Universidad de Alcalá

Práctica 2

Navegación local y global

1. INTRODUCCIÓN

En esta práctica se desarrollarán todos los conceptos de navegación local y global estudiados a lo largo del Tema 3 de la asignatura.

En primer lugar se estudiarán los planificadores locales, también conocidos como evitadores de obstáculos, con el fin de dotar a la plataforma de una navegación local segura. Para ello se estudiará el planificador local propuesto por Borenstein y conocido como VFH (Vector Field Histogram), que tiene implementaciones tanto en la robotics toolbox de Matlab como en ROS.

Una vez que se disponga de un navegador local seguro, se propondrá emplear el navegador global propuesto por Kavraki y conocido como PRM (Probabilistic RoadMap) disponible en la robotics toolbox de Matlab, haciendo uso en paralelo del VFH de tal forma que se alcance un destino global garantizando la integridad tanto del robot como del entorno que le rodea.

El punto de partida de la práctica serán los mapas obtenido en la práctica 1 de la asignatura (Figuras 1 y 2) empleando técnicas de SLAM, tanto para el entorno simulado como para el entorno real del laboratorio.

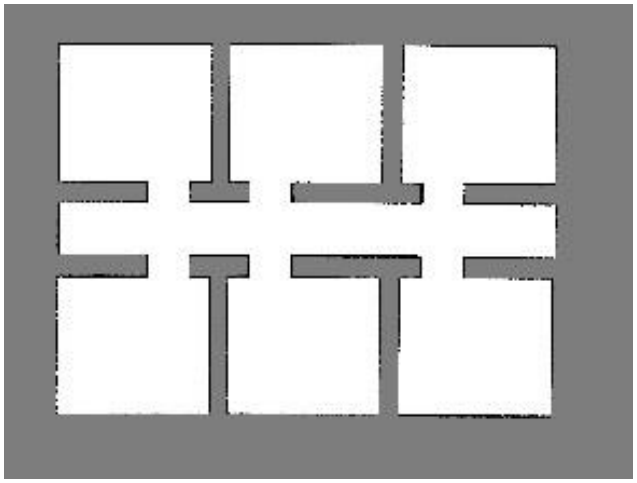

Figura 1. Mapa del entorno simulado "simple_rooms"

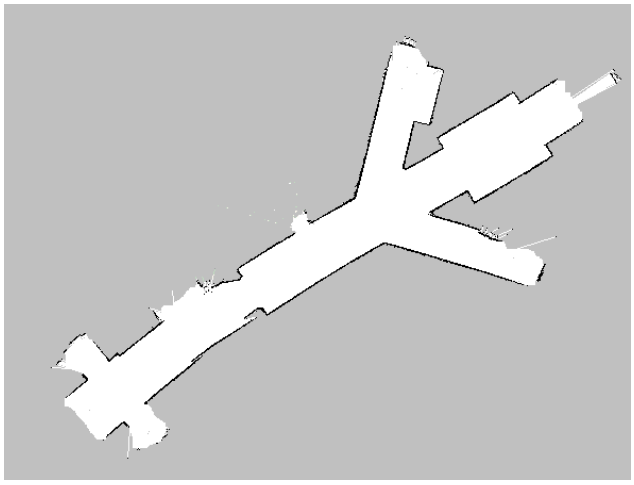

Figura 2. Mapa real de los laboratorios

2. PLANIFICACIÓN LOCAL (VFH)

VFH es el algoritmo que propuso Borenstein en el año 1991, en él se reducen las casillas de ocupación bidimensionales a un histograma polar de 1D (Figura 3), que hace más fácil seleccionar la dirección óptima a seguir ante la detección de un obstáculo que no estaba contemplado en el mapa.

La navigation toolbox de Matlab proporciona el algoritmo VFH a través de la clase "*controllerVFH*", además de un conjunto de propiedades/parámetros que pueden configurar el funcionamiento del mismo.

