ANEXO 1. Código del simulador

Código propio del simulador

```
float PrA[][]=new float[100][4];  //Puntos de casa coordenadas
rectangulares Absolutas
float ppL[][]=new float[100][6];  //Puntos de casa coordenadas polares
Locales
              //el segmento forma parte de la recta Y=ppL[4]*x+ppL[5]
              //si el segmento es vertical se hacen ppL[4]=0 y
ppL[5]=0

float LIDAR[][]=new float[360][2]; //Puntos calculados del lidar
                                   //están en la forma angulo, radio
float LIDARXYr[][]=new float[360][2];  //Puntos calculados del lidar
en coordenadas cartesianas referencia robot
float LIDARXYg[][]=new float[360][2];  //Puntos calculados del lidar
en coordenadas globales
float Segmentos1[][]=new float[500][6];  //Segmentos de la forma
(ángulo,a,x1,y1,x2,y2) extraidos en cada toma de datos
float SegmentosMapa[][]=new float[500][6]; //Conjunto de segmentos que
componen el mapa de la forma (ángulo,a,x1,y1,x2,y2)
int Extension=1;                           //Tamaño del mapa de
segmentos de cada toma de datos
int iaux=0;

ArrayList<Integer> Puntos = new ArrayList();
ArrayList<Integer> Clust = new ArrayList();
int Segmentos[][]=new int[500][2];

int NCasa;
BufferedReader reader;
float Esc=.06;                    //Escala para dibujar
int escala=50;                    //Escala del plano para los puntos
de casa

float X;                    //Posición del robot
float Y;
float Fi;     //Orientación del robot (respecto del eje X global)

float Xs;                    //Posición simulada del robot
float Ys;
float Fis;     //Orientación del robot simulada (respecto del eje X
global)

float Xr=0;     //Ponemos a cero las coordenadas del robot para
iniciar la creación del mapa
float Yr=0;
float Fir=0;
float Xd=3000;     //Coordenada X a la que queremos llegar
float Yd=1000;     //Coordenada Y a la que queremos llegar
float Fid;         //Orientación de llegada

float m;           //Pendiente de la línea de unión con el destino
float a;           //Valor a de y=mx+a de la línea de unión con el
destino
float orient;      //Ángulo de la dirección de unión con el destino
```

```
float Vd;                           //Velocidad rueda derecha
float Vi;                           //Velocidad rueda izquierda
float R=4;                          //Radio rueda
float t=0.05;                       //Cada cuantos segundos muestrea
float n,sumX,sumY,sumx,sumy,sumXY,sumxX,sumyY,sumxY,sumyX;

void setup()
{
  Fi=30;
  Fi=Fi*PI/180;
  Fid=Fid*PI/180;
  size(1000,600);
  leedatos();
  stroke(0);
  X=3000;
  Y=4600;
  Vd=0;
  Vi=0;
}

void draw()
{
  //background(7000000);
  strokeWeight(1);
  dibuja_casa();
  dibuja_robot();
  calcula_casa();
  //verifica_ppL();
  calcula_lidar();
  //verifica_lidar();
  mapa();
  RefFija();
  Cluster();
  SegmentaClust();
  AmpliaMapa();
  CalculaPos();
  CalculaUnion();
  CalculaTrayectoria();
  Limpia1();
  Limpia2();
}

/*
    Calcula los puntos que vería el LIDAR
    quedan de la forma angulo, radio
*/
void calcula_lidar()
{
  float ang;
  float d,r,radio;
  boolean implicado=false;
  r=0;
  ang=0;
  d=PI/180;         //incremento de angulo 1°
  for (int i=0;i<360;i++)
  {
    radio=100000;
    for (int j=0;j<NCasa;j++)
    {
      float t1,t2;    //Fases menor y mayor de los extremos
      t2=ppL[j][3];
```

```
      t1=ppL[j][1];
      if (ppL[j][1]>ppL[j][3])
        {
          t1=ppL[j][3];
          t2=ppL[j][1];
        }
      implicado=(((t2-t1)>PI) && ((0<=ang && ang<=t1)||(t2<=ang &&
ang<=2*PI))) || ((t2-t1)<=PI && t1<=ang && ang<=t2);

      //Si el radio corta con el segmento.
