Anexos

Anexo I. Algoritmo detección de fronteras

En este Anexo se transcribe el código de la función utilizado para la detección de fronteras partiendo de un mapa y de la selección del método de agrupación de fronteras utilizado en el apartado 4.1.

Este código parte de un mapa de ocupación, del cual se obtiene su matriz de ocupación (*occupancyMatrix*) y mediante la acción *'ternary'* las celdas de dicha matriz tomarán los valores 1, 0 o -1 en función de a qué valor se aproximen más. Como se ha comentado en el apartado 4.1 la información que ofrecen estos valores indica el estado de la celda (ocupada, libre o desconocida). Para cada valor de esta matriz se estudiarán todas las celdas cuyo valor sea 0 con la finalidad de analizar las celdas de su alrededor buscando que sean valores desconocidos pues dichas celdas serían las fronteras que se busca detectar. El resultado que ofrece este algoritmo es un vector formado por la posición de las celdas de la matriz de ocupación en las que hay fronteras.

Algoritmo detectorFronteras(map)

```
global metodocluster;
nuevolustersCentroids=[];
mat = occupancyMatrix(map,'ternary');
[a,b]=size(mat);
VectorFronteras=[];
for i=2:a-1
   for j=2:b-1
      if mat(i,j)==0
        if mat(i+1,j)==-1 ||mat(i-1,j)==-1 || mat(i,j+1)==-1 || mat(i,j-1)==-1
          VectorFronteras= [VectorFronteras; i j];
        end
      end
   end
end
```

Una vez detectadas todas las fronteras se procede a agruparlas mediante los dos *clusters* mencionados en el apartado 4.2 *clusterXY* y *meanshift*, a su vez se ejecuta la condición que si estas agrupaciones contienen menos de 6 fronteras, estas agrupaciones no se deberán de tener en cuenta.

```
if metodocluster=="clusterXY"
  [clustersCentroids,clustersGeoMedians,clustersXY] =
  clusterXYpoints('vector.mat',0.8,1,'centroid','merge')
  newclustersCentroids=clustersCentroids;
```

```
 [a, b]=size(clustersXY);
 for j=a:-1:1
     if (length(clustersXY{j}) <6)
        newclustersCentroids(j,:) = [ ];
        clustersXY(j,:)=[ ];
     end
end
nuevolustersCentroids=newclustersCentroids;
elseif metodocluster=="meanshift"
 profile on
 bandwidth = 0.9;
 x=(vector)';
  [clustCent,point2cluster,clustMembsCell] = MeanshiftCluster(x,bandwidth);
 numClust = length(clustMembsCell);
 newclustCent=clustCent;
 [a, b]=size(clustMembsCell);
 for j=a:-1:1
     if (length(clustMembsCell{j}) <6)
        newclustCent(:,j) = [];
        clustMembsCell(j,:)=[];
     end
 end
 nuevolustersCentroids=newclustCent';
end
```