En esta sección se proponen diferentes pruebas para conseguir configurar los parámetros del algoritmo en modo simulado y una vez conseguidos los mejores parámetros se validarán en modo real. Para ello, se empezará estudiando cómo utilizar el evitador.

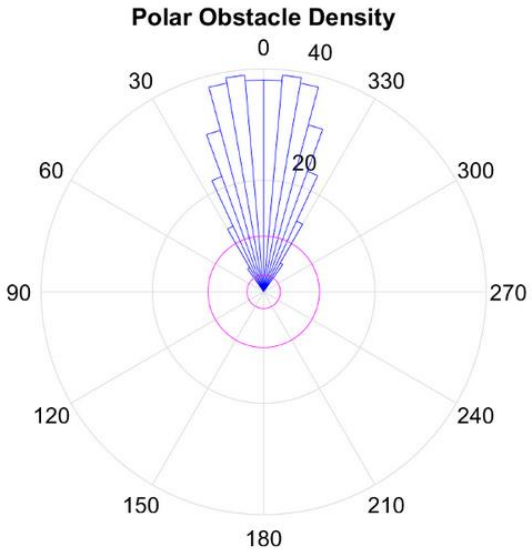

Figura 3. Histograma polar VFH

2.1. ESTUDIO DEL ALGORITMO

Para comenzar a trabajar con el evitador de obstáculos, se pide:

1. Describir el funcionamiento del algoritmo VFH disponible en la robotics toolbox de Matlab.
2. Describir los parámetros que se pueden configurar en dicho algoritmo.
3. ¿Cuáles son los parámetros de ajuste de la función de coste?
4. ¿Qué método está disponible para ayudar en la depuración del funcionamiento del algoritmo?

La información necesaria puede encontrarse en las siguientes páginas:

https://www.mathworks.com/help/nav/ug/vector-field-histograms.html
https://www.mathworks.com/help/nav/ug/obstacle-avoidance-with-turtlebot-and-vfh.html

2.2. PRUEBAS EN SIMULACIÓN

2.2.1. Comportamiento tipo wander. Script wander_vfh.m

Para probar el funcionamiento del evitador se va a implementar un comportamiento de tipo *Wander* (deambulación), que consistirá en que el robot avance en línea recta mientras le sea posible, y en caso de detectar algún obstáculo en dicha dirección la modifique hacia una nueva zona libre. Este comportamiento puede obtenerse utilizando el evitador VFH con dirección de destino *targetDir*=0, y realizando un control de la velocidad angular proporcional a la dirección seleccionada por el algoritmo.

Para ello se realizará un script llamado **wander_vfh.m** con la siguiente estructura (se deben rellenar los apartados mostrados en rojo/negrita):

```
%Crear figuras distintas para el láser y el visualizador del VFH
% NOTA: para dibujar sobre una figura concreta, antes de llamar a la
% correspondiente función de dibujo  debe convertirse en la figura activa
% utilizando  figure(fig_laser) o figure(fig_vfh) respectivamente.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
close all
fig_laser=figure;  title('LASER')
fig_vfh=figure;  title('VFH')
```

```
%Crear el objeto VFH…
%%%%%%%%%%%%%%%%%%%%%%
VFH=controllerVFH;
%y ajustamos sus propiedades
%%%%%%%%%%%%%%%%%%%%%%%%%%
VFH.NumAngularSectors=;
VFH.DistanceLimits=;
VFH.RobotRadius=;
VFH.SafetyDistance=;
VFH.MinTurningRadius=;
VFH.TargetDirectionWeight=;
VFH.CurrentDirectionWeight=;
VFH.PreviousDirectionWeight=;
VFH.HistogramThresholds=;
VFH.UseLidarScan=true;  %para permitir utilizar la notación del scan

%Rellenamos los campos por defecto de la velocidad del robot, para que la lineal
%sea siempre 0.1 m/s


%Bucle de control infinito
while(1)

    %Leer y dibujar los datos del láser en la figura 'fig_laser'


    %Llamar al objeto VFH para obtener la dirección a seguir por el robot para
    %evitar los obstáculos. Mostrar los resultados del algoritmo (histogramas)
    %en la figura 'fig_vfh'


    %Rellenar el campo de la velocidad angular del mensaje de velocidad con un
    %valor proporcional a la dirección anterior (K=0.1)


    %Publicar el mensaje de velocidad


    %Esperar al siguiente periodo de muestreo

end
```

