# SÉPTIMA PRÁCTICA

# NAVEGACIÓN AUTÓNOMA

MIGUEL IAN GARCÍA POZO

GIERM

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE MÁLAGA

# Índice

# Implementación de la Navegación Autónoma

## - Desarrollo

Para esta práctica, queremos realizar un código que simule la navegación autónoma de un robot a lo largo de un mapa con obstáculos, para ello, vamos a juntar varias de las anteriores realizadas en esta asignatura. El mapa en el que vamos a navegar es el que vemos en la Figura 1.

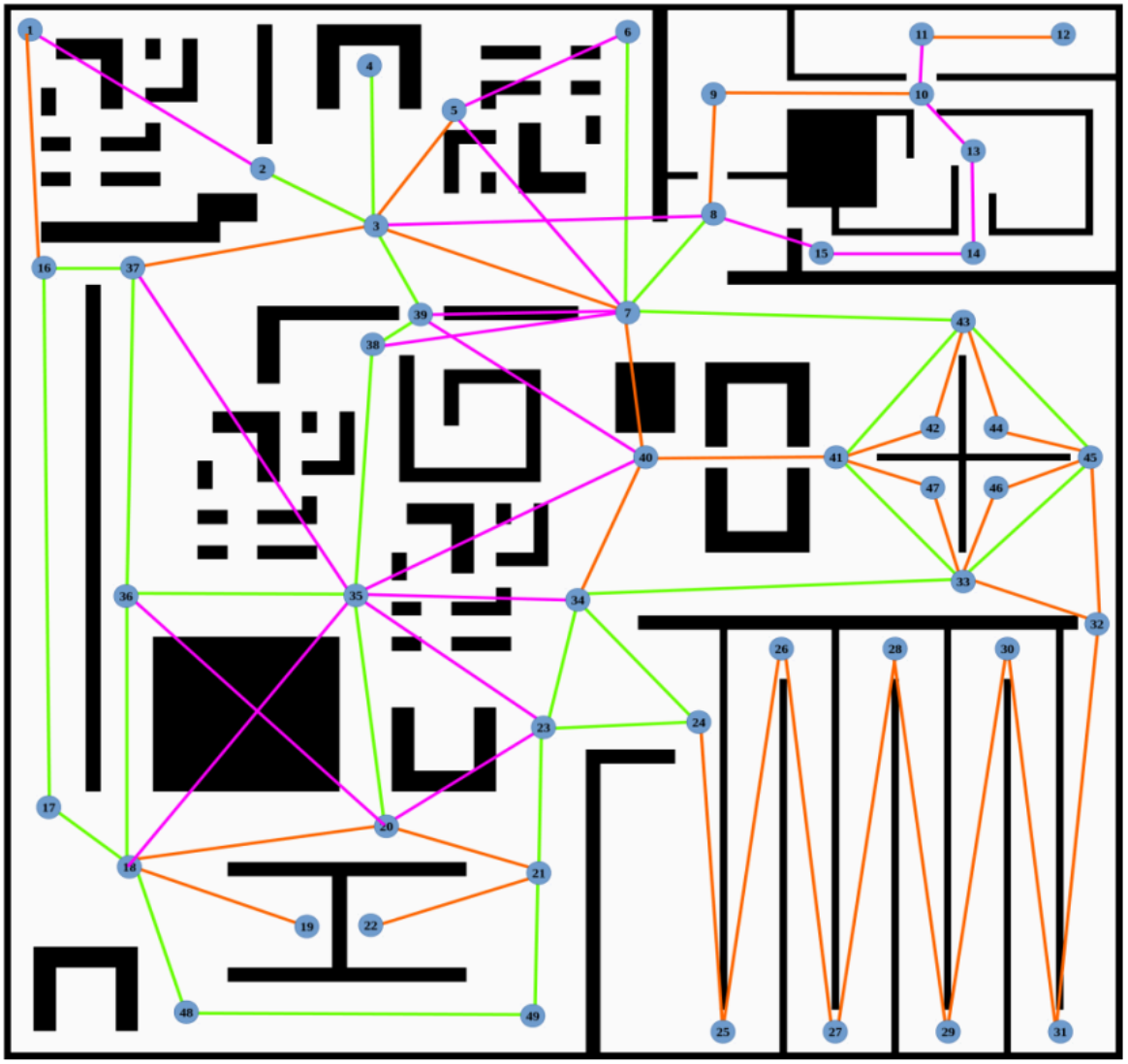



Figura 1.- Mapa en el que vamos  a hacer la navegación autónoma

Para navegar por este mapa, vamos a usar el algoritmo de Dijkstra visto en la práctica 5 para obtener el camino óptimo entre nuestro origen y nuestro destino y también vamos a usar la navegación reactiva por campos potenciales de la práctica 4. Por lo tanto, el código realizado lo podemos ver en las Figuras 2, 3, 4, 5, 6 y 7.