En las siguientes líneas de código, común a ambos métodos *clusters*, se estudia para los valores de las fronteras obtenidos si la posición de estas fronteras se encuentra en zona desconocida u ocupada, y si es el caso, se estudiará en un radio de 5 celdas si hay alguna celda vacía. En caso de que así sea, la posición de dicha celda, será la posición final de la frontera, y en caso de que no haya ninguna celda vacía en el radio de 5, se obviará dicha frontera. Finalmente se obtiene y se muestra el vector Frontera el cual engloba todas las fronteras obtenidas tras la realización de este algoritmo.

```
nuevovector = world2grid(map,nuevolustersCentroids);
fronterascorrectas=[];
for i=1:length(nuevovector)
  if mat(nuevovector(i,1),nuevovector(i,2))~=0
     for n=1:5
        if mat(nuevovector(i,1)+n,nuevovector(i,2))==0
          fronterascorrectas=[fronterascorrectas;nuevovector(i,1)+n  nuevovector(i,2)];
          break
        elseif mat(nuevovector(i,1)-n,nuevovector(i,2))==0
          fronterascorrectas=[fronterascorrectas;nuevovector(i,1)-n nuevovector(i,2)];
```

```
            break
          elseif mat(nuevovector(i,1),nuevovector(i,2)+n)==0
            fronterascorrectas=[fronterascorrectas;nuevovector(i,1) nuevovector(i,2)+n];
            break
          elseif mat(nuevovector(i,1),nuevovector(i,2)-n)==0
            fronterascorrectas=[fronterascorrectas;nuevovector(i,1) nuevovector(i,2)-n];
            break
         elseif mat(nuevovector(i,1)-n,nuevovector(i,2)+n)==0
            fronterascorrectas=[fronterascorrectas;nuevovector(i,1)-n nuevovector(i,2)+n];
            break
          elseif mat(nuevovector(i,1)-n,nuevovector(i,2)-n)==0
            fronterascorrectas=[fronterascorrectas;nuevovector(i,1)-n nuevovector(i,2)-n];
            break
          elseif mat(nuevovector(i,1)+n,nuevovector(i,2)+n)==0
           fronterascorrectas=[fronterascorrectas;nuevovector(i,1)+n nuevovector(i,2)+n];
            break
          elseif mat(nuevovector(i,1)+n,nuevovector(i,2)-n)==0
            fronterascorrectas=[fronterascorrectas;nuevovector(i,1)+n nuevovector(i,2)-n];
            break
          end
      end
          else
           fronterascorrectas=[fronterascorrectas;nuevovector(i,1) nuevovector(i,2)];
  end
end
Fronteras= fronterascorrectas;
mat = occupancyMatrix(map,'ternary');
vector2= grid2world(map,fronterascorrectas);
```

Anexo II. Algoritmo cálculo de la frontera útil

En este Anexo, se muestra el código redactado con la finalidad de aplicar el proceso de selección de fronteras. Las variables de entrada son el mapa del entorno *map*, la posición inicial del robot, el radio de la zona a estudiar *radius*, y la distancia mínima a la que debe de estar el robot de su punto de destino. Cuando el robot se encuentre a menos de esa distancia de su destino, se dará por concluida la acción de continuar hacia la frontera y busque la siguiente frontera, ya que si la distancia de la trayectoria creada por el camino es menor que este valor, el robot no será capaz de moverse hacia dicha frontera pues ya se encuentra bajo esa condición. Esto es aplicable principalmente en el criterio de elección de la frontera más cercana. Este código da como resultado un vector que representa la frontera seleccionada como de mayor utilidad para el proceso de exploración robótica expresada en metros.