Para depurar el comportamiento del algoritmo, realizar las siguientes pruebas:

1. Ajustar el parámetro "*NumAngularSectors*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
2. Ajustar el parámetro "*DistanceLimits*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
3. Ajustar el parámetro "*RobotRadius*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
4. Ajustar el parámetro "*SafetyDistance*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.

5. Ajustar el parámetro "*MinTurningRadius*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
6. Ajustar el parámetro "*TargetDirectionWeight*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
7. Ajustar el parámetro "*CurrentDirectionWeight*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
8. Ajustar el parámetro "*PreviousDirectionWeight*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
9. Ajustar el parámetro "*HistogramThresholds*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo. Registrar el recorrido del robot y representarlo haciendo uso de la función plot de Matlab.
10. Ajustar todos los parámetros a la configuración óptima seleccionada.

2.2.2. Comportamiento wander y localización. Script wander_localiza.m

Una vez depurado el comportamiento, se va a proceder a incorporar el localizador AMCL al bucle de control (ya realizado en la práctica 1), de manera que el robot pueda irse localizando a medida que deambula por el entorno.

Se recomienda inicializar el localizador utilizando una distribución gaussiana de las partículas entorno a la posición real del robot (por ejemplo, en el caso del entorno 'simple_rooms', distribuir las partículas en la habitación en la que se sabe que está inicialmente el robot). De esta manera, podemos colocar el robot en una zona conocida amplia y el localizador se encargará de obtener con precisión la localización del robot a medida que este se va moviendo por el entorno.

Por otro lado, se deberán testear los valores de la matriz de covarianza en la estimación de la posición de manera que, cuando éstos se encuentren por debajo de un umbral (a ajustar experimentalmente), se considere al robot correctamente localizado, momento en que finalizará el programa.

Para ello, se va a programar un script de nombre **wander_localiza.m** (a partir de el ya realizado "wander.m), con la siguiente estructura (se deben rellenar los apartados mostrados en rojo/negrita):

```
%Cargar el mapa
%%%%%%%%%%%%%%%

%Crear el objeto VFH…y ajustar sus propiedades
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Inicializar el localizador AMCL (práctica 1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Rellenamos los campos por defecto de la velocidad del robot, para que la lineal
%sea siempre 0.1 m/s

%Bucle de control infinito
while(1)

    %Leer y dibujar los datos del láser en la figura 'fig_laser'

    %Leer la odometría

    %Obtener la posición pose=[x,y,yaw] a partir de la odometría anterior

    %Ejecutar amcl para obtener la posición estimada estimatedPose y la
    %covarianza estimatedCovariance (mostrar la última por pantalla para
```

```
    %facilitar la búsqueda de un umbral)

    %Si la covarianza está por debajo de un umbral, el robot está localizado y
    %finaliza el programa
    if (estimatedCovariance(1,1)<umbralx && estimatedCovariance(2,2)<umbraly &&
estimatedCovariance(3,3)<umbralyaw)
            disp('Robot Localizado');
            break;
    end

    %Dibujar los resultados del localizador con el visualizationHelper


    %Llamar al objeto VFH para obtener la dirección a seguir por el robot para
    %evitar los obstáculos. Mostrar los resultados del algoritmo (histogramas)
    %en la figura 'fig_vfh'


    %Rellenar el campo de la velocidad angular del mensaje de velocidad con un
    %valor proporcional a la dirección anterior (K=0.1)


    %Publicar el mensaje de velocidad


    %Esperar al siguiente periodo de muestreo

end
```

2.3. PRUEBAS CON EL ROBOT REAL

Una vez validados los parámetros de ajuste del comportamiento del algoritmo, se pide:

1. Emplear la mejor configuración de parámetros obtenida en el último apartado de la sección de simulación para comprobar el funcionamiento en modo real.
2. ¿Es suficientemente realista la simulación como para validar dichos parámetros en modo real?
3. En caso de no ser suficiente, ajustar los parámetros en modo real.