- Código "NavAuto.m"

```
function [coste,ruta] = NavAuto(nodos,g)

    % Pedimos los nodos de salida y destino al usuario
    s = input("Introduzca el nodo de salida: ");
    x = input("Introduzca el nodo de destino: ");

    [m,n] = size(g); % Obtenemos el tamaño del grafo de entrada
    caminos = Inf(1,m); % Definimos un vector de infinitos
    % Creamos una matriz con: columna de nodos, columna de costes y columna
    % de nodo anterior
    caminos = [1:m;caminos;zeros(1,m)]';
    % Cambiamos la primera fila por la del nodo inicio y cambiamos su coste
    % a cero
    caminos([1 s],:) = caminos([s 1],:);
    caminos(1,2) = 0;
    % Creamos la matriz donde vamos a guardar los nodos que ya hemos
    % revisado y la matriz donde pondremos la ruta solución
    revisado = zeros(m,3);
    solucion = zeros(m,3);
    % i es el nodo que estamos revisando y p es la posición en la matriz de
    % revisados donde vamos a poner el nodo que acabemos de ver
    i = s;
    p = 1;
while i ~= x
    % Creamos un bucle para recorrer las columnas de la matriz del
    % grafo
    for j = 1:length(g(1,:))
        % Este bucle nos devolverá la fila donde está ubicado el nodo
        % que buscamos en la matriz "caminos"
        for l = 1:length(caminos(:,1))
           if (caminos(l,1) == j)
                nodo = l;
           end
        end
        % Comprobamos si el elemento que estamos viendo es distinto de
        % cero y si el coste hasta el nodo es menor que el coste
        % anterior y guardamos la información en la matriz "caminos"
        if (g(i,j) ~= 0 )
            if (g(i,j)+caminos(1,2) < caminos(nodo,2))
                caminos(nodo,2) = g(i,j) + caminos(1,2);
                caminos(nodo,3) = i;
            end
        end
    end
    % Guardamos en "revisado" el nodo, volvemos a cero la columna del nodo en la matriz del grafo
    % eliminamos la primera fila para no volver a comprobarla y ordenamos la matriz
    g(:,caminos(1,1)) = 0;
    revisado(p,:) = caminos(1,:);
    p = p+1;
    caminos(1,:) = [];
    caminos = sortrows(caminos,2);
    i = caminos(1,1);
end
```

Figura 2.- Primera parte del código de la navegación autónoma

```
% Guardamos la fila del nodo destino
revisado(p,:) = caminos(1,:);
% Guardamos el nodo destino en la solución y creamos la variable con la
% que vamos a ver el nodo del que hemos llegado
solucion(1,:) = revisado(p,:);
ant = p;
% Este bucle se va a encargar de recorrer la matriz "revisado" y
% encontrar cuál es la ruta a seguir
for j = 2:length(revisado)
    for l = 1:length(revisado(:,1))
       if (revisado(l,1) == revisado(ant,3) && revisado(ant,3)~= 0)
            ant = l;
            solucion(j,:) = revisado(l,:);
       end
    end
end
% Elimino las filas de la matriz que siguen valiendo 0
solucion(sum(abs(solucion),2)==0,:)=[ ];
solucion(:,sum(abs(solucion),1)==0)=[ ];
% Defino el coste y la ruta, siendo esta la columna de los nodos de la
% solución invertida, para poder ver el recorrido desde la salida a la
% llegada
coste = solucion(1,2);
ruta = fliplr(solucion(:,1)');
```

Figura 3.- Segunda parte del código de navegación autónoma

Hasta aquí, el código lleva a cabo el algoritmo de Dijkstra. Cuando ya tenemos calculada la ruta, llamamos al código en el que tenemos la navegación reactiva, que se encargará de hacer todo el recorrido evitando cualquier obstáculo. Esto lo vemos en la Figura 4.

```
% i es el nodo en el que está el robot
i = 1;
recorrido = [1]; % Aquí guardaremos si el robot ha llegado a cada nodo del camino
% Carga del mapa de ocupacion
map_img=imread('mapa2.pgm');
map_neg=imcomplement(map_img);
map_bin=imbinarize(map_neg);
mapa=binaryOccupancyMap(map_bin);
show(mapa);

% Bucle que va a pintar en pantalla a la vez que simular el recorrido
% reactivo del robot
while (i <= (length(ruta)-1) && recorrido(i)~=0)
    % Llamamos a la función de la navegación reactiva y obtenemos si
    % llega o no al nodo
    recorrido = [recorrido EvitarObs(ruta(i),ruta(i+1),mapa,nodos)];
    i = i + 1;
end
% Una vez terminado comprobamos si el robot ha llegado al final o si se
% ha quedado atascado por el camino y no ha podido llegar.
if recorrido(i) ~= 1
    fprintf('No se ha podido llegar al destino.\n')
else
    fprintf('Destino alcanzado.\n')
end
end
```