      if (implicado==true)
      {
        //Si el segmento está alineado con el radio
        if (ang==ppL[j][1] && ang==ppL[j][3])
          {
            r=ppL[j][0];
            if (ppL[j][2]<r)
              r=ppL[j][2];
          }
        else
          {
            //Si la recta es vertical
            if ((ppL[j][4]==ppL[j][5]) && (ppL[j][4]==0))
                r=ppL[j][0]*cos(ppL[j][1])/cos(ang);
            else
                r=ppL[j][5]/(sin(ang)-ppL[j][4]*cos(ang));
          }
      }
      if (r<0) r=-r;
      if (r<radio)
        radio=r;
    }

    LIDAR[i][0]=ang;
    LIDAR[i][1]=radio*(1+random(-1,1)*FRuido);
    ang=ang+d;
  }
}


//Pinta los puntos del lidar en coordenadas globales


void verifica_lidar()
{
  float x,y;
  stroke(255,0,0);
  for (int i=0;i<360;i++)
  {
    float coco=LIDAR[i][0]+Fi;
    x=Esc*(X+LIDAR[i][1]*cos(coco));
    y=Esc*(Y+LIDAR[i][1]*sin(coco));
    point(x,y);
    punto(500+x,y);
  }
}


//Dibuja una cruz en la posicion x,y

void punto(float x, float y)
```

```
{
  for (int i=-1;i<2;i++)
  {
      point (x+i,y);
      point (x,y+i);
  }
}

/*
*    Calcula los puntos de casa en coordenadas polares locales
*    Tambien calcula la recta y=ppL[i][4]*x+pp[i]L[5] a la que
pertenece el segmento
*    Si el segmento es vertical se hace ppL[i][4]=0 y ppL[i][5]=0
*/

void calcula_casa()
{  float x1,y1,x2,y2;        //coordenadas cartesianas locales
paralelas al sistema aboluto
   for (int i=0;i<NCasa;i++)
   {
     x1=PrA[i][0]-X;
     y1=PrA[i][1]-Y;
     ppL[i][0]=sqrt(x1*x1+y1*y1);    //El radio es el mismo en todas
las locales
     ppL[i][1]=atan2(y1,x1)-Fi;      //El ángulo en las solidarias al
robot
     //Si el ángulo es negativo pilla el equivalnte positivo
     if (ppL[i][1]<0) ppL[i][1]=2*PI+ppL[i][1];

     x2=PrA[i][2]-X;
     y2=PrA[i][3]-Y;
     ppL[i][2]=sqrt(x2*x2+y2*y2);
     ppL[i][3]=atan2(y2,x2)-Fi;
     //Si el ángulo de fase es negativo pilla el equivalente positivo
     if (ppL[i][3]<0) ppL[i][3]=2*PI+ppL[i][3];

     //Si la recta es vertical hace cero los términos 4 y 5 (pendiente
y offset de la recta)
     if(ppL[i][0]*cos(ppL[i][1])==ppL[i][2]*cos(ppL[i][3]))
     {
       ppL[i][4]=0;
       ppL[i][5]=0;
     }
     else
     {
       ppL[i][4]=(ppL[i][0]*sin(ppL[i][1])-
ppL[i][2]*sin(ppL[i][3]))/(ppL[i][0]*cos(ppL[i][1])-
ppL[i][2]*cos(ppL[i][3]));
       ppL[i][5]=ppL[i][0]*sin(ppL[i][1])-
ppL[i][4]*ppL[i][0]*cos(ppL[i][1]);
     }
   }
}


//Dibuja el robot con su posición y orientación

void dibuja_robot()
{
  float r=25;
  float cx,cy;
```

```
  cx=X*Esc;
  cy=Y*Esc;
  stroke(0);
  ellipse(cx,cy,r,r);
  stroke(0,0,255);
  line(cx,cy,cx+r*cos(Fi),cy+r*sin(Fi));
  stroke(255,0,0);
  line(cx,cy,cx-r*sin(Fi),cy+r*cos(Fi));
}


//Dibuja las paredes

void dibuja_casa()
{
  stroke(0);
  for (int i=0;i<NCasa;i++)
    line (Esc*PrA[i][0],Esc*PrA[i][1],Esc*PrA[i][2],Esc*PrA[i][3]);
}

/*
*  Lee los puntos de las paredes de casa, se multiplican por escala
*  se les suma 100 en las dos componenetes para no pasar por el 0,0
*/
void leedatos()
{
  String linea;
  reader = createReader("casa.txt");
  try
  {
