Marta Rosa Flores // Daniel Triano Moreno

ampliación de robótica

Memoria de prácticas: Manipuladores

Índice:

[Práctica 1: Planificación cartesiana - cuaternios. 2](#_Toc73441927)

[Apartado 1.1: 2](#_Toc73441928)

[Apartado 1.2: 4](#_Toc73441929)

[Apartado 1.3: 5](#_Toc73441930)

[Práctica 2: Control de fuerzas indirecto 7](#_Toc73441931)

[Apartado 2.1 7](#_Toc73441932)

[Apartado 2.2 10](#_Toc73441933)

[Apartado 2.3 12](#_Toc73441934)

[Práctica 3: Evitar Obstáculos 15](#_Toc73441935)

[Apartado 3.1 15](#_Toc73441936)

[Apartado 3.2 17](#_Toc73441937)

[Apartado 3.3 18](#_Toc73441938)

[Práctica 4: Planificación de caminos (Dijkstra) 20](#_Toc73441939)

[Apartado 4.1 20](#_Toc73441940)

[Práctica 5: Ampliación de caminos (A\*) 28](#_Toc73441941)

[Apartado 5.1 28](#_Toc73441942)

[Apartado 5.2 31](#_Toc73441943)

[Apartado 5.3 31](#_Toc73441944)

[Práctica 6: Navegación Autónoma 33](#_Toc73441945)

[Apartado 6.1 33](#_Toc73441946)

[Apartado 6.2 35](#_Toc73441947)

[Apartado 6.3 36](#_Toc73441948)

# 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.

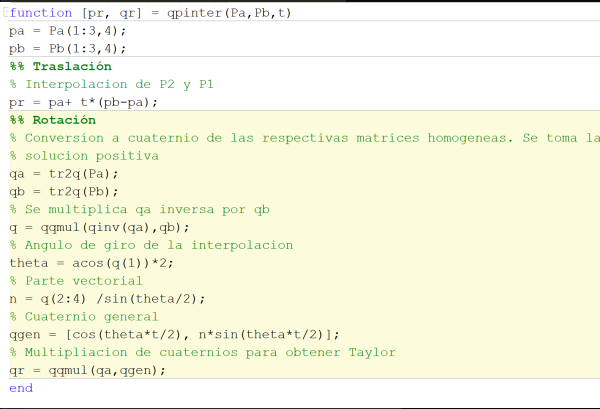


Figura 1: Función qpinter.

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.

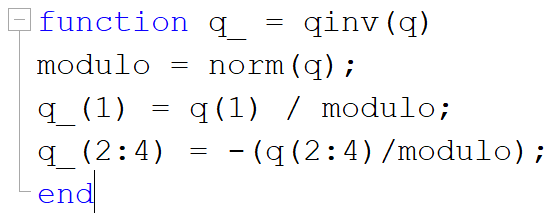


Figura 2- Función qinv que invierte un cuaternio

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

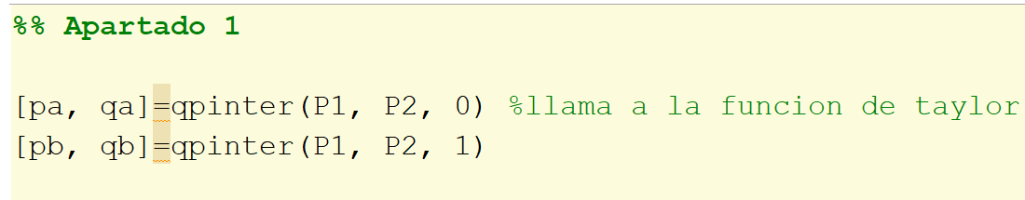


Figura 3

El resultado es el observado en la Figura 4.

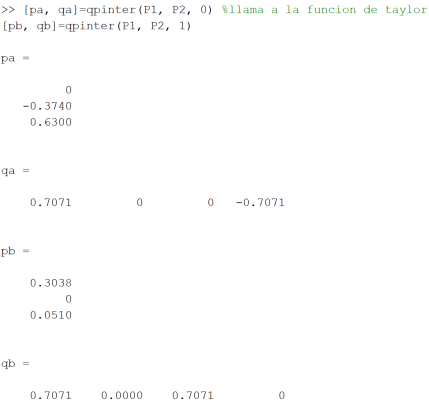


Figura 4: Resultado tras ejecutar la función Taylor.

## Apartado 1.2:

**Realizar una función MATLAB con el formato P=tramo(P0,P1,P2,tau,T,t) que calcula la transformación P correspondiente al movimiento de P0 a P2 vía P1 suavizado por el método de Taylor. Los parámetros tau y T, corresponden respectivamente al intervalo de transición y tiempo total empleado en recorrer el camino tal y como se muestra en la figura 1, y t indica el tiempo en el cual se alcanza la localización del camino P calculada.**

Para poder realizar este apartado, se hará uso de las ecuaciones X y X.

Se crea la función “tramo”, tal y como se observa en la Figura 5. En esta función se calculan los distintos cuaternios de Taylor iniciales dependiendo del tramo en el que nos encontramos gracias a las funciones condicionales que se detallan en el código.

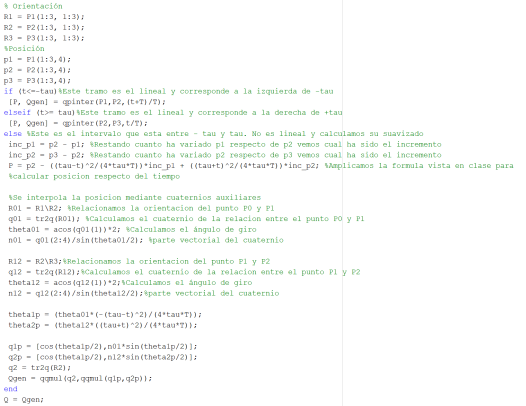


Figura 5 – Script de la función tramo.

Una vez se han reconocen las distintas regiones de trabajo, se procede a implementar las ecuaciones X, X.

## Apartado 1.3:

**Representar gráficamente la evolución de la posición y la orientación (en ángulos de EulerZYZ) a lo largo de toda la trayectoria.**

Se ejecuta el script proporcionado por el enunciado para comprobar la correcta ejecución de las funciones implementadas, obteniendo los resultados mostrados en las siguientes figuras.

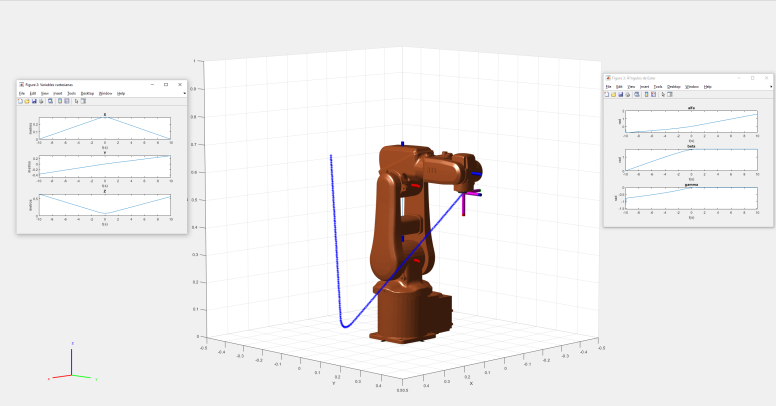


Figura 6: Trayectoria seguida por el robot

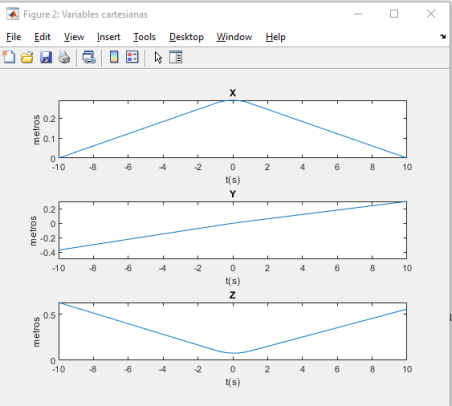


Figura 7: Variación de la trayectoria en variables cartesianas

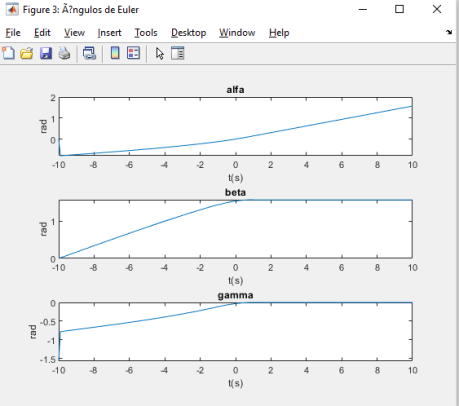


Figura 8: Variación de la trayectoria en ángulos de Euler ZYZ

# Práctica 2: Control de fuerzas indirecto

## Apartado 2.1

**Si se considera una elasticidad del entorno sobre el eje X igual a kx=103 N/m, calcular la frecuencia natural no amortiguada y el coeficiente de amortiguación del modelo dinámico equivalente del robot en contacto con el entorno. Al tomar kx=104 N/m, calcular los nuevos valores de los parámetros dinámicos.**

En este ejercicio, se realizará un control de fuerzas resistivo utilizando las ecuaciones dinámicas dadas en el enunciado para aplicar la ley de control sobre una superficie suave y elástica.

En primer lugar, se han considerado las ecuaciones presentes en el campus virtual y replicadas a continuación.

|  |
| --- |
|  |
|  |

Introducciendo dichas ecuaciones en MATLAB (ver fig) y utilizando los valores numéricos proporcionados en el enunciado, obtenemos los siguientes resultados.