Figura 4.- Tercera parte del código de navegación autónoma

- Código “EvitarObs.m”

```
function alcanzable = EvitarObs(ni,nf,mapa,nodos)

    % Configuracion del sensor (laser de barrido)
    max_rango=10;
    angulos=-pi/2:(pi/180):pi/2; % resolucion angular barrido laser

    % Caracteristicas del vehiculo y parametros del metodo
    v=0.4;            % Velocidad del robot
    D=1.5;           % Rango del efecto del campo de repulsión de los obstáculos
    alfa=2;           % Coeficiente de la componente de atracción
    beta=10;      % Coeficiente de la componente de repulsión

    % Marcar los puntos de inicio y destino
    hold on;
    title('Señala los puntos inicial y final de la trayectoria del robot');
    origen=nodos(ni,2:3);
    plot(origen(1), origen(2), 'go','MarkerFaceColor','green');  % Dibujamos el origen
    destino=nodos(nf,2:3);
    plot(destino(1), destino(2), 'ro','MarkerFaceColor','red');  % Dibujamos el destino

    % Inicialización
    robot=[origen 0];     % El robot empieza en la posición de origen (orientacion cero)
    path = [];
    path = [path; robot]; % Se añade al camino la posicion actual del robot
    iteracion=0;              % Se controla el nº de iteraciones por si se entra en un minimo local

% Calculo de la trayectoria
while norm(destino-robot(1:2)) > v && iteracion<1000    % Hasta menos de una iteración de la meta (10 cm)
    % Definimos a 0 el vector con la fuerza de repulsión
    Frep = [0 0];
    % Calculamos con la ecuación el valor de la fuerza de atracción
    Fatr = alfa.*(destino - robot(1:2));
    % Usamos la función que nos devolverá la distancia a los obstáculos cercanos
    obs = Lidar(robot,mapa,angulos,max_rango);
    % Comprobamos si tenemos un objeto cercano recorriendo la matriz "obs"
    for i = 1:length(obs)
        % Comprobamos que el valor que estamos viendo no es nan, que indica
        % fuera de rango
       if (not(isnan(obs(i,1))))
            % calculamos la distancia entre el robot y el punto del
            % obstáculo
            rho = (robot(1:2)-obs(i,:));
            d = sqrt(rho(1)^2+rho(2)^2);
            % Comprobamos si la distancia es menor a la umbral definida
            % anteriormente
            if (d<=D)
                % Sumamos a la fuerza de repulsión acumulada, la fuerza de repulsión que
                % genera el punto que estamos comprobando
                Frep = Frep + beta*((1/d)-(1/D))*((robot(1:2)-obs(i,:))/d^3);
            end
       end
    end
    % Calculamos la Fuerza resultante
    Fsol = Fatr + Frep;
    % Hacemos unitario el vector fuerza para que no influya en la velocidad
    unitario = Fsol/sqrt(Fsol(1)^2+Fsol(2)^2);
    % Definimos la nueva posición del robot tras moverse
    robot = [robot(1)+unitario(1)*v robot(2)+unitario(2)*v atan2(unitario(2),unitario(1))];
    path = [path;robot];    % Se añade la nueva posición al camino seguido
    plot(path(:,1),path(:,2),'r'); % Pintamos el movimiento
    drawnow
    iteracion=iteracion+1;
end
```

Figura 5.- Primera parte del código de evitar obstáculos

```
    if iteracion==1000   % Se ha caído en un mínimo local
        alcanzable = 0;
    else
        alcanzable = 1;
    end
end
```

Figura 6.- Segunda parte del código de evitar obstáculos

Como podemos ver, este código es el de la navegación por campos potenciales, con la diferencia de que las últimas cuatro líneas devuelven la confirmación de haber llegado o no al nodo siguiente como un 1 o un 0.

- Código "Lidar.m"

Es el código que simula al sensor Lidar.

```
%% funcion para simular el sensor
function [obs]=Lidar(robot, mapa, angulos, max_rango)
    obs=rayIntersection(mapa,robot,angulos, max_rango);
end
```

Figura 7.- Función que simula el sensor LIDAR

Teniendo ya claro todo el código y su funcionamiento, podemos empezar a hacer simulaciones.