    linea = reader.readLine();
  }
  catch (IOException e)
  {
    e.printStackTrace();
    linea = null;
  }

  if (linea != null)
  {
    String[] trozos = split(linea, TAB);
    NCasa = int(trozos[0]);
  }

  int i=0;
  while(linea!=null)
  {
    try
    {
      linea = reader.readLine();
    }
    catch (IOException e)
    {
      e.printStackTrace();
      linea = null;
    }
    if (linea != null)
    {
      String[] trozos = split(linea, TAB);
      PrA[i][0] = 100+escala*float(trozos[0]);
      PrA[i][1] = 100+escala*float(trozos[1]);
```

```
      PrA[i][2] = 100+escala*float(trozos[2]);
      PrA[i][3] = 100+escala*float(trozos[3]);
      i++;
    }
  }
}

//Calcula la posición según las velocidades de las ruedas y el
intervalo de tiempo de recogida de datos "t"

void CalculaPos()
{
  float V;
  float omeg;
  V=(Vd+Vi)/2;
  omeg=(Vd-Vi)/300;
  Fi=Fi+omeg*t;                        //Respecto de las coordenadas
globales de la pantalla
  X=X+V*cos(Fi+(PI/2))*t;
  Y=Y+V*sin(Fi+(PI/2))*t;

  Fis=Fis+omeg*t;                      //Para la nueva referencia
global, que es la pos inicial del robot. Simuladas
  Xs=Xs+V*cos(Fis+(PI/2))*t;
  Ys=Ys+V*sin(Fis+(PI/2))*t;

  Fir=Fir*(1+random(-1,1)*FRuido);   //Para la nueva referencia
global, que es la pos inicial del robot. Reales
  Xr=Xr*(1+random(-1,1)*FRuido);
  Yr=Yr*(1+random(-1,1)*FRuido);
}
```

Código de las funciones de los algoritmos SLAM

```
//Información que almacena en cada momento que el lidar recoge
información pasada a coordenadas cartesianas referencia robot

void mapa()
{
 strokeWeight(2);
 stroke(255,0,0);
 for (int i=0;i<360;i++)
 {
  LIDARXYr[i][0]= LIDAR[i][1]*cos(LIDAR[i][0]);
  LIDARXYr[i][1]= LIDAR[i][1]*sin(LIDAR[i][0]);
  //point((LIDARXYr[i][0]+X)*Esc,(LIDARXYr[i][1]+Y)*Esc);
 }
}

//Información que almacena en cada momento el lidar, pasada a
coordenadas cartesianas referencia global (posición inicial del robot)

void RefFija()
{
  stroke(0,0,255);
  strokeWeight(1);
  for (int i=0;i<360;i++)
```

```
  {
    LIDARXYg[i][0]= LIDARXYr[i][0]*cos(Fir)-
LIDARXYr[i][1]*sin(Fir)+Xr;
    LIDARXYg[i][1]=
LIDARXYr[i][0]*sin(Fir)+LIDARXYr[i][1]*cos(Fir)+Yr;
    point((LIDARXYg[i][0]+11000)*Esc,(LIDARXYg[i][1]+4000)*Esc);
  }
}

//Se separan los puntos en cluster o zonas diferenciadas

void Cluster()
{
  int aux=1;
  Clust.add(0);
  stroke(0,0,255);
  strokeWeight(5);
  int a=0;
  for (int i=1;i<359;i++)
  {
    float
dis=sqrt((LIDAR[i][1])*(LIDAR[i][1])+(LIDAR[i+1][1])*(LIDAR[i+1][1])-
2*(LIDAR[i][1])*(LIDAR[i+1][1])*cos((LIDAR[i+1][0])-(LIDAR[i][0])));
    float disMin=400;
    if (dis>disMin)
    {
      if ((i+1)!=(Clust.get(a)+aux)) //Si dos clusters son
consecutivos se elimina
      {
        Clust.add(i+1);
        point((LIDARXYg[i][0]+11000)*Esc,(LIDARXYg[i][1]+4000)*Esc);
//Final de uno

point((LIDARXYg[i+1][0]+11000)*Esc,(LIDARXYg[i+1][1]+4000)*Esc);
//Inicio de otro
        a=a+1;
        aux=1;
      }
      else
      {
        aux=aux+1;
      }
    }
  }
  Clust.remove(0);
}

//Algoritmo iterative end-fit point

void Segmenta1(int start, int end)
{
  float epsilon = 30;
  double dmax = 0.0;
  int idx = 0;