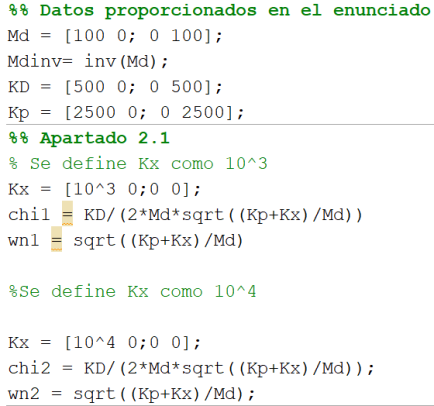


Figura : Cálculo

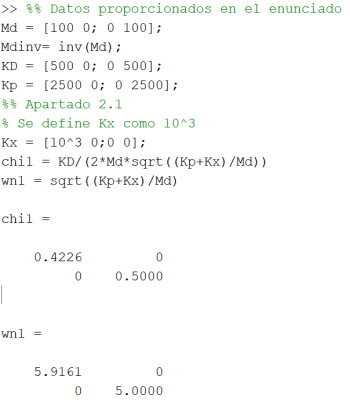


Figura : Resultados obtenidos para la frecuencia natural y el coeficiente de amortiguación para Kx=103

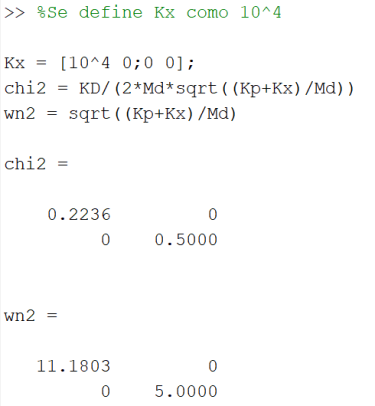


Figura : Resultados obtenidos para la frecuencia natural y el coeficiente de amortiguación para Kx=103

## Apartado 2.2

**Construir un modelo en Simulink para realizar la simulación dinámica del manipulador en contacto con el entorno, mediante el uso del modelo equivalente, si la posición de contacto es [1, 0], la deseada es [1’1, 0’1] con los dos valores de elasticidad del entorno considerados en el apartado primero.**

Para este apartado, despejamos la ecuación dada por el enunciado y se he reproducido en Simulink, tal y como se aprecia en Figura 12.

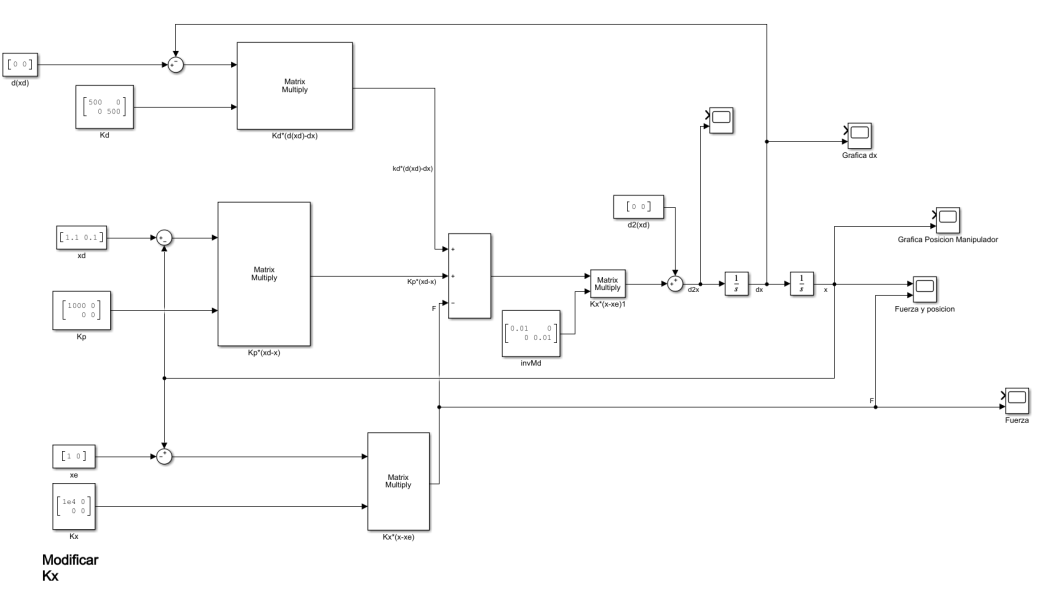


Figura : Simulación de la ecuación del manipulador.

Se han simulado las respuestas y estas se observan en las fig. X, X. Como podemos ver, se alcanza régimen estacionario en la salida tal y como deseábamos. Se han simulado las posiciones para Kx=103 y Kx=104.



Figura : Posición del manipulador con Kx=103



Figura : Posición del manipulador con Kx=104

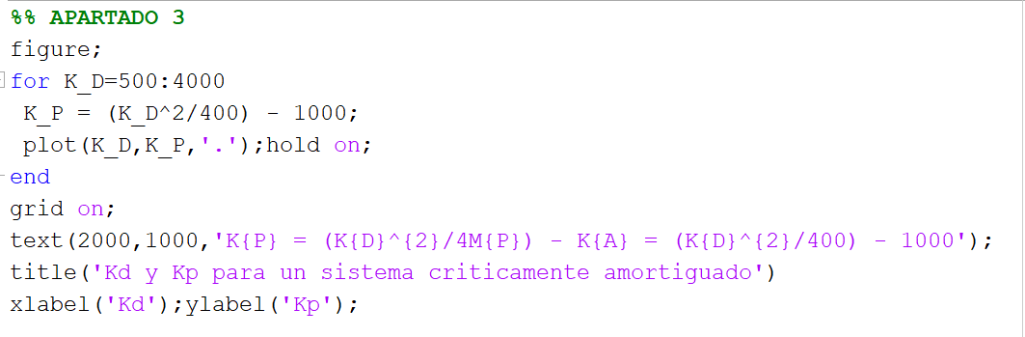
Para el segundo caso, se observa una mayor sobreoscilación que en el primero y un mayor error en régimen permanente. También se aprecia cómo en el segundo caso se tarda más en lograr la estabilidad.

## Apartado 2.3

**Calcular un conjunto de parámetros del controlador que verifiquen que el comportamiento transitorio del sistema se encuentre críticamente amortiguado. ¿Existe un conjunto único de valores?**

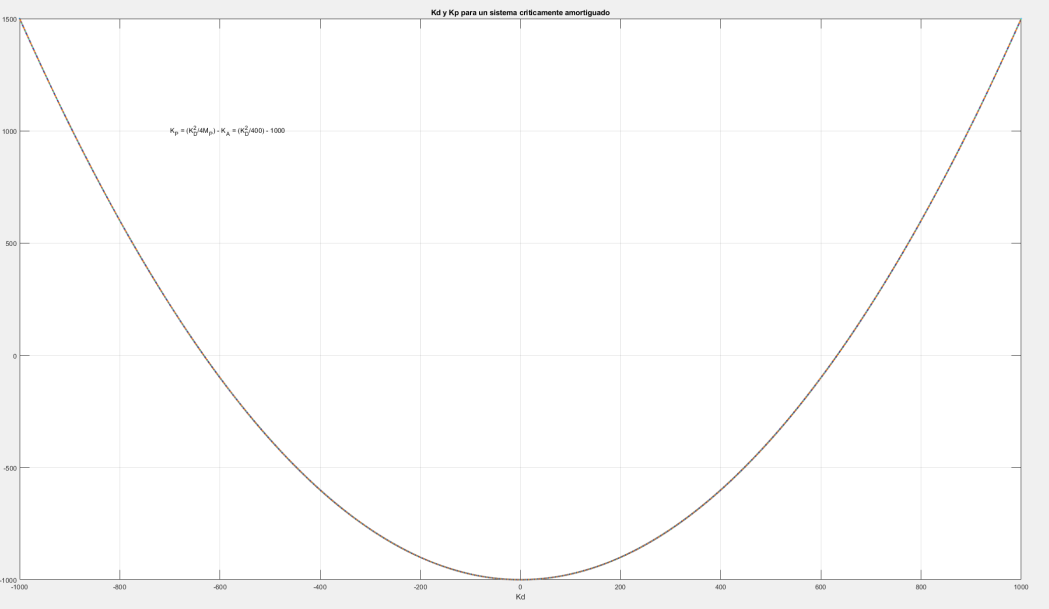
Establecemos el valor de como condición para que el sistema sea críticamente amortiguado. Teniendo en cuenta las siguientes ecuaciones, despejamos los valores de Kp y Kd.

En el siguiente script de MATLAB, utilizamos la primera de las ecuaciones con un valor de .

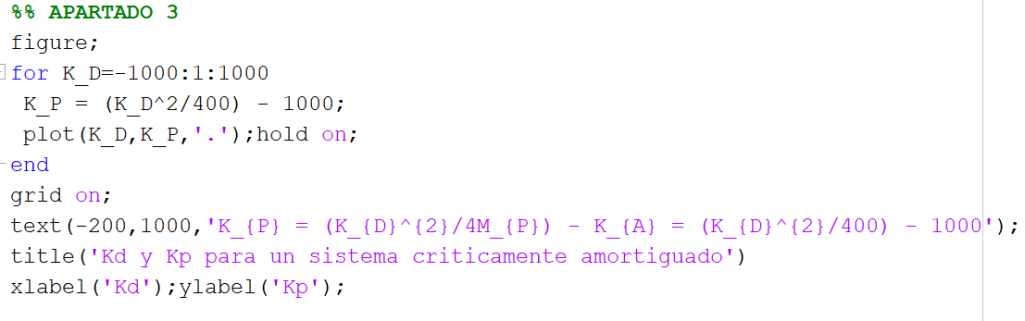


Figura

El resultado se aprecia en la Figura 16. Un rango notable de las soluciones obtenidas es negativo, por lo tanto, se ha decidido acotar el sistema para obtener valores de Kp y Kd positivos. Para ello se ha implementado el código mostrado en la fig X.