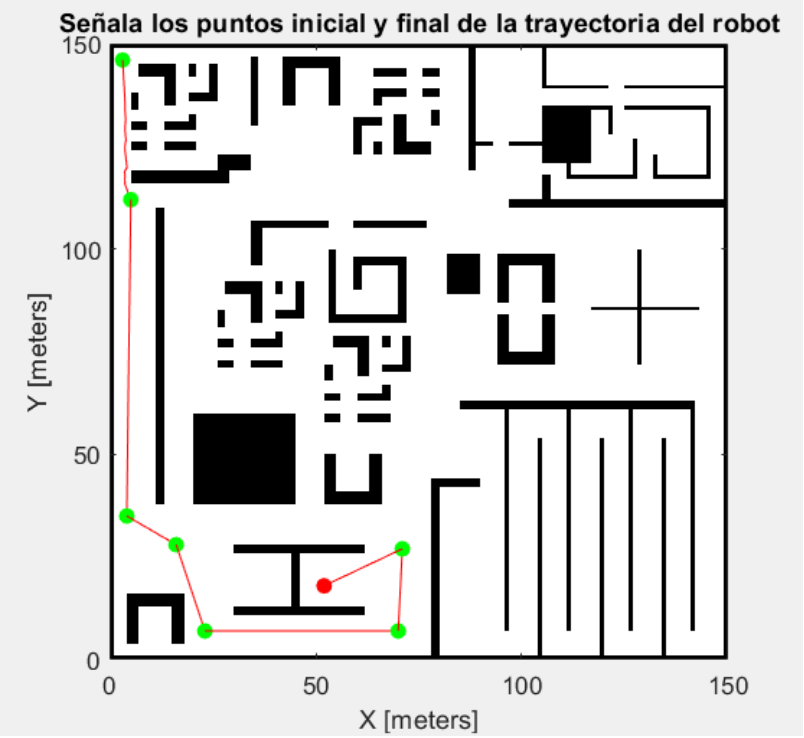
## - Simulación

Para realizar la simulación, debemos ejecutar el código "mapa2.m" que contiene las matrices de las coordenadas de los nodos y de los costes que hay entre ellos. Después de esto, ya solo queda introducir el comando de la Figura 8 e introducir después los nodos inicial y final.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 22
```

Figura 8.- Comando para dar comienzo a la simulación

El comando de la Figura 8 ejecuta el programa y abre una ventana en la que se ve como el robot va evitando obstáculos pasando por cada uno de los nodos del camino óptimo a seguir. Veamos cuál es el resultado de esta simulación en la Figura 9.

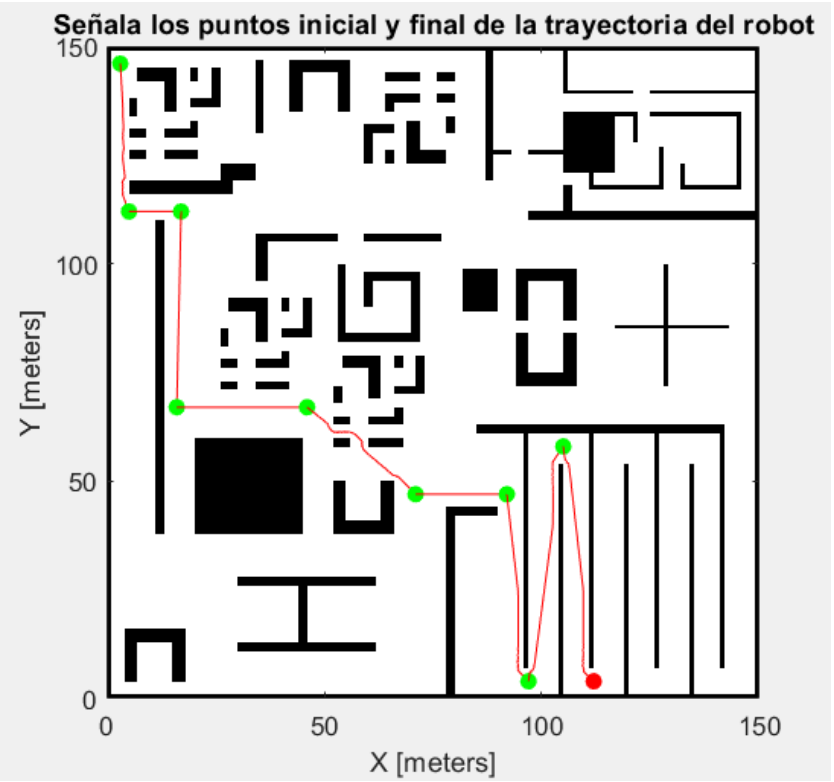



```
Destino alcanzado.

coste =

     9


ruta =

     1    16    17    18    48    49    21    22
```

Figura 9.- Simulación del camino desde el nodo 1 al 22

Vemos que la línea roja en la Figura 9 es el camino que sigue el robot mientras que los puntos son los nodos por los que ha ido pasando. Probemos con otro caso más largo en la Figura 10.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 27
```




```
Destino alcanzado.

coste =

    15


ruta =

     1    16    37    36    35    23    24    25    26    27
```

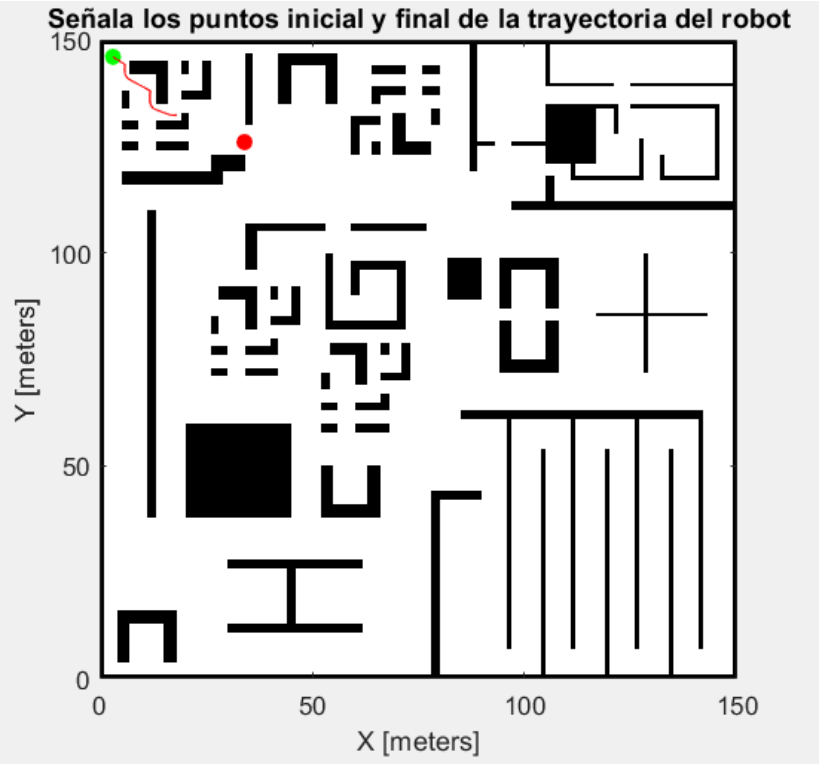
Figura 10.- Simulación del robot yendo desde el nodo 1 al 27

Comprobamos así que nuestro programa de navegación autónoma funciona correctamente, aunque no siempre puede llegar al destino.

