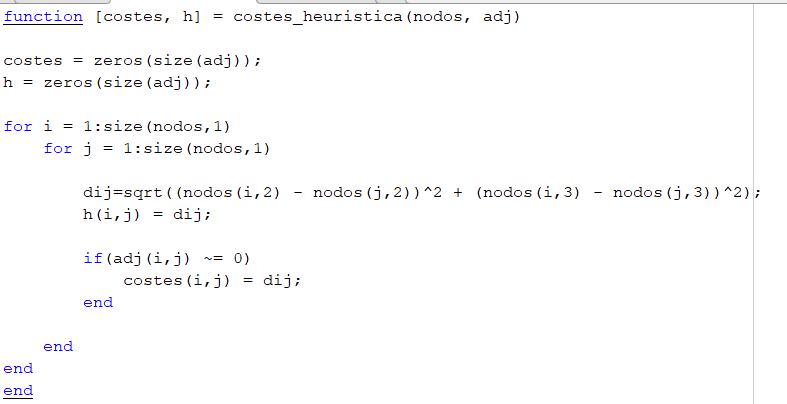


Figura : Construcción de la nueva matriz

Se calculará posteriormente la distancia euclídea entre cada uno de los nodos y se almacenarán los resultados en forma de matriz.

Siempre y cuando haya adyacencia entre dos nodos, se le asignarán la distancia euclídea a la variable de costes.

Así pues, se observan los resultados tras calcular la nueva matriz de costes y emplear el algoritmo de Dijkstra\* (ver Figura 60).

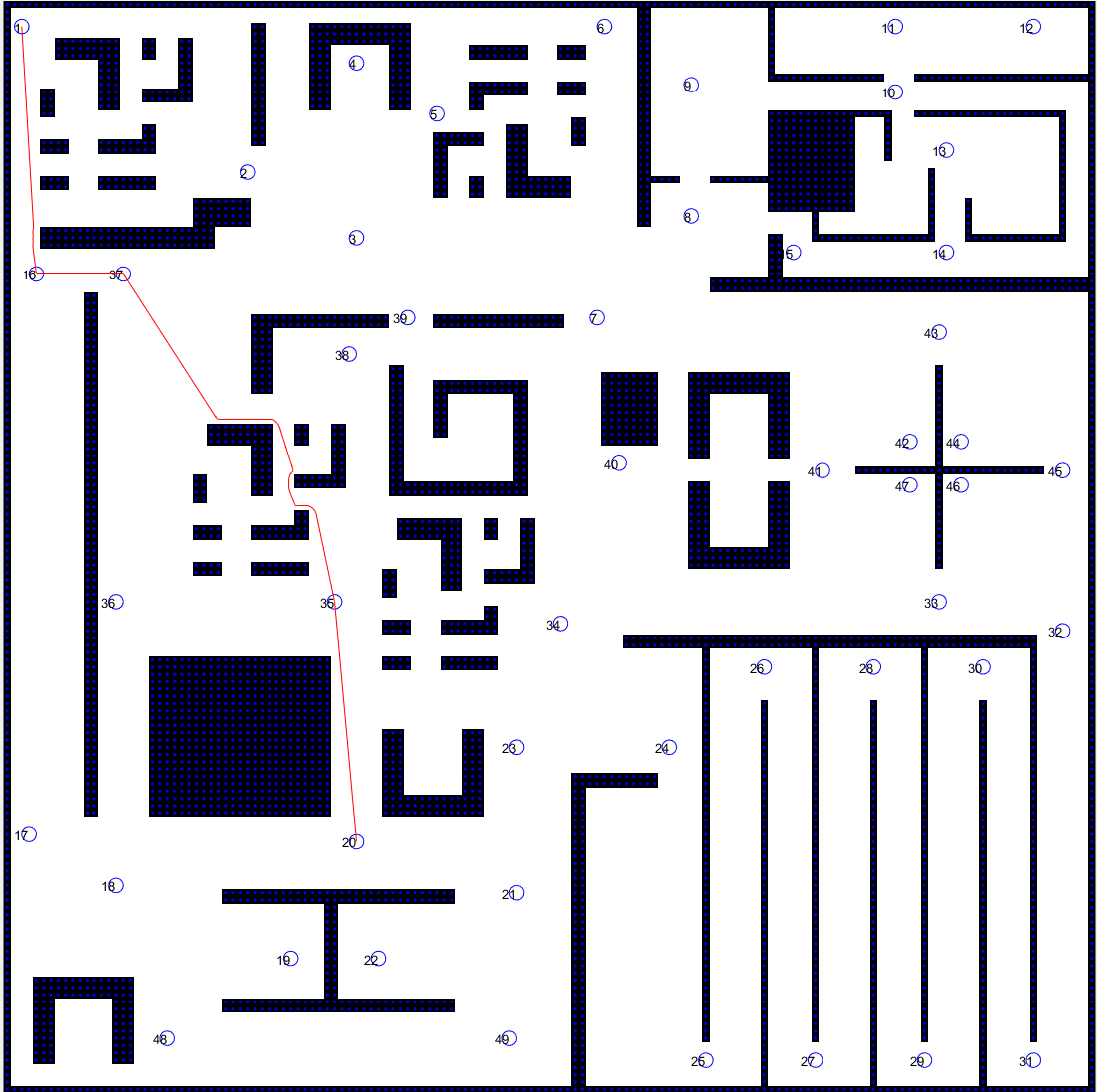


Figura : Resultado tras la ejecución del código del apartado 3.