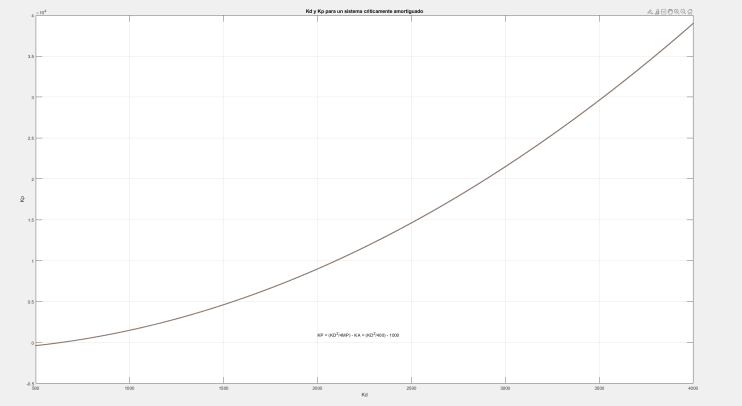


Figura



Figura

En la fig X, se observa el resultado de ejecutar dicho código, y se muestra un rango de soluciones positivas.



Figura

# 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).

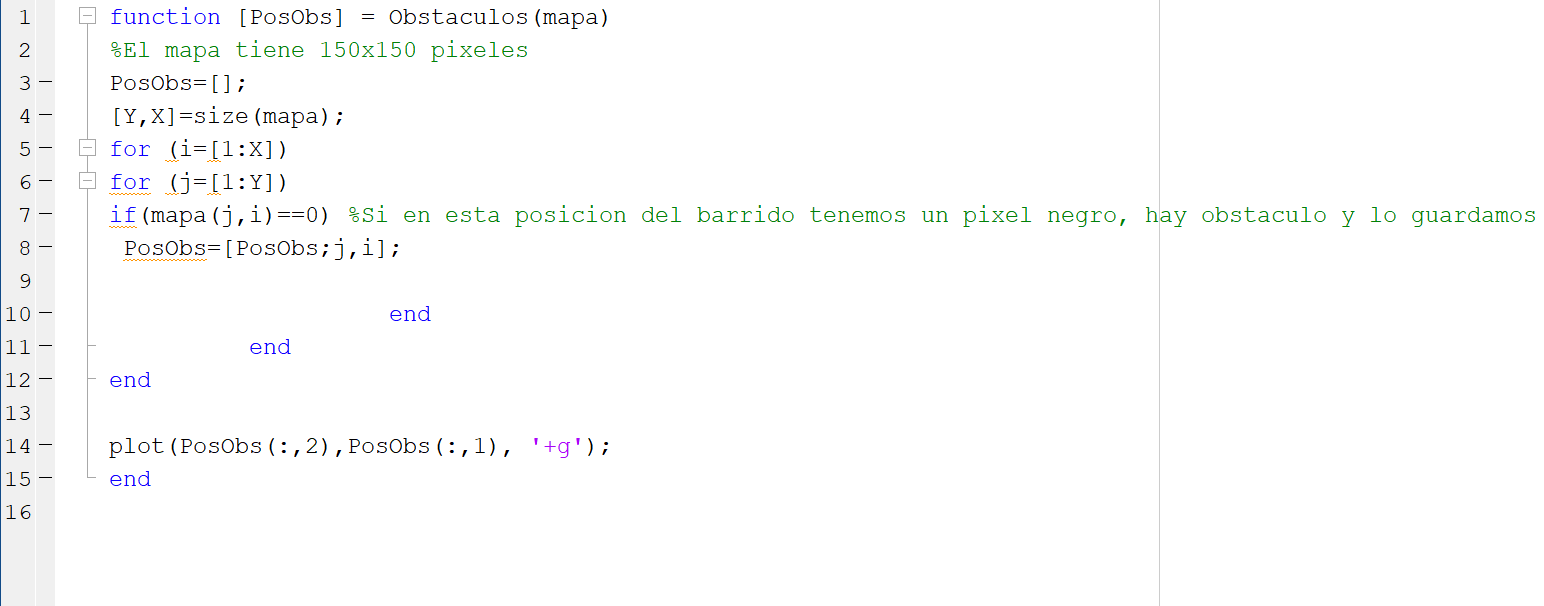


Figura 19: Función Obstáculos para detectar las discontinuidades en el mapa.

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) | (10) |
| = | (17) | (11) |

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.

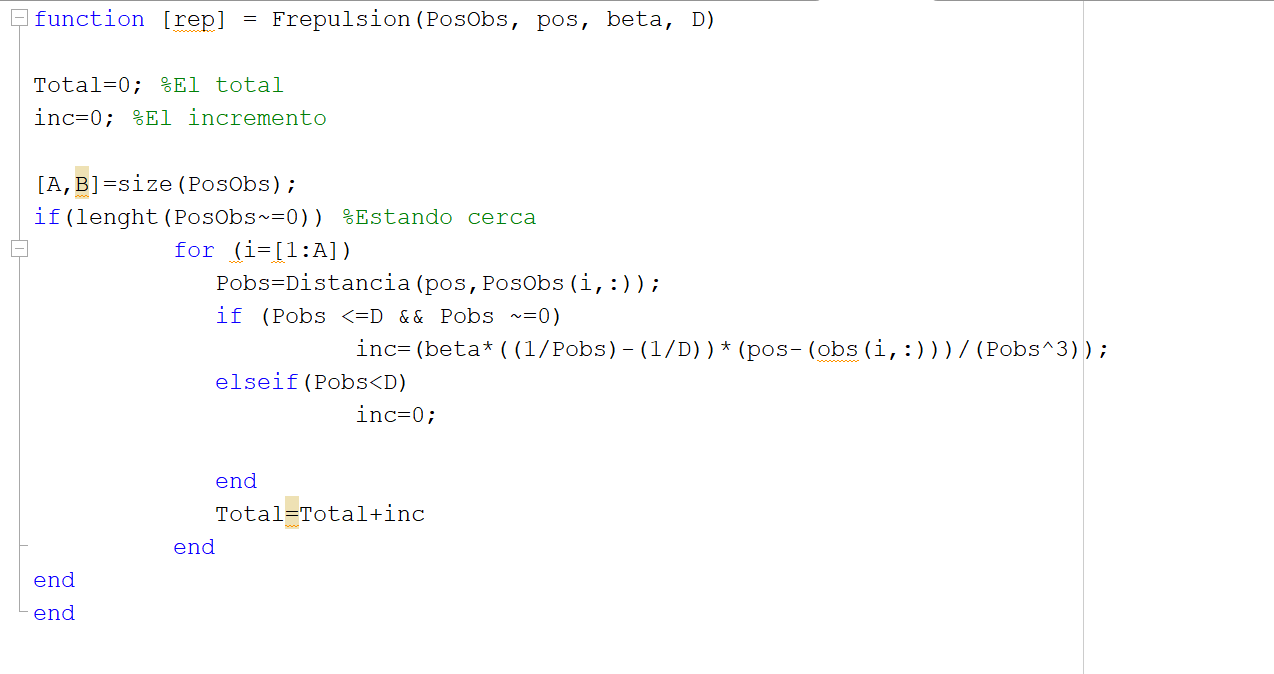


Figura 20: Configuración de las fuerzas de repulsión.

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

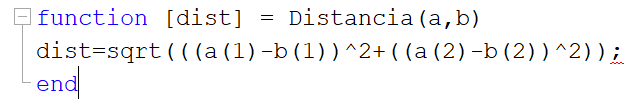


Figura 21: Función distancia.

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.

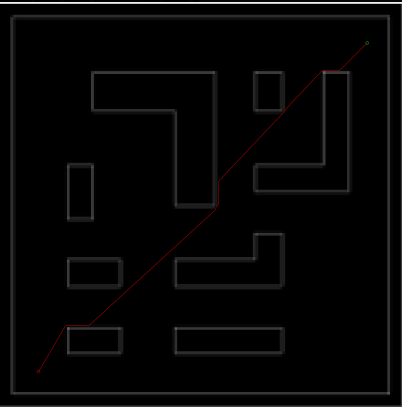


Figura 22: Recorrido seguido por el robot hasta llegar al punto final.

## 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:

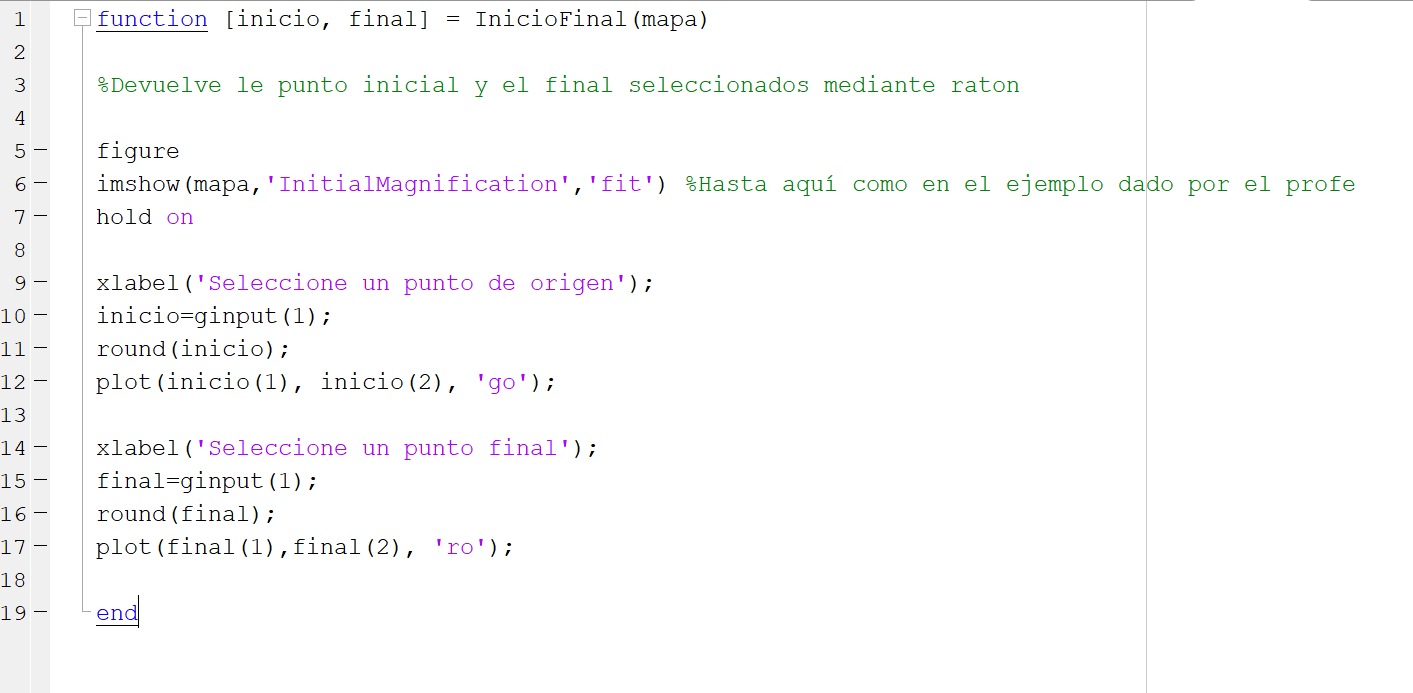


Figura 23: Código para introducir los puntos de origen y destino manualmente.