# Nodos no alcanzables

Si seguimos probando distintos casos de nodos inicial y final, nos damos cuenta que entre el nodo 1 y la parte de la derecha arriba no se pueden comunicar, ya que los obstáculos tienen la forma idónea para crear mínimos locales en los que nuestro robot se queda atrapado. Veámoslo en la Figura 11.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 12
```




```
No se ha podido llegar al destino.

coste =

    16


ruta =

     1     2     3     8     9    10    11    12
```

Figura 11.- Simulación para obligar al robot a ir hacia la parte derecha superior del mapa

En la Figura 11 vemos como, claramente, el robot no puede ni llegar del nodo 1 al 2 debido a ese mínimo local que se forma entre ellos dos.

Para mejorar estos casos, se me ocurre utilizar una fuerza más en el cálculo de la fuerza resultante, que va a ser tangente a la fuerza de repulsión. De esta manera, el robot al caer en un mínimo local en el que la fuerza de atracción y de repulsión se contrarrestan, se movería bordeando el obstáculo hasta poder salir de este mínimo local.

El código de evitar obstáculos es prácticamente el mismo con las líneas de la Figura 12 como única diferencia.

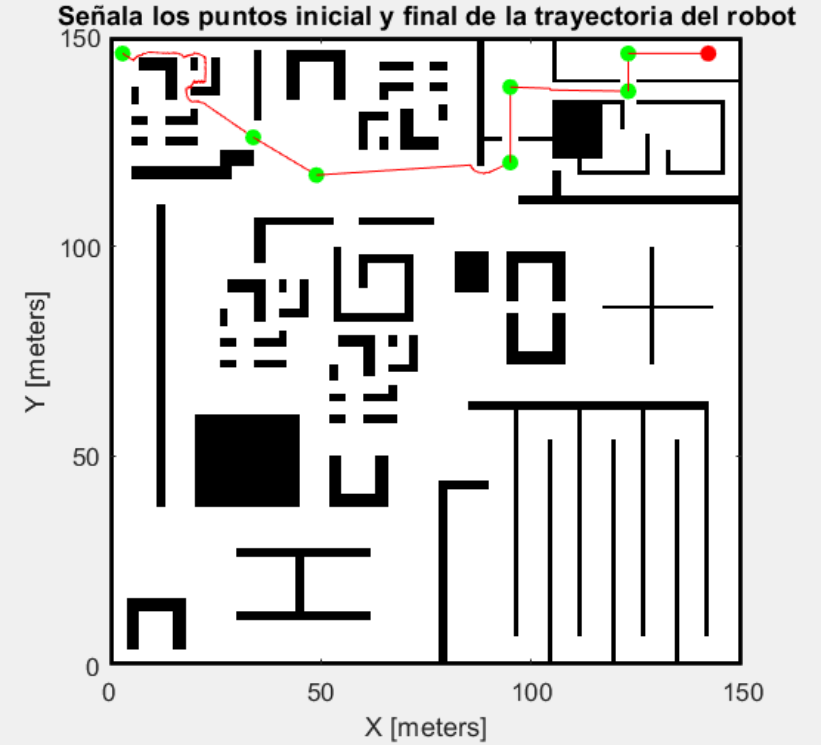
```
% Creo el vector de la fuerza tangente a la de repulsión como su
% vector normal. El vector normal de una recta Ax+By+C=0 es [A B]
% mientras que el vector director es [-B A], por lo tanto, el
% vector normal positivo a Frep será [Frep(2) -Frep(1)] y el
% negativo será el opuesto.
% Calculando el sector por el que nos vamos a mover desde el origen
% al destino, decidimos una dirección de la fuerza tangente
if (sector<=pi/2 && sector>=0)
    Ftg=-[Frep(2) -Frep(1)]; % Vector normal negativo a Frep
elseif(sector>=-pi/2 && sector<=0)
    Ftg=[1.5*Frep(2) -Frep(1)]; % Vector normal positivo a Frep
elseif(sector<=-pi/2)
    Ftg=-[Frep(2) -Frep(1)]; % Vector normal negativo a Frep
else
    Ftg=[Frep(2) -Frep(1)]; % Vector normal positivo a Frep
end
```

```
% Calculamos la Fuerza resultante
Fsol = Fatr + Frep;
% Comprobamos en que sentido queremos que nos afecte la fuerza
% tangente
if (cambio == 1)
    Ftg = -Ftg;
    % Comprobamos si nos estamos quedando en un mínimo local, ya
    % que la fuerza de atracción no variará para cambiar el sentido
    % de la fuerza tangente
    if (iteracion-ite >= 100)
        if (norm(Fatr) >= norm(ant(iteracion-50,:))-0.1 && norm(Fatr) <= norm(ant(iteracion-50,:))+0.1)
            cambio = 0;
            Ftg = -Ftg;
        end
    end
else
    % Comprobamos si nos estamos quedando en un mínimo local, ya
    % que la fuerza de atracción no variará para cambiar el sentido
    % de la fuerza tangente
    if (iteracion>50)
        ite = iteracion;
        if (norm(Fatr) >= norm(ant(iteracion-50,:))-0.1 && norm(Fatr) <= norm(ant(iteracion-50,:))+0.1)
            Ftg = -Ftg;
            cambio = 1;
        end
    end
end
% Guardamos la fuerza tangente anterior para compararla con la
% siguiente
ant = [ant; Fatr];
% Calculamos la fuerza resultante con la fuerza tangente
Fsol = Fsol + Ftg;
```

Figura 12.- Código con el que conseguimos salir de los mínimos locales

Como viene explicado en los comentarios del código de la Figura 12, creamos una fuerza tangente a la fuerza de repulsión con la que modificamos la dirección en la que se va a mover el robot. Para saber en qué dirección debe actuar esta fuerza tangente, comprobamos si nos estamos alejando demasiado del destino, gracias a la fuerza de atracción. Pongamos en práctica este código con el mismo ejemplo anterior en la Figura 13.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 12
```