Algoritmo FuncionUtilidad(map, PosicionInicialRobot,radius,goalRadius)

```
global metodoeleccion;
[Frontera] = detectorfronteras(map);
mat = occupancyMatrix(map,'ternary');
DistanciaHastaFronteras=[ ];
DistanciaCadaPath=0;
Fronteras= grid2world(map,Frontera);
path=[ ];
FronterasCamino=Fronteras;
startLocation =[PosicionInicialRobot(1) PosicionInicialRobot(2) 0];
for l=length(Fronteras):-1:1
  endLocation =Fronteras(l,:);
  [robotpath] = calcularpath(map, startLocation, endLocation)
  if isempty(robotpath)
    FronterasCamino(l,:) = [ ];
  else
    dist = 0;
    for a=2:length(robotpath(:,1))
    dist =sqrt((robotpath(a,1)-robotpath(a-1,1))^2 +(robotpath(a,2)-robotpath(a-1,2))^2);
    DistanciaCadaPath= DistanciaCadaPath+dist;
    end
    if DistanciaCadaPath>goalRadius
      DistanciaHastaFronteras=[DistanciaHastaFronteras; DistanciaCadaPath]
      DistanciaCadaPath=0;
   else
     FronterasCamino(l,:) = [ ];
   end
  end
end
MaximaDistancia=max(DistanciaHastaFronteras);
```

```
FronterasConCamino=flip(FronterasCamino);
DistanciaHastaFronterasNorm=(DistanciaHastaFronteras)/MaximaDistancia;
celdasdesconocidas=[];
r=radius;
n=map.Resolution*r;
FronteraConCamino= world2grid(map,FronterasConCamino)
for g=1:length(FronteraConCamino)
   puntosdesconocidos=0;
   limite_inferior1=min((FronteraConCamino(g,1)-n),map.GridSize(1,1));
   limite_superior1=min((FronteraConCamino(g,1)+n),map.GridSize(1,1));
   limite_inferior2=min((FronteraConCamino(g,2)-n),map.GridSize(1,2));
   limite_superior2=min((FronteraConCamino(g,2)+n),map.GridSize(1,2));
   for i=limite_inferior1:limite_superior1
      for j= limite_inferior2:limite_superior2
         if mat(i,j)==-1
           puntosdesconocidos=puntosdesconocidos+1;
         end
      end
   end
      celdasdesconocidas=[celdasdesconocidas; puntosdesconocidos];
end
CeldasDesconocidas=celdasdesconocidas;
Porcentajeceldasdesconocidas= celdasdesconocidas/sum(celdasdesconocidas,'all');
if metodoeleccion== "distancia"
  PosicionFronteraElegida=find(DistanciaHastaFronteras==
  min(DistanciaHastaFronteras));
  FronteraUtilMetros=FronterasConCamino(PosicionFronteraElegida,:);
elseif metodoeleccion== "entropia"
   [~, PosicionFrontElegida] = max(CeldasDesconocidas)
   FronteraUtilMapa=FronteraConCamino(PosicionFrontElegida,:)
   FronteraUtilMetros= grid2world(map,FronteraUtilMapa);
elseif metodoeleccion== "aleatorio"
   NumeroFronteras=1:length(FronterasConCamino);
   FronteraUtilMetros = randsample(NumeroFronteras,1);
   FronteraRandom=[FronteraPosible(PosicionAleatoria,1),FronteraPosible(
    PosicionAleatoria,2)];
elseif metodoeleccion== "entropiadistancia"
  CeldasDesconocidas=celdasdesconocidas/(2*n+1)^2;
  UtilidadFronteras=CeldasDesconocidas+0.5./DistanciaHastaFronterasNorm;
  [~, PosicionFrontElegida] = max(UtilidadFronteras)
  FronteraUtilMapa=FronteraConCamino(PosicionFrontElegida,:)
  FronteraUtilMetros= grid2world(map,FronteraUtilMapa)
 end
```

Anexo III. Algoritmo cálculo del camino

En este Anexo se transcribe el código de la función utilizado para crear un camino entre dos puntos partiendo de un mapa, de la posición inicial del robot en forma de vector 1x3 y de la posición final deseada en forma de vector 1x2, también se ha de especificar que método de cálculo del camino se desea aplicar, PRM o RRT.

```
Algoritmo calcularpath(map, startLocation, endLocation)

global metodopath;
show(map);
```

A continuación se muestran las variables definidas para la aplicación del algoritmo de planificación PRM de los valores de las variables comentadas en el apartado 4.3.
Para este método se necesitan como datos de entrada la posición inicial y la final ambas en la forma de vector 1x2 e indicar que el PRM usado es el *mobileRobotPRM*
El número de nodos seleccionado ha sido 30 ya que era el mínimo número para el cual en la mayoría de mapas se creaba un camino y se ha añadido un bucle que incrementa este valor para los casos en cuales con 30 nodos sea insuficiente para crear un camino. Como los entornos son de dimensiones bajas, de alrededor de $40m^2$, se selecciona una distancia de conexión de 1.5 metros ya que interesa que los caminos sean lo más eficaces posibles y si esta distancia es muy elevada, se podría crear un camino que para llegar a un destino determinado deba de recorrer más distancia que la necesaria.

```
if metodopath=="PRM"
    startLocation(:,3)=[];
    prm = mobileRobotPRM;
    prm.Map = map;
    prm.NumNodes = 30;
    prm.ConnectionDistance = 1.5;
   path = findpath(prm, startLocation, endLocation);
   while isempty(path) &&  prm.NumNodes < 1500
      prm.NumNodes = prm.NumNodes + 250;
      update(prm);
      path = findpath(prm, startLocation, endLocation);
   end
  robotpath=path;
```