3. PLANIFICACIÓN GLOBAL (PRM)

En este apartado se empleará el planificador global conocido como PRM (Probabilistic RoadMap) propuesto por Kavraki en 1996. En él se propone el uso de nodos conectados y que son distribuidos por el entorno de forma aleatoria y se comprueba la conectividad (cuando no son bloqueados por un obstáculo) con los demás nodos para saber si se establece la conexión o no.

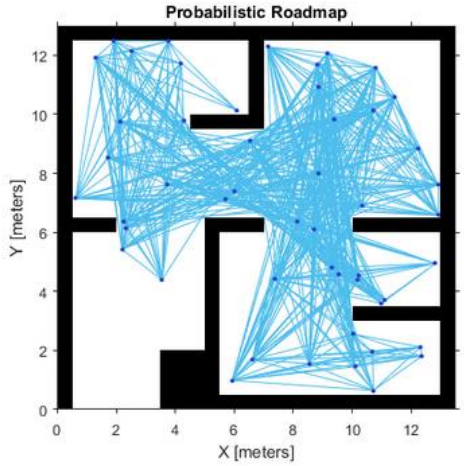

Figura 3. Ejemplo de planificación con PRM.

Empleando el mismo entorno que se ha empleado para el apartado anterior, se propone realizar el estudio del planificador PRM y se pide ajustar el mismo en modo simulación para poder validarlo finalmente en modo real.

3.1. ESTUDIO DEL ALGORITMO

Para comenzar a trabajar con el planificador global, se pide:

1. Describir el funcionamiento del algoritmo PRM disponible en la robotics toolbox de Matlab.
2. Describir los parámetros que se pueden configurar en dicho algoritmo.
3. ¿Cuáles son los métodos que sirven para actualizar y recalcular la planificación? ¿Cuándo resulta interesante emplearlos?

La información necesaria puede encontrarse en las siguientes páginas:

https://es.mathworks.com/help/robotics/ug/probabilistic-roadmaps-prm.html

https://es.mathworks.com/help/robotics/ref/robotics.prm-class.html

3.2. PRUEBAS EN SIMULACIÓN

3.2.1. Comportamiento wander_localiza_planifica

En primer lugar, y tomando como punto de partida el fichero 'wander_localiza' anterior, se va a crear un nuevo script llamado **wander_localiza_planifica.m** que, una vez localizado el robot (por lo tanto, conocida su posición global dentro del mapa *estimatedPose*), planifique una ruta para llegar a una posición de destino *endLocation* utilizando el planificador PRM.