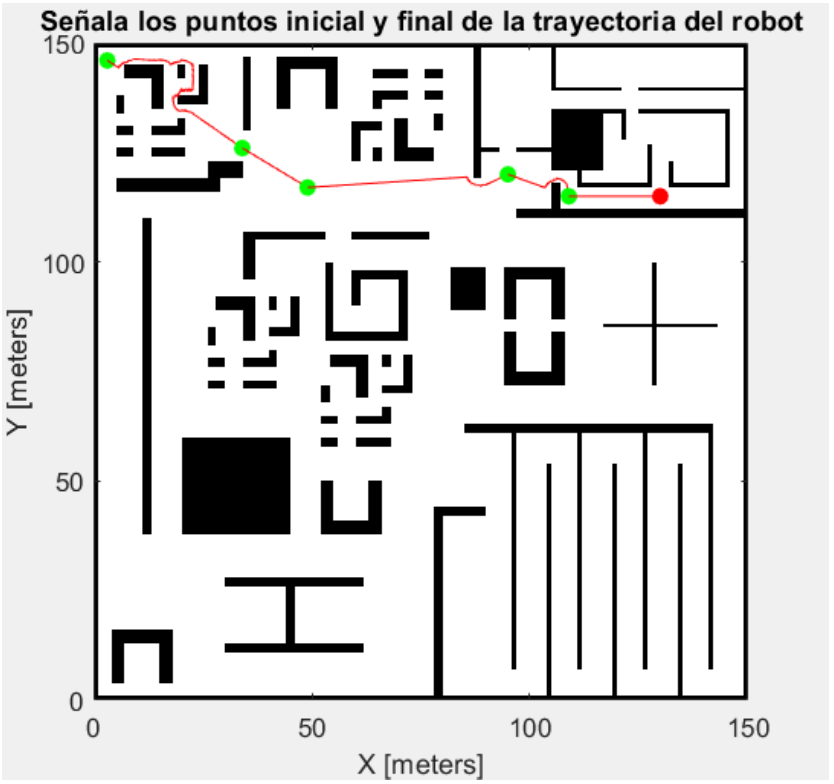



```
Destino alcanzado.

coste =

    16


ruta =

     1     2     3     8     9    10    11    12
```

Figura 13.- Simulación con la que se consigue salir de los mínimos locales

Vemos como, en la Figura 13, el robot es capaz de superar los mínimos locales con facilidad gracias a esta nueva fuerza tangente. Probemos ahora yendo al nodo 14 en la Figura 14.

```
>> [coste,ruta] = NavAuto(nodos,costes
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 14
```




```
Destino alcanzado.

coste =

    13


ruta =

     1     2     3     8    15    14
```

Figura 14.- Simulación desde el nodo 1 al 14 saliendo de los mínimos locales

En la Figura 14, podemos ver que el robot llega a su destino con cierta facilidad. Probemos ahora a movernos por la parte inferior derecha del mapa en la Figura 15.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 27
Destino alcanzado.

coste =

    15


ruta =

     1    16    37    36    35    23    24    25    26    27
```

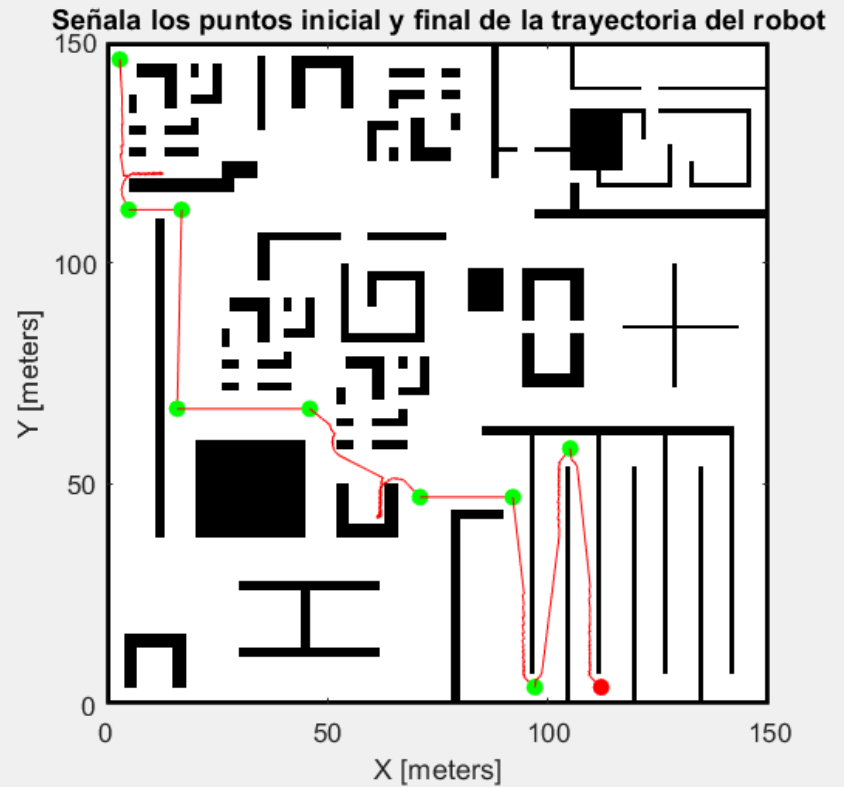



Figura 15.- Simulación desde el nodo 1 al 27

Aquí podemos ver como el robot, en algunos casos, se mueve en una dirección que lo desvía de su camino lógico, pero se acaba dando cuenta de esto, por lo que cambia la dirección de la fuerza tangente y consigue continuar el camino. Hagamos otra prueba en la Figura 16.