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

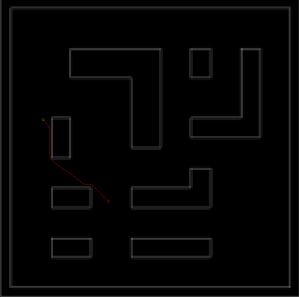


Figura 24: El robot realizando una ruta para llegar al destino indicado.

## 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.

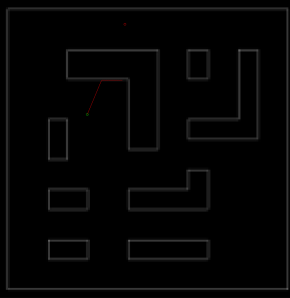


Figura 25: Se observa un mínimo local del que el robot no logra salir.

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).

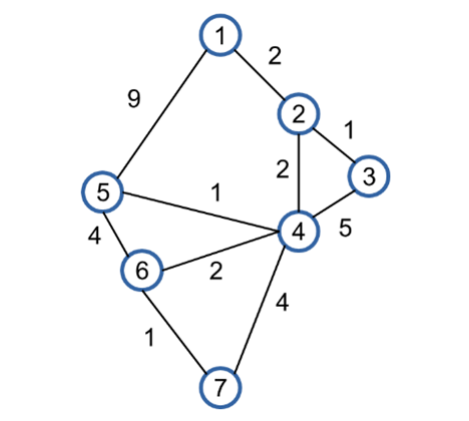


Figura 26: Mapa topológico de costes y nudos.

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

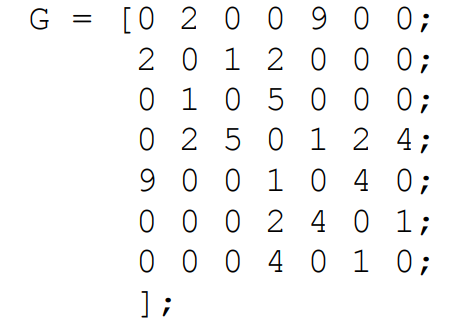


Figura 27: Matriz topológica de costes.

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 28: 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.

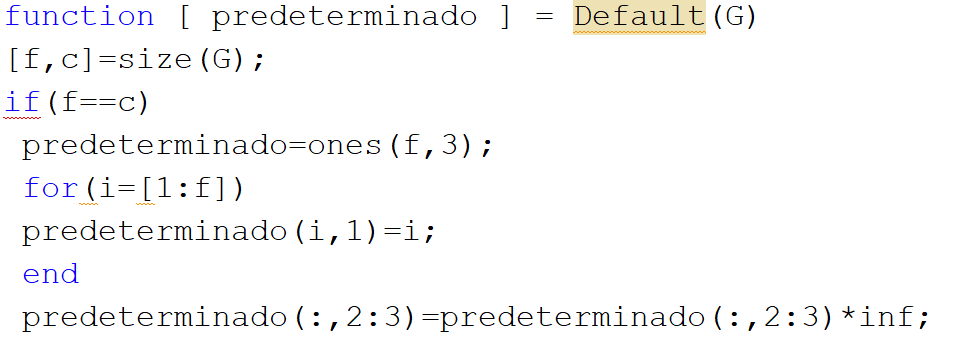


Figura 29: Función Default.

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.

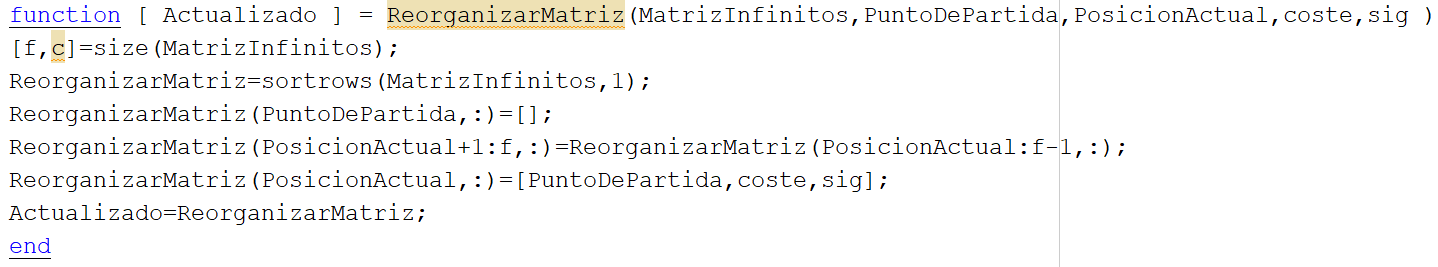


Figura 30: Función ReorganizarMatriz.

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.

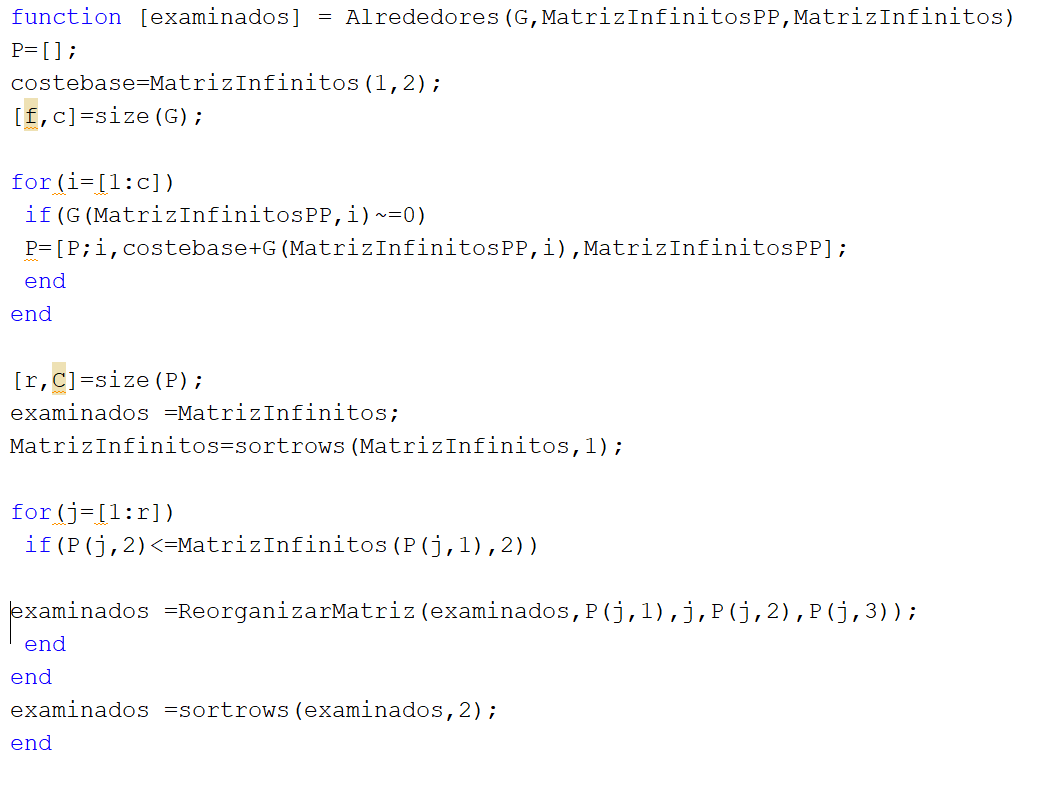


Figura 31: Función Alrededores.

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.

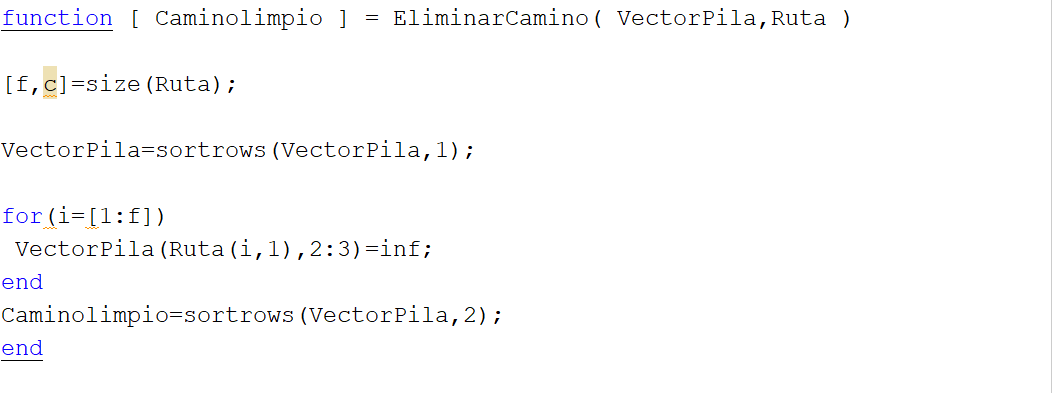


Figura 32: Función EliminarCamino.