En las siguientes líneas de código, se muestra cómo se ha aplicado el algoritmo de planificación RRT. Para este algoritmo, es necesario introducir la posición inicial y final de la forma de un vector 3x1, también se ha de introducir el espacio de estados, *stateSpaceSE2 y* el validador de estados *validatorOccupancyMap(ss)* ambos con sus respectivos valores de las variables mostradas en el código.

```
elseif metodopath=="RRT"
        goal=[endLocation(1,1), endLocation(1,2), 0];
        start=[startLocation(1,1), startLocation(1,2),0];
        ss = stateSpaceSE2;
        sv = validatorOccupancyMap(ss);
        sv.Map = map;
        sv.ValidationDistance = 0.05;
        ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
        planner = plannerRRT(ss,sv);
        planner.MaxIterations=4000;
        planner.MaxConnectionDistance=0.8;
        planner.GoalReachedFcn = @exampleHelperCheckIfGoal;
        rng(100,'twister');
        [pthObj,solnInfo] = plan(planner,start, goal);
        robotpath= [pthObj.States(:,1) pthObj.States(:,2)] ;
    end
end
```

Anexo IV. Código del algoritmo de exploración robótica

En este anexo, se muestra el algoritmo que realiza la exploración robótica. Inicialmente se realiza la conexión Gazebo-ROS en MATLAB y se inicializa al robot, se procederá a la creación del objeto *Lidar* SLAM, a la obtención del mapa inicial de ocupación y la inicialización del controlador VFH definiendo sus variables.

A continuación, se inicializa el bucle mediante el cual se va a realizar la exploración robótica, este bucle se ejecutará mientras el vector de fronteras detectadas no esté vacío lo cual indica que hay zonas a explorar todavía.

Dentro de este bucle se selecciona la frontera de mayor utilidad aplicando el código mostrado en el Anexo III, una vez obtenida esta frontera, se procede a calcular el camino que ha de recorrer el robot para llegar a ella mediante la utilización de la función que calcula el *path* mostrada en el Anexo II.

Una vez definido el conjunto de puntos a seguir por el robot en el *path (waypoints)*, se procede a inicializar el controlador *PurePursuit* el cual se encargará de la realización del seguimiento de la trayectoria, este seguimiento de trayectoria se ejecutará hasta que el robot reconozca que se va a chocar o se encuentre a una distancia menor del *goalRadius*. Para el primer caso, el robot retrocede una determinada distancia para volver a calcular un nuevo *path* hacia la misma frontera que se había definido como de mayor utilidad y se procede a seguir la trayectoria hacia esa frontera de nuevo pero por otro camino. En el segundo caso, se da por alcanzada la frontera elegida, dando por concluido el bucle y se procede a calcular la siguiente frontera mediante la función mostrada en el Anexo I y en caso de que no hubiese ninguna frontera, se da la ejecución del algoritmo por finalizada.

```
clc
clear all
rosshutdown
ipaddress = "http://192.168.9.132:11311";
rosinit(ipaddress)
rostopic list
robotCmd = rospublisher('/cmd_vel', 'geometry_msgs/Twist') ;
velMsg = rosmessage(robotCmd);
lidarSub = rossubscriber("/scan");
odomSub = rossubscriber("/odom");
odomMsg = receive(odomSub,3);
maxLidarRange = 8;
mapResolution =20;
slamAlg = lidarSLAM(mapResolution,maxLidarRange);
slamAlg.LoopClosureThreshold = 350;
slamAlg.LoopClosureSearchRadius = 4;
controlRate = rateControl(20);
```

```
pause(1)
velMsg.Linear.X =0.15;
send(robotCmd,velMsg)
t_move=1.5;
tic
while toc < t_move
  scanMsg = receive(lidarSub);
  range = double(lidarSub.LatestMessage.Ranges);
  angles = double(scanMsg.readScanAngles);
  range(range==Inf)= maxLidarRange+2;
  scans = lidarScan(range,angles);
  [isScanAccepted,loopClosureInfo,optimizationInfo] = addScan(slamAlg,scans);
  waitfor(controlRate);
end
velMsg.Linear.X = 0;
send(robotCmd,velMsg)
[scans,optimizedPoses]  = scansAndPoses(slamAlg);
map = buildMap(scans,optimizedPoses,mapResolution,maxLidarRange);
vfh = controllerVFH;
vfh.UseLidarScan = true;
vfh.DistanceLimits = [0.05 1];
vfh.RobotRadius = 0.16;
vfh.MinTurningRadius = 0.1;
vfh.SafetyDistance = 0.05;
targetDir = 0;
rate = rateControl(10);
```