```
>> [coste,ruta] = NavAuto(nodos,costes)
Introduzca el nodo de salida: 1
Introduzca el nodo de destino: 30
Destino alcanzado.

coste =

    14


ruta =

     1     2     3     7    43    45    32    31    30
```

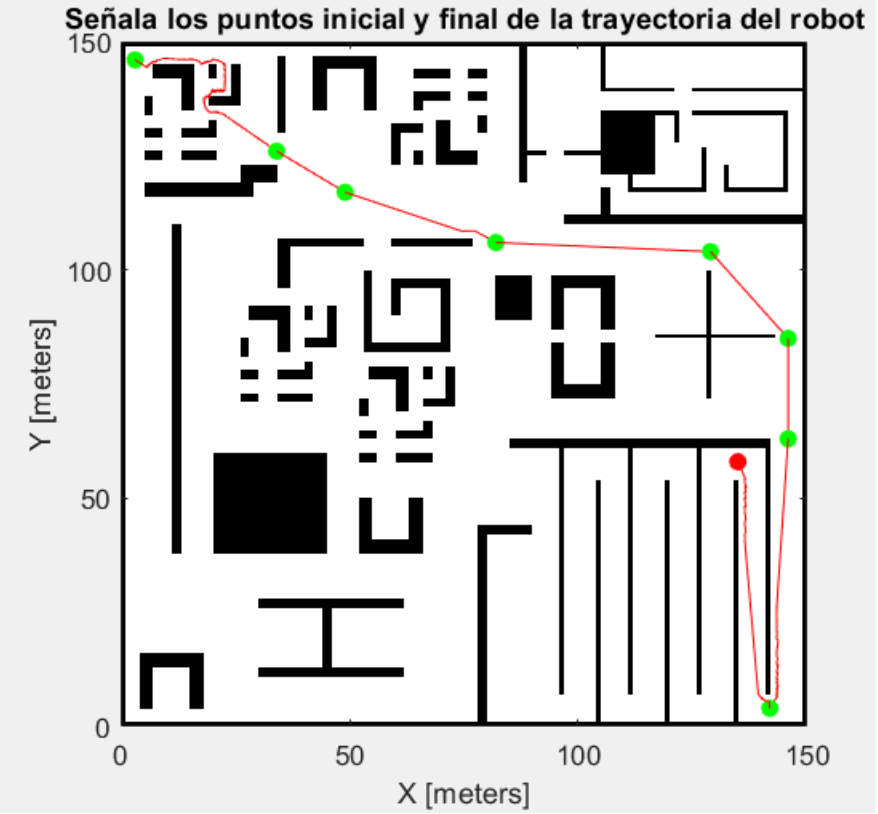



Figura 16.- Simulación desde el nodo 1 al 30

Con todas estas comprobaciones, más o menos podríamos decir que este método resuelve el problema, ya que no es del todo preciso y a veces se desvía del camino lógico y sencillo, pero al menos, consigue resolver ese problema que teníamos al principio.

# Implementación del A*

- Matriz de la Heurística

Para definir la matriz de heurística podemos optar por un par de opciones, o elegimos la heurística en función de la distancia euclídea, por lo que también deberíamos cambiar la matriz de costes además de calcular todos los caminos a todos los nodos, o, podemos simplemente usar la matriz de costes que ya existe para calcular todos los caminos óptimos hasta cada uno de los nodos. Vamos a optar por la primera opción ya que nos ahorramos el tener que usar el algoritmo de dijkstra para cada uno de los nodos hasta cada uno de los destinos.

Para calcular la heurística y los costes según la distancia euclídea entre los nodos simplemente crearemos un bucle for en el que rellenaremos cada una de las filas de la nueva matriz de costes y la matriz de heurística. Dentro de este bucle, crearemos otro por el que rellenaremos cada una de las columnas y ya simplemente nos queda calcular las distancias euclídeas entre las coordenadas de cada uno de los nodos. El código entonces quedaría como lo vemos en la Figura 17.

```
for i = 1:49
    for j = 1:49
        if(j==i)
            H(i,j) = 0;
        else
            % Calculamos distancia euclídea entre el nodo i y el j y le
            % restamos un número aleatorio entre 1 y 3 para que la matriz
            % de heurística sea mejor
            H(i,j) = sqrt((nodos(i,2)-nodos(j,2))^2+(nodos(i,3)-nodos(j,3))^2)-randi(3);
        end
        % Comprobamos si los nodos están o no conectados y calculamos sus
        % distancias euclídeas
        if (costes(i,j) ~= 0)
            costes(i,j) = sqrt((nodos(i,2)-nodos(j,2))^2+(nodos(i,3)-nodos(j,3))^2);
        end
    end
end
```

Figura 17.- Código para calcular la heurística y los costes según la distancia euclídea

Con este código, creamos la matriz de heurística y modificamos la de costes, por lo que podemos ejecutar entonces el código de navegación con A*.

Antes de simular, debemos decir que la única modificación en el código de navegación autónoma fue cambiar el algoritmo de Dijkstra por el de A* realizado en la práctica 6. Esto lo podemos ver en las Figuras 18 y 19.

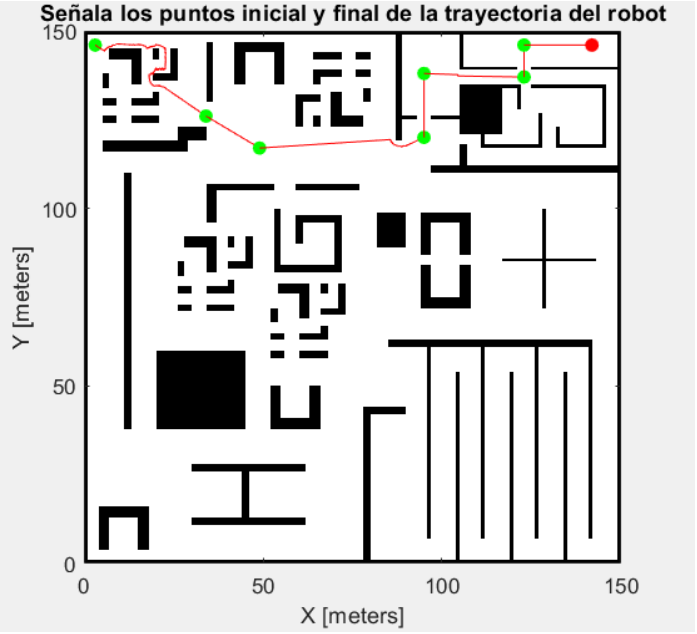
```
function [coste,ruta] = NavAutoAest(nodos,g,h)