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.

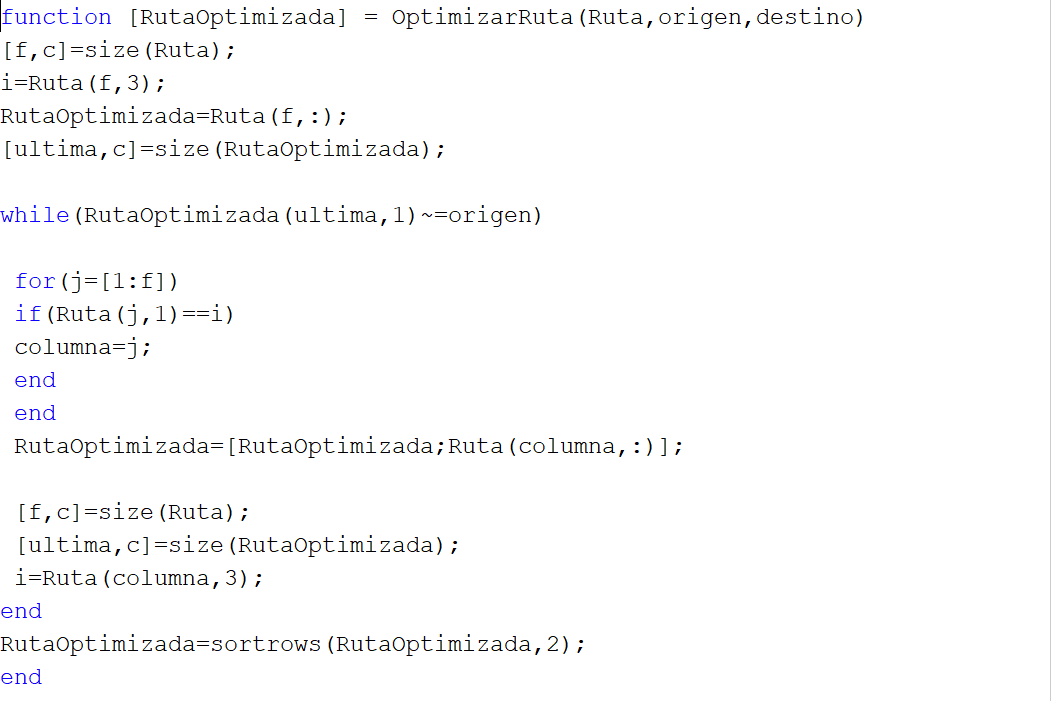


Figura 33: Función optimizar ruta.

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).

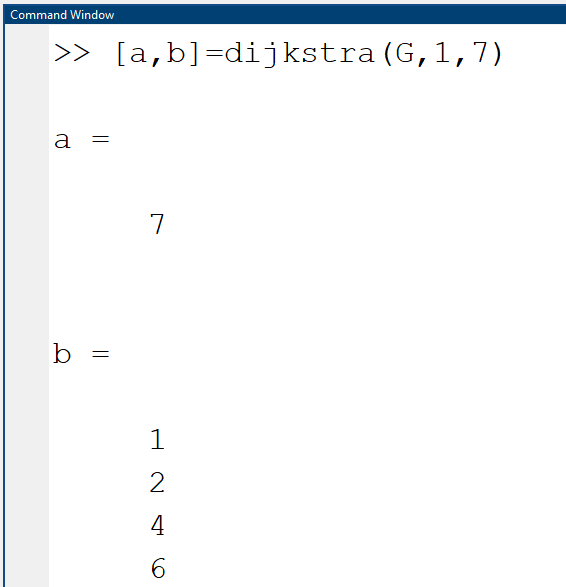


Figura 34: Resultados obtenidos partiendo del nodo 1 hacia el 7.

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

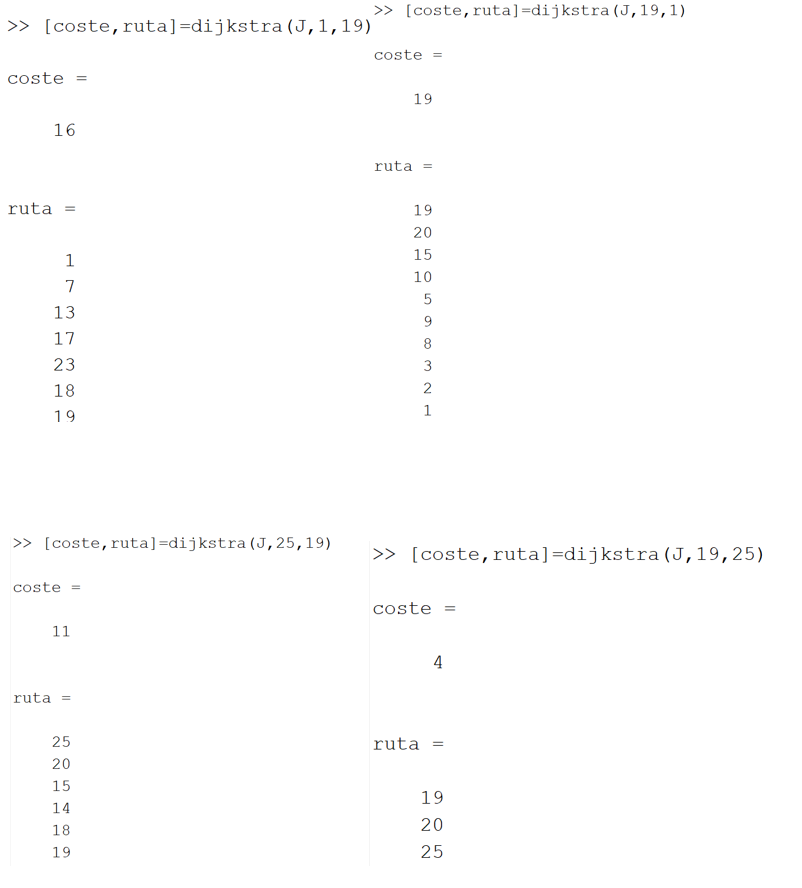


Figura 35: Resultados obtenidos (1).

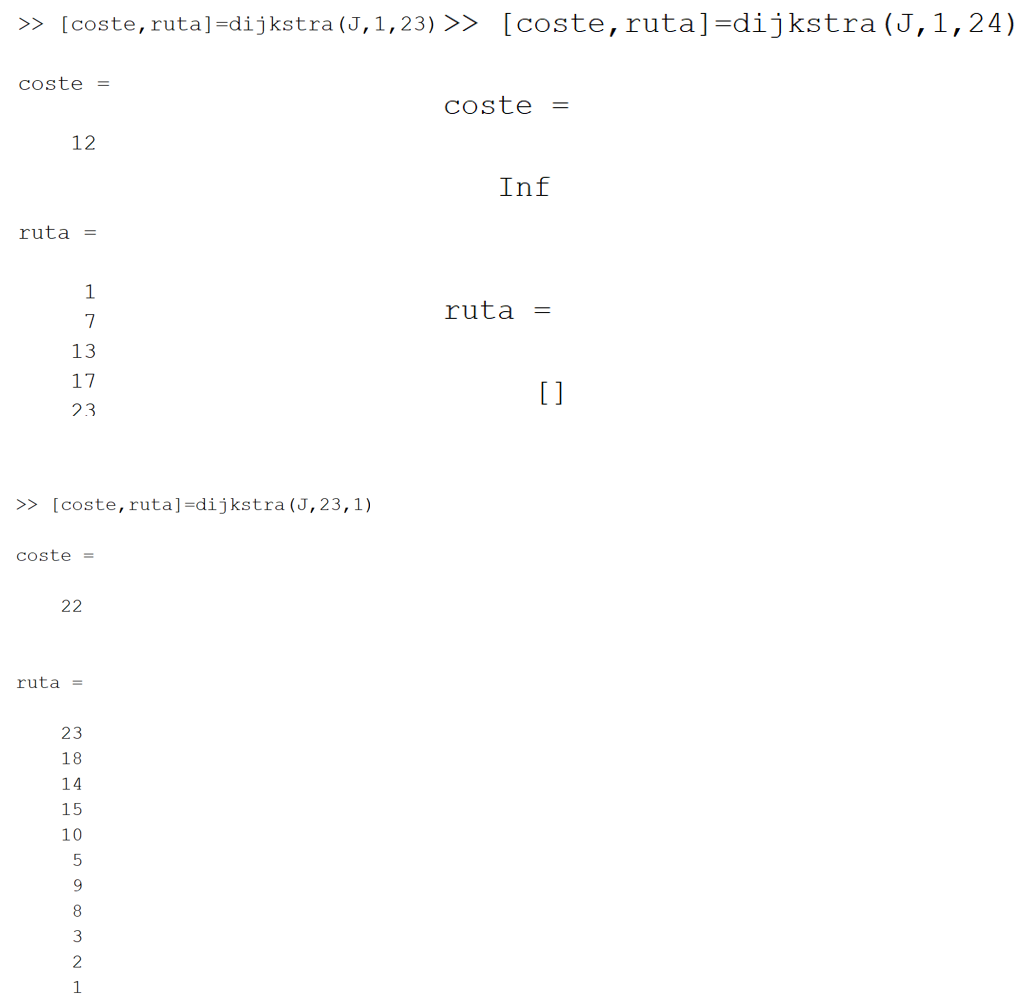


Figura 36: Resultados obtenidos (2).