Se inicializa el bucle mediante el cual se va a realizar la exploración robótica, este bucle se repetirá hasta que no se detecte ninguna frontera.

```
Fronteras=detectorFronteras(map)
goalRadius = 0.165;
Collision= false;
while  ~  isempty(Fronteras)
 if Collision == true
    velMsg.Linear.X = -0.1;
    velMsg.Angular.Z = 0;
    send(robotCmd,velMsg)
    t_check=1.3;
    tic
    while toc< t_check
      scanMsg = receive(lidarSub);
      range = double(lidarSub.LatestMessage.Ranges);
      angles = double(scanMsg.readScanAngles);
      range(range==Inf)= maxLidarRange+2;
```

```
    scans = lidarScan(range,angles);
    [isScanAccepted,loopClosureInfo,optimizationInfo] = addScan(slamAlg,scans);
    waitfor(controlRate);
  end
  velMsg.Linear.X = 0;
  send(robotCmd,velMsg)
  [scans,optimizedPoses]  = scansAndPoses(slamAlg);
  map = buildMap(scans,optimizedPoses,mapResolution,maxLidarRange);
  mapInflated = copy(map);
  inflate(mapInflated, 0.16);
  odomMsg = receive(odomSub,3);
  startLocation =[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y];
  robotpath=[];
  matr = occupancyMatrix(mapInflated,'ternary');
  endLoc= world2grid(mapInflated,FronteraUtilMetros(1,:));
  startLoc= world2grid(mapInflated,startLocation);
  if matr(endLoc(1,1),endLoc(1,2))==0 &&  matr(startLoc(1,1),startLoc(1,2))==0
    odomMsg = receive(odomSub,3);
    startLocation=[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y 0];
    [robotpath] = calcularpath(mapInflated,startLocation, FronteraUtilMetros(1,:))
    path=robotpath;
  end
else
  Collision= false;
  close all
  mapInflated = copy(map);
  inflate(mapInflated, 0.165);
  show(mapInflated)
  odomMsg = receive(odomSub,3);
  startLocation = [odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y];
  radio=0.5;
  global metodocluster;
  metodocluster="clusterXY";
  global metodopath;
  metodopath="PRM";
  global metodoeleccion;
  metodoeleccion="entropiadistancia";%distancia ,entropia ,aleatorio, entropiadistancia
  FronteraUtilMetros=[];
  odomMsg = receive(odomSub,3);
  startLocation =[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y];
  [CeldaLibre] = CorregirCeldaOcupada(startLocation)
 odomMsg = receive(odomSub,3);
 startLocation =[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y];
 [FronteraUtilMetros] = FuncionUtilidad(mapInflated, startLocation,radio, goalRadius)
  odomMsg = receive(odomSub,3);
```

```
   startLocation =[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y 0];
   [robotpath] = calcularpath(mapInflated,startLocation, FronteraUtilMetros(1,:))
   path=robotpath;
end
Collision= false;
controller = controllerPurePursuit;
controller.Waypoints = path;
controller.DesiredLinearVelocity =0.16;
controller.MaxAngularVelocity = 0.55;
controller.LookaheadDistance = 0.75;
release(controller);
```