    % Pedimos los nodos de salida y destino al usuario
    s = input("Introduzca el nodo de salida: ");
    x = input("Introduzca el nodo de destino: ");

    [m,n] = size(g); % Obtenemos el tamaño del grafo de entrada

    caminos = inf(2,m); % Definimos un vector de infinitos
    % Creamos una matriz con: columna de nodos, columna de costes, columna de costes con la heurística
    % y columna de nodo anterior
    caminos = [uint32(1):uint32(m);caminos;zeros(1,m)]';
    % Cambiamos la primera fila por la del nodo inicio y cambiamos su coste
    % a cero
    caminos([1 s],:) = caminos([s 1],:);
    caminos(1,2) = 0;
    caminos(1,3) = h(x,s);

    % Creamos la matriz donde vamos a guardar los nodos que ya hemos
    % revisado y la matriz donde pondremos la ruta solución
    revisado = zeros(m,4);
    solucion = zeros(m,4);

    % i es el nodo que estamos revisando y p es la posición en la matriz de
    % revisados donde vamos a poner el nodo que acabemos de ver
    i = s;
    p = 1;

            while i ~= x
                % Creamos un bucle para recorrer las columnas de la matriz del
                % grafo
                for j = 1:length(g(1,:))
                    % Este bucle nos devolverá la fila donde está ubicado el nodo
                    % que buscamos en la matriz "caminos"
                    for l = 1:length(caminos(:,1))
                       if (caminos(l,1) == j)
                            nodo = l;
                       end
                    end
                    % Comprobamos si el elemento que estamos viendo es distinto de
                    % cero y si el coste hasta el nodo es menor que el coste
                    % anterior y guardamos la información en la matriz "caminos"
                    if (g(i,j) ~= 0 )
                        if (g(i,j)+caminos(1,2) < caminos(nodo,2))
                            caminos(nodo,2) = g(i,j) + caminos(1,2);
                            % Guardamos el coste con la heurística
                            caminos(nodo,3) = g(i,j) + caminos(1,2) + h(x,j);
                            caminos(nodo,4) = i;
                        end
                    end
                end
                % Ordenamos la matriz, guardamos en "revisado" el nodo y eliminamos
                % la primera fila para no volver a comprobarlo.
                g(:,caminos(1,1)) = 0;
                revisado(p,:) = caminos(1,:);
                p = p+1;
                caminos(1,:) = [];
                % Reordenamos la matriz en función del coste con heurística menor
                caminos = sortrows(caminos,3);
                i = caminos(1,1);
            end
```

Figura 18.- Primera parte del código con el algoritmo A*

```
revisado(m,:) = caminos(1,:);

solucion(1,:) = revisado(m,:);
ant = m;
% Este bucle se va a encargar de recorrer la matriz "revisado" y
% encontrar cuál es la ruta a seguir
for j = 2:length(revisado)
    for l = 1:length(revisado(:,1))
       if (revisado(l,1) == revisado(ant,4) && revisado(ant,4)~= 0)
            ant = l;
            solucion(j,:) = revisado(l,:);
       end
    end
end

% Elimino las filas de la matriz que siguen valiendo 0
solucion(sum(abs(solucion),2)==0,:)=[ ];
solucion(:,sum(abs(solucion),1)==0)=[ ];

% Defino el coste y la ruta, siendo esta la columna de los nodos de la
% solución invertida, para poder ver el recorrido desde la salida a la
% llegada
coste = solucion(1,2);
ruta = fliplr(solucion(:,1)');
```

Figura 19.- Segunda parte del código con el algoritmo A*

El resto del código es idéntico al del apartado anterior. Probemos entonces con el algoritmo de planificación de caminos A* a navegar de manera autónoma por el mapa. Lo vemos en la Figura 20.




```
Destino alcanzado.

coste =

   174


ruta =

     1     2     3     8     9    10    11    12
```

Figura 20.- Simulación usando el algoritmo A* con la matriz heurística calculada anteriormente

En la Figura 20 vemos que el camino seguido es idéntico al del apartado anterior, esto se debe a que la modificación que hemos llevado a cabo solo cambia la manera de calcular la ruta, pero no cómo llevarla a cabo.

# Enlace a Github

https://github.com/MiguelIlan/AmpliacionRobotica/tree/main/Rob%C3%B3tica%20m%C3%B3vil/NavegacionAutonoma