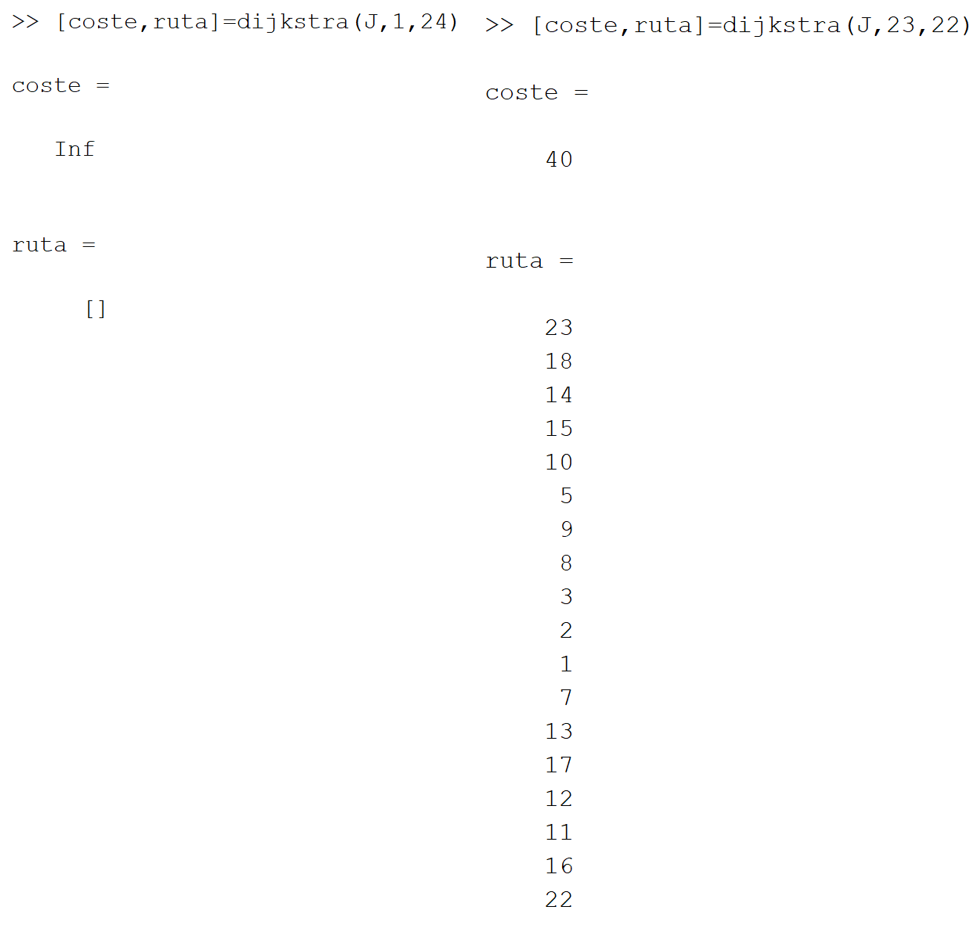


Figura 37: Resultados obtenidos (3).

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.

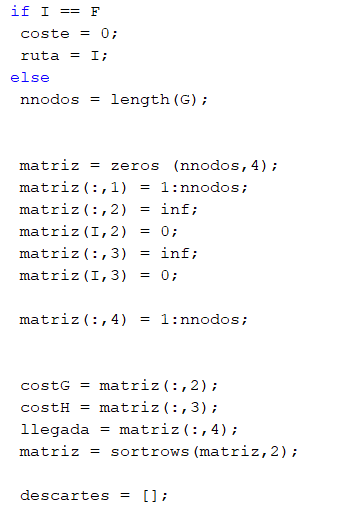


Figura 38: Análisis iterativo del código

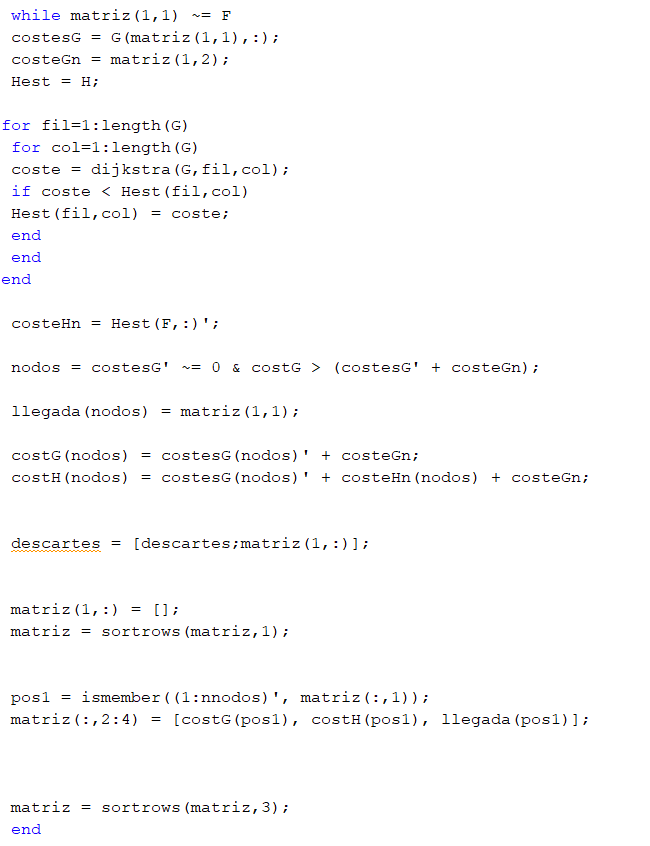


Figura 39: Inicialización del código. Incluye la posterior mejora.

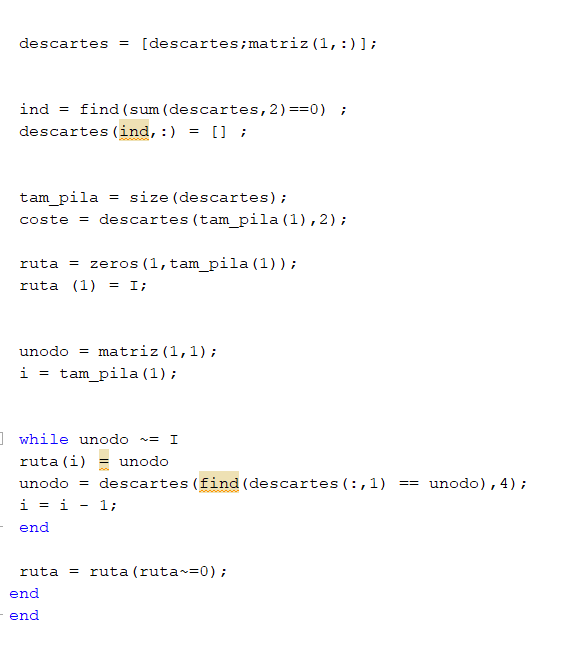


Figura 40: Parte final del código, se ordenan los resultados.

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:

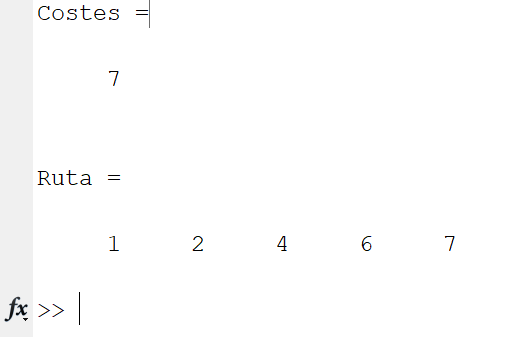


Figura 41: Resultados de Dijkstra\* con nudo de entrada 1 y llegada 7.

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).

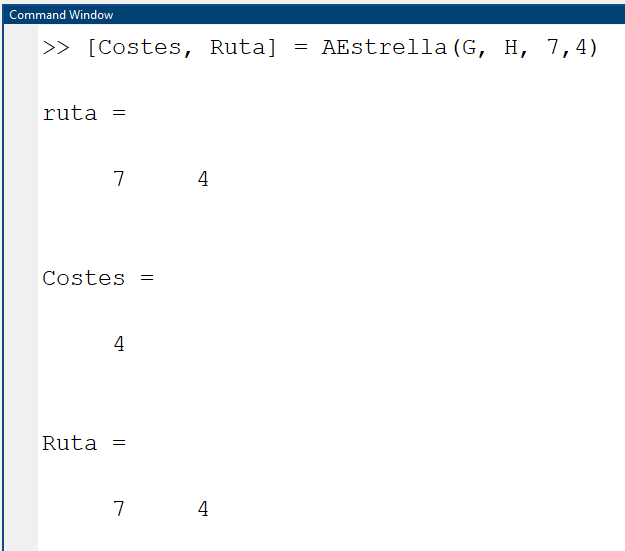


Figura 42: Resultado de Dijkstra\* con entrada 7 y destino 4.

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).

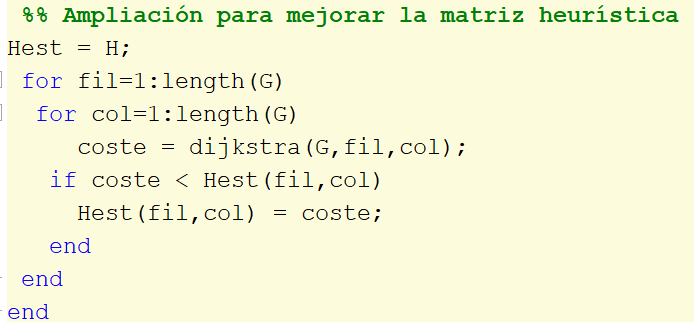


Figura 43: Corrección de código planteado.