La estructura del programa, en la que se muestra en color rojo/negrita los apartados a añadir, es la siguiente:

```
%Definir la posicion de destino
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
endLocation = [x y];

%Cargar el mapa
%%%%%%%%%%%%%%%

%Crear el objeto VFH…y ajustar sus propiedades
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Inicializar el localizador AMCL (práctica 1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Rellenamos los campos por defecto de la velocidad del robot, para que la lineal
%sea siempre 0.1 m/s

%Bucle de control infinito
while(1)

    %Leer y dibujar los datos del láser en la figura 'fig_laser'

    %Leer la odometría

    %Obtener la posición pose=[x,y,yaw] a partir de la odometría anterior

    %Ejecutar amcl para obtener la posición estimada estimatedPose y la
    %covarianza estimatedCovariance (mostrar la última por pantalla para
    %facilitar la búsqueda de un umbral)

    %Si la covarianza está por debajo de un umbral, el robot está localizado y
    %finaliza el programa

    %Dibujar los resultados del localizador con el visualizationHelper

    %Llamar al objeto VFH para obtener la dirección a seguir por el robot para
    %evitar los obstáculos. Mostrar los resultados del algoritmo (histogramas)
    %en la figura 'fig_vfh'


    %Rellenar el campo de la velocidad angular del mensaje de velocidad con un
    %valor proporcional a la dirección anterior (K=0.1)


    %Publicar el mensaje de velocidad

    %Esperar al siguiente periodo de muestreo

end

%%%%%%%%%%%  AL SALIR DE ESTE BUCLE EL ROBOT YA SE HA LOCALIZADO %%%%%%%%%%
%%%%%%%%%%%  COMIENZA LA PLANIFICACIÓN GLOBAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Paramos el robot, para que no avance mientras planificamos

%Hacemos una copia del mapa, para "inflarlo" antes de planificar
cpMap= copy(map);
inflate(cpMap,0.25);

%Crear el objeto PRM y ajustar sus parámetros
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
planner = mobileRobotPRM;
planner.Map=
planner.NumNodes=
planner.ConnectionDistance =

%Obtener la ruta hacia el destino desde la posición actual del robot y mostrarla
%en una figura
```

Para ajustar el planificador, realice las siguientes pruebas:

1. Ajustar el parámetro "*NumNodes*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo.
2. Ajustar el parámetro "*ConnectionDistance*" a un valor alto, un valor bajo y un valor intermedio y comprobar su efecto sobre el comportamiento del algoritmo.
3. Ajustar todos los parámetros a la configuración óptima seleccionada.

3.2.2. Comportamiento wander_localiza_planifica_controla

Una vez planificada la ruta, el robot debe recorrerla utilizando un controlador de movimiento. Matlab proporciona la implementación de un controlador *PurePursuit* al que, pasándole un listado de puntos o *waypoints* (es decir, el resultado del planificador PRM), calcula las velocidades lineal y angular apropiadas para realizar dicho recorrido. La información sobre el funcionamiento de dicho controlador puede encontrarse en las siguientes páginas:

https://es.mathworks.com/help/robotics/ug/pure-pursuit-controller.html
https://es.mathworks.com/help/nav/ug/pure-pursuit-controller.html
https://es.mathworks.com/help/robotics/ref/robotics.purepursuit-system-object.html
https://es.mathworks.com/help/nav/ref/plannerprm.html