Una vez definido el controlador *PurePursuit* se procede a realizar el seguimiento de la trayectoria a la vez que se actualiza el *OccupancyMap* con las partes del entorno que se van conociendo. La trayectoria se ejecuta hasta que el robot se encuentre a una distancia menor del *goalRadius.* Una vez alcanzada la frontera elegida se da por concluido el bucle y se procede a calcular la siguiente frontera.

```
robotGoal = path(end,:);
odomMsg = receive(odomSub,3);
quat = [odomMsg.Pose.Pose.Orientation.W odomMsg.Pose.Pose.Orientation.X
odomMsg.Pose.Pose.Orientation.Y odomMsg.Pose.Pose.Orientation.Z];
eulZYX = quat2eul(quat);
robotCurrentPose=[odomMsg.Pose.Pose.Position.X  odomMsg.Pose.Pose.Position.Y
eulZYX(1)]';
distanceToGoal = norm([odomMsg.Pose.Pose.Position.X
odomMsg.Pose.Pose.Position.Y] - robotGoal);
while( distanceToGoal > goalRadius )
      odomMsg = receive(odomSub,3);
      quat = [odomMsg.Pose.Pose.Orientation.W odomMsg.Pose.Pose.Orientation.X
      odomMsg.Pose.Pose.Orientation.Y odomMsg.Pose.Pose.Orientation.Z];
      eulZYX = quat2eul(quat);
      robotCurrentPose= [odomMsg.Pose.Pose.Position.X
      odomMsg.Pose.Pose.Position.Y  eulZYX(1)]';
      [v, omega] = controller(robotCurrentPose);
      if abs(omega)>0.4
         v=0.025;
      end
      velMsg.Linear.X = v;
      velMsg.Angular.Z= omega;
      t_scan=0.125;
       tic
       while toc< t_scan
          scanMsg = receive(lidarSub);
```

```
        range = double(lidarSub.LatestMessage.Ranges);
        angles = double(scanMsg.readScanAngles);
        range(range==Inf)= maxLidarRange+2;
        scans = lidarScan(range,angles);
        steerDir = vfh(scans, targetDir);
        if isnan(steerDir)
            Collision = true;
             break
        end
        send(robotCmd,velMsg)
        [isScanAccepted,loopClosureInfo,optimizationInfo] = addScan(slamAlg,scans);
        [scans,optimizedPoses]  = scansAndPoses(slamAlg);
        waitfor(controlRate);
    end
    if Collision == true
       break
    end
    velMsg.Linear.X = 0 ;
    velMsg.Angular.Z= 0;
    send(robotCmd,velMsg)
    odomMsg = receive(odomSub,3);
    quat = [odomMsg.Pose.Pose.Orientation.W odomMsg.Pose.Pose.Orientation.X
    odomMsg.Pose.Pose.Orientation.Y odomMsg.Pose.Pose.Orientation.Z];
    eulZYX = quat2eul(quat);
    robotCurrentPose= [odomMsg.Pose.Pose.Position.X
    odomMsg.Pose.Pose.Position.Y  eulZYX(1)]';
    distanceToGoal = norm(robotCurrentPose(1:2) - robotGoal(:))
    map = buildMap(scans,optimizedPoses,mapResolution,maxLidarRange);
  end
  velMsg.Linear.X=0;
  velMsg.Angular.Z = 0;
  send(robotCmd,velMsg)
  scanMsg = receive(lidarSub);
  range = double(lidarSub.LatestMessage.Ranges);
  angles = double(scanMsg.readScanAngles);
  range(range==Inf)= maxLidarRange+2;
  scans = lidarScan(range,angles);
  [isScanAccepted,loopClosureInfo,optimizationInfo] = addScan(slamAlg,scans);
  [scans,optimizedPoses]  = scansAndPoses(slamAlg);
  map = buildMap(scans,optimizedPoses,mapResolution,maxLidarRange);
  show(map)
  [Fronteras] = detectorfronteras(mapInflated);
end
```