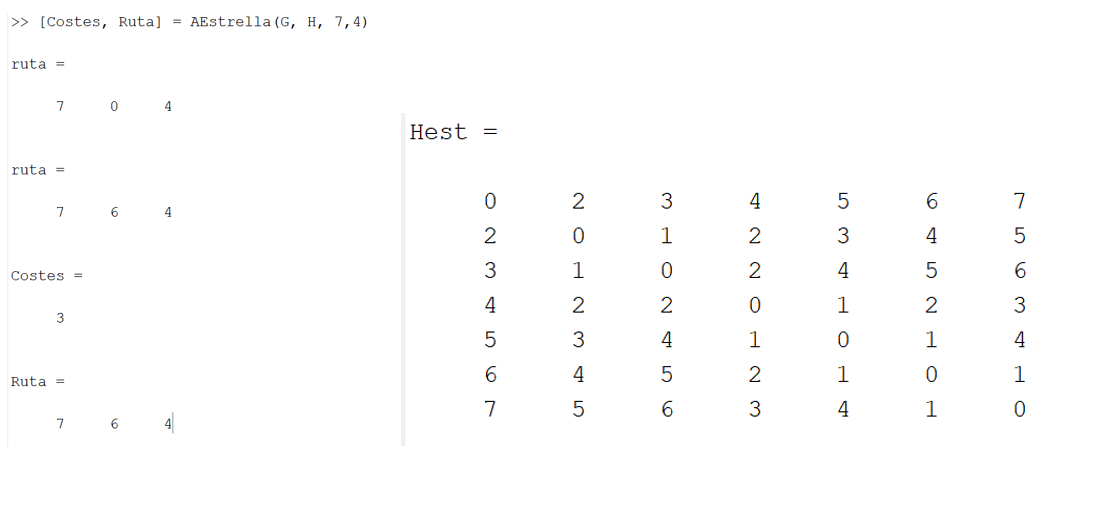


Figura 44:Problema solventado y matriz heurística corregida.

# 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.

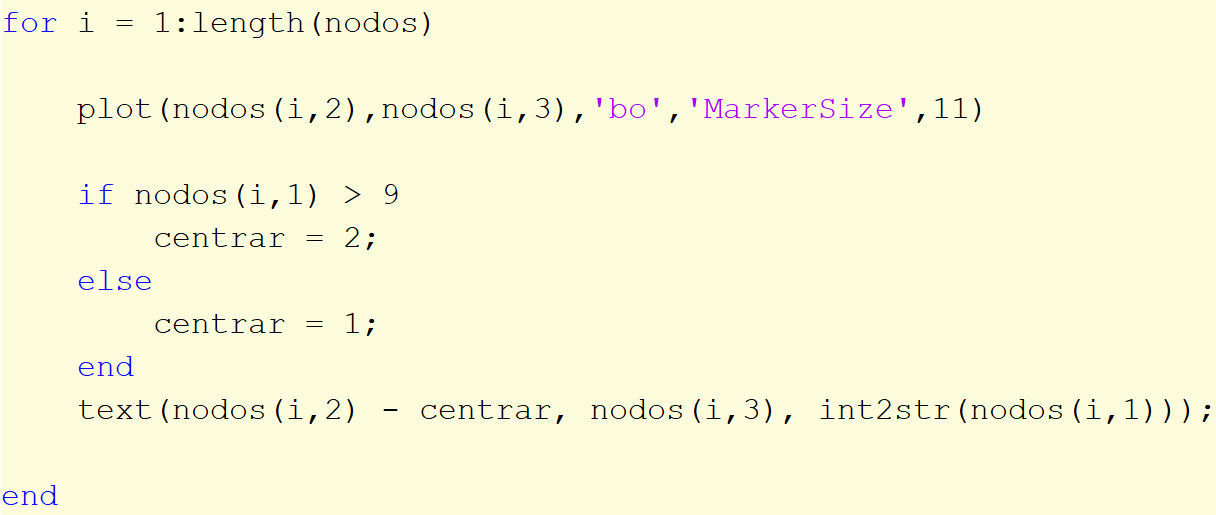


Figura 45: Código para dibujar los nodos.

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

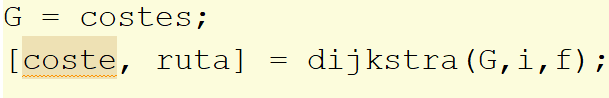


Figura 46: Llamada a la función Dijkstra.

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.

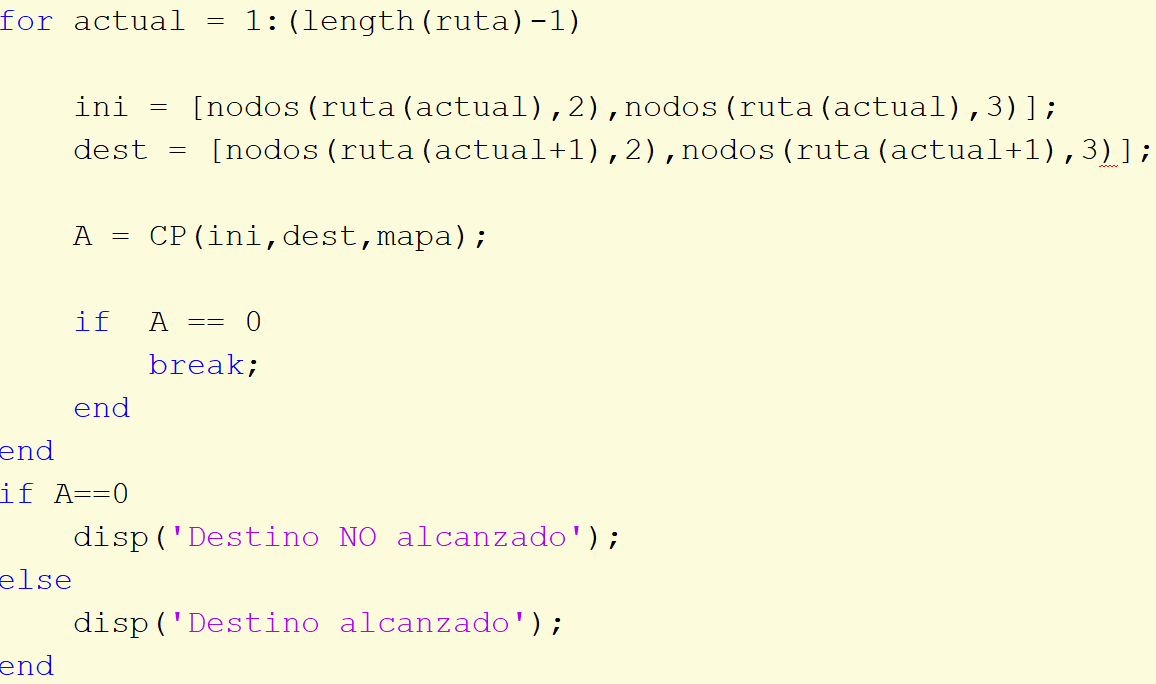


Figura 47: Campos potenciales en bucle.

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 ).

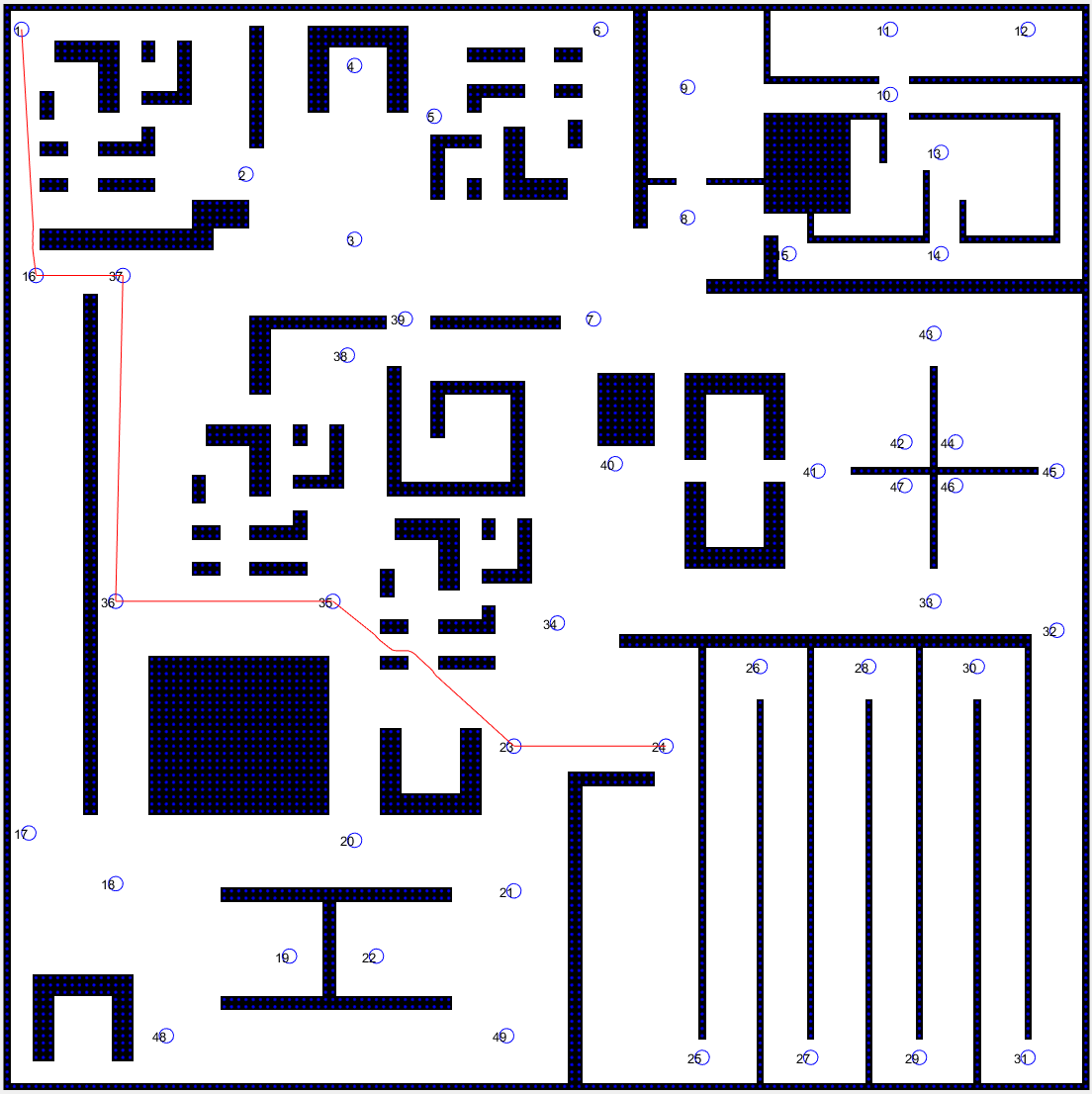


Figura 48: Resultado obtenido en el apartado 1.