Se pide a continuación implementar un script llamado **wander_localiza_planifica_controla.m**, basado en el script realizado anteriormente que, una vez planificada la ruta, implemente un bucle de control utilizando el controlador "*PurePursuit*". A continuación se proporciona la estructura del programa, mostrándose en rojo/negrita las partes a añadir:

```
%Definir la posicion de destino
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Cargar el mapa
%%%%%%%%%%%%%%%

%Crear el objeto VFH…y ajustar sus propiedades
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Inicializar el localizador AMCL (práctica 1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Crear el objeto PurePursuit y ajustar sus propiedades
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
controller=controllerPurePursuit;
controller.LookaheadDistance =
controller.DesiredLinearVelocity=
controller.MaxAngularVelocity) =

%Rellenamos los campos por defecto de la velocidad del robot, para que la lineal
%sea siempre 0.1 m/s

%Bucle de control infinito
```

```
while(1)

    %Leer y dibujar los datos del láser en la figura 'fig_laser'

    %Leer la odometría

    %Obtener la posición pose=[x,y,yaw] a partir de la odometría anterior

    %Ejecutar amcl para obtener la posición estimada estimatedPose y la
    %covarianza estimatedCovariance (mostrar la última por pantalla para
    %facilitar la búsqueda de un umbral)

    %Si la covarianza está por debajo de un umbral, el robot está localizado y
    %finaliza el programa

    %Dibujar los resultados del localizador con el visualizationHelper

    %Llamar al objeto VFH para obtener la dirección a seguir por el robot para
    %evitar los obstáculos. Mostrar los resultados del algoritmo (histogramas)
    %en la figura 'fig_vfh'


    %Rellenar el campo de la velocidad angular del mensaje de velocidad con un
    %valor proporcional a la dirección anterior (K=0.1)


    %Publicar el mensaje de velocidad

    %Esperar al siguiente periodo de muestreo

end

%%%%%%%%%%%  AL SALIR DE ESTE BUCLE EL ROBOT YA SE HA LOCALIZADO %%%%%%%%%%
%%%%%%%%%%%  COMIENZA LA PLANIFICACIÓN GLOBAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Paramos el robot, para que no avance mientras planificamos

%Hacemos una copia del mapa, para "inflarlo" antes de planificar

%Crear el objeto PRM y ajustar sus parámetros
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Obtener la ruta hacia el destino desde la posición actual del robot y mostrarla
%en una figura


%%%%%%%%%%% COMIENZA EL BUCLE DE CONTROL %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Indicamos al controlador la lista de waypoints a recorrer (ruta)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
controller.Waypoints=

%Bucle de control
%%%%%%%%%%%%%%%%%
while(1)

      %Leer el láser y la odometría

      %Leer el láser y la odometría

      %Obtener la posición pose=[x,y,yaw] a partir de la odometría anterior
```

```
      %Ejecutar amcl para obtener la posición estimada estimatedPose

      %Dibujar los resultados del localizador con el visualizationHelper

      %Ejecutar el controlador PurePursuit para obtener las velocidades lineal
      %y angular

      %Rellenar los campos del mensaje de velocidad

      %Publicar el mensaje de velocidad

      %Comprobar si hemos llegado al destino, calculando la distancia euclidea
      %y estableciendo un umbral
      if (destino alcanzado)
            %Parar el robot

            break;
      end

      %Esperar al siguiente periodo de muestreo

end
```

3.2.3. Comportamiento navegación_total.

Finalmente, se va a incluir el evitador de obstáculos en el bucle de control para evitar que el robot quede bloqueado cuando se aproxima demasiado a las paredes, y a su vez evitar obstáculos imprevistos que bloqueen la ruta. Para ello, se va a combinar la velocidad angular calculada por "PurePursuit" con la calculada por el evitador de obstáculos (proporcional a la dirección seleccionada por VFH).

Crear un script llamado **navegacion_total.m**, a partir del anterior, que incorpore los apartados mostrados en rojo en el siguiente esquema:

```
  …
(la parte anterior del código no cambia)

%%%%%%%%%%% COMIENZA EL BUCLE DE CONTROL %%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Indicamos al controlador la lista de waypoints a recorrer (ruta)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Bucle de control
%%%%%%%%%%%%%%%%%
while(1)

      %Leer el láser y la odometría

      %Leer el láser y la odometría

      %Obtener la posición pose=[x,y,yaw] a partir de la odometría anterior

      %Ejecutar amcl para obtener la posición estimada estimatedPose

      %Dibujar los resultados del localizador con el visualizationHelper

      %Ejecutar el controlador PurePursuit para obtener las velocidades lineal
      %y angular
```

```
      %Llamar a VFH pasándole como "targetDir" un valor proporcional a la
      %velocidad angular calculada por el PurePursuit

      %Calcular la velocidad angular final como una combinación lineal de la
      %generada por el controlador PurePursuit y la generada por VFH

      %Rellenar los campos del mensaje de velocidad

      %Publicar el mensaje de velocidad

      %Comprobar si hemos llegado al destino, calculando la distancia euclidea
      %y estableciendo un umbral

      %Esperar al siguiente periodo de muestreo

end
```

3.3. PRUEBAS CON EL ROBOT REAL

Una vez validados los algoritmos en simulación, se pide:

1. Emplear la mejor configuración de parámetros obtenida en el apartado de simulación para comprobar el funcionamiento en modo real.
2. ¿Es suficientemente realista la simulación como para validar dichos parámetros en modo real?
3. En caso de no ser suficiente, ajustar los parámetros en modo real.