## 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).

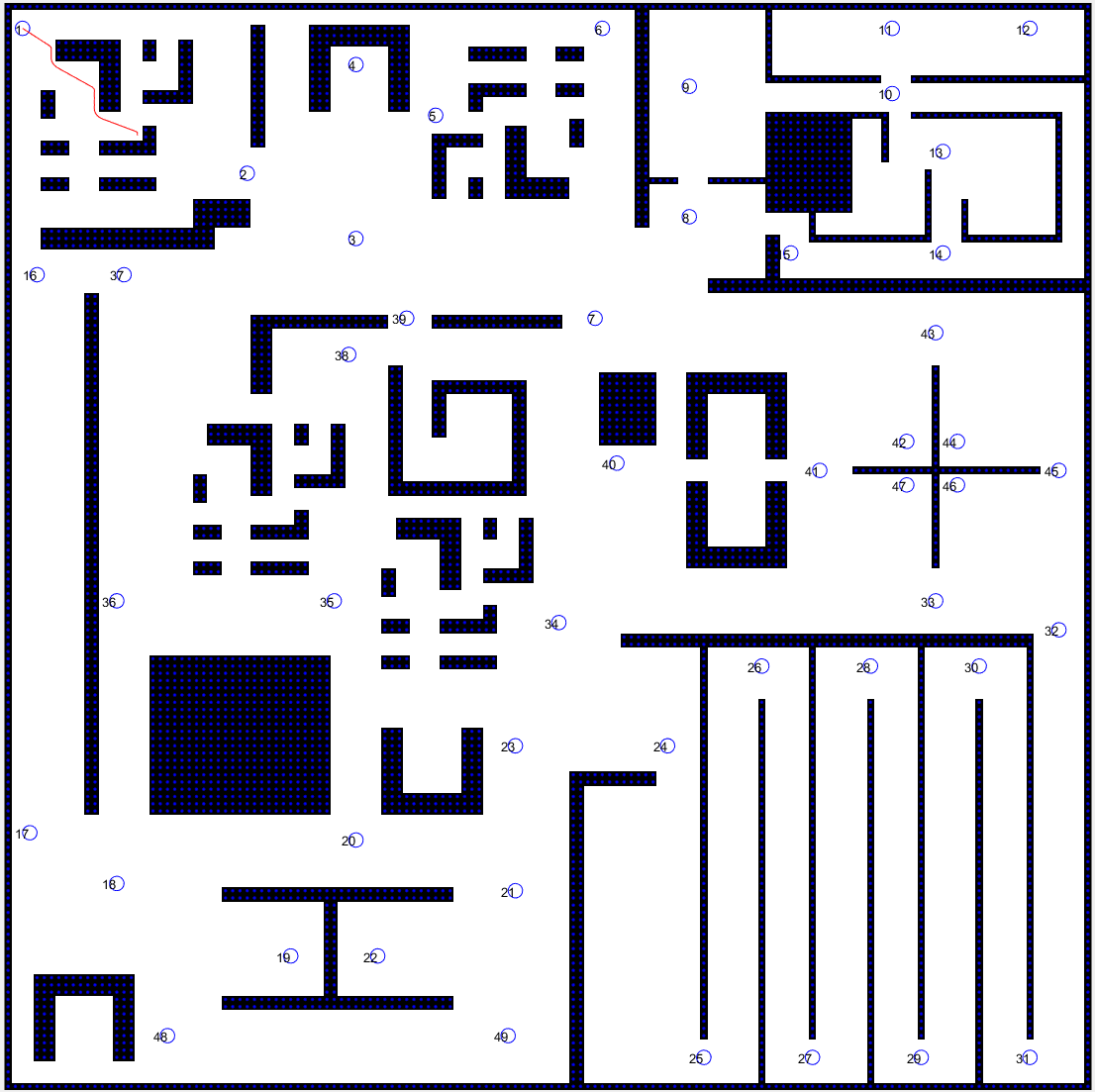


Figura 49: Resultado fallido con nodo de entrada 1 y nodo de salida 31.

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.

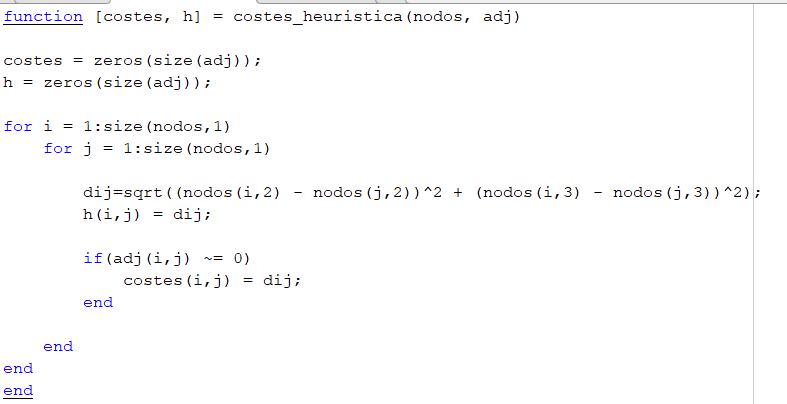


Figura 50: Construcción de la nueva matriz

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).

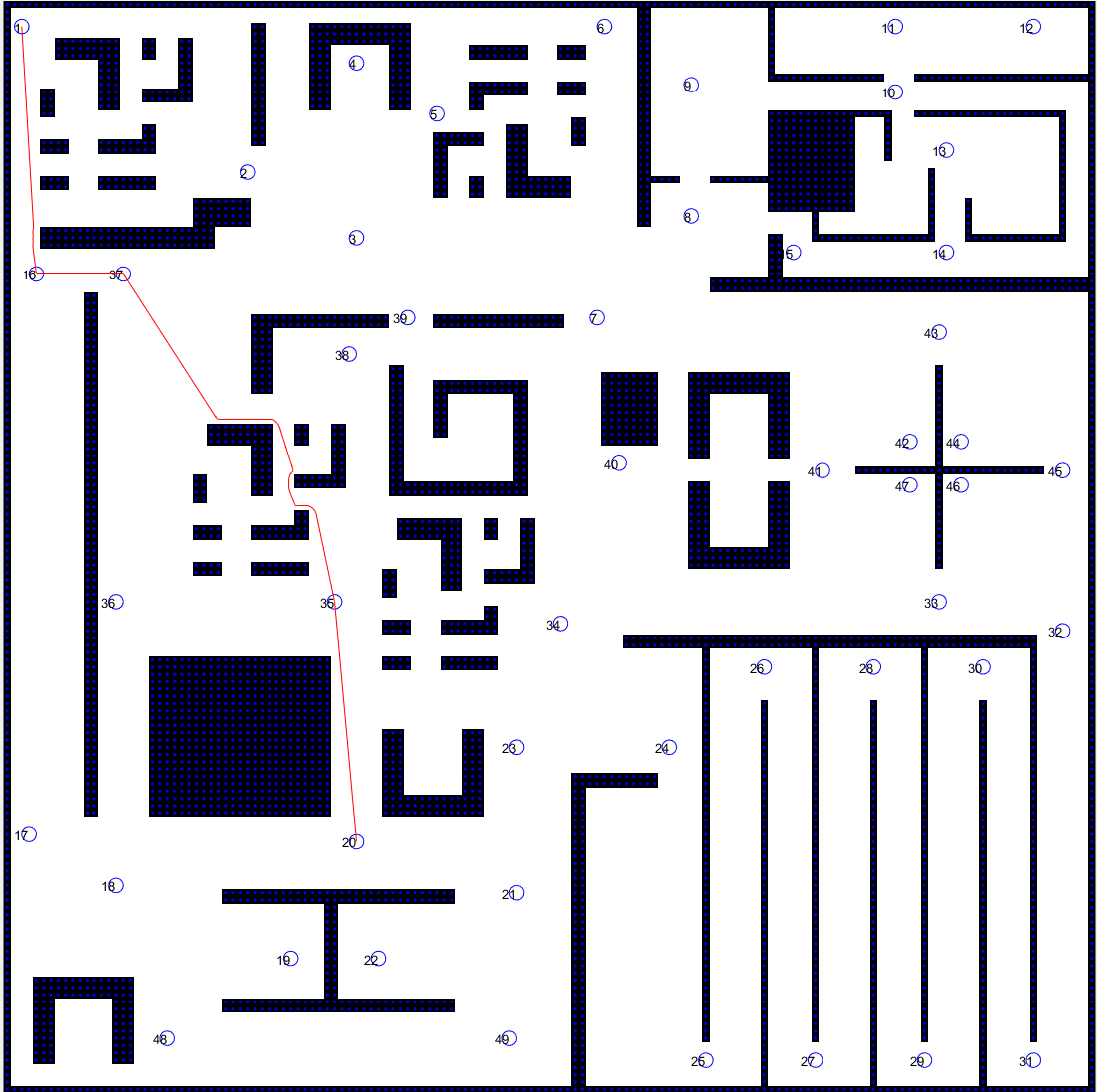


Figura 51: Resultado tras la ejecución del código del apartado 3.