  float m;
  m = (LIDARXYg[end][1]-LIDARXYg[start][1])/(LIDARXYg[end][0]-
LIDARXYg[start][0]);
  float a;
  a = LIDARXYg[start][1]-m*LIDARXYg[start][0];
  for (int i = start; i < end; ++i)
  {
```

```
    double d = abs(m*LIDARXYg[i][0]-LIDARXYg[i][1]+a)/sqrt(m*m+1);
    if (d > dmax)
    {
      idx = i;
      dmax = d;
    }
  }
  // Seguimos un procedimiento recursivo
  if (dmax > epsilon)
  {
    Puntos.add(idx);
    Segmenta1(start, idx);
    Segmenta1(idx, end);
  }
}

//Dibuja con los datos de los puntos vértices los puntos y segmentos

void Segmenta2(int start, int end)
{
   for (int i=start;i<end;i++)
  {
    stroke(255,0,0);
    strokeWeight(5);
    int i1=Puntos.get(i);
    point((LIDARXYg[i1][0]+11000)*Esc,(LIDARXYg[i1][1]+4000)*Esc);
  }
}

//Hace las funciones de segmentación para los diferentes clusters
encontrados

void SegmentaClust()
{
  float SegmentosInt[][]=new float[500][4]; //Guardamos los segmentos
en la forma (X1,Y1,X2,Y2)

  Segmenta1(0,Clust.get(0));
  for (int i=1;i<Clust.size();i++)
  {
    Segmenta1(Clust.get(i-1),Clust.get(i));
  }
  int a=Clust.get((Clust.size()-1));
  Segmenta1(a,359);
   for(int i=0;i<(Puntos.size()-1);i++) //Ordenamos el vector puntos
con las posiciones de los vertices o extremos
  {                                     //de los segmentos extraidos
    for(int j=i+1;j<Puntos.size();j++)
    {
      if(Puntos.get(i)>Puntos.get(j))
      {
        int variableauxiliar=Puntos.get(i);
        Puntos.set(i,Puntos.get(j));
        Puntos.set(j,variableauxiliar);
       }
     }
   }
   Puntos.add(359);
  Segmenta2(0,Puntos.size());
  int a1=0;
  int is=0;
```

```
  for (int i=0;i<(Puntos.size()-1);i++) //Construimos el vector de
segmentos
  {
    if ((Puntos.get(i+1))>=Clust.get(a1))
    {
        Segmentos[is][0]=Puntos.get(i);
        Segmentos[is][1]=Clust.get(a1)-1;

        is=is+1;
        Segmentos[is][0]=Clust.get(a1);
        Segmentos[is][1]=Puntos.get(i+1);

        is=is+1;
        a1=a1+1;
    }
    else
    {
      Segmentos[is][0]=Puntos.get(i);
      Segmentos[is][1]=Puntos.get(i+1);
      is=is+1;
    }
    if (a1==Clust.size())
    {
      Clust.add(360);
    }
  }
  is=is-1;

  for (int i=0;i<=is;i++)
  {
    if (Segmentos[i][0]!=Segmentos[i][1])           //Se eliminan
segmentos de un punto ya que dan problemas al calcular m
    {
      strokeWeight(1);
      SegmentosInt[i][0]=LIDARXYg[Segmentos[i][0]][0];
      SegmentosInt[i][1]=LIDARXYg[Segmentos[i][0]][1];
      SegmentosInt[i][2]=LIDARXYg[Segmentos[i][1]][0];
      SegmentosInt[i][3]=LIDARXYg[Segmentos[i][1]][1];

line((LIDARXYg[Segmentos[i][0]][0]+11000)*Esc,(LIDARXYg[Segmentos[i][0
]][1]+4000)*Esc,(LIDARXYg[Segmentos[i][1]][0]+11000)*Esc,(LIDARXYg[Seg
mentos[i][1]][1]+4000)*Esc);

      float m;
      m = (SegmentosInt[i][1]-SegmentosInt[i][3])/(SegmentosInt[i][0]-
SegmentosInt[i][2]);
      float a2;
      a2 = SegmentosInt[i][1]-m*SegmentosInt[i][0];
      float fi;
      fi=atan(a/(-a/m));
      if (a2<0 && fi<0)
      {
        fi=2*PI-abs(fi);
      }
      else
      {
        if (a2<0 && fi>0)
        {
          fi=fi+PI;
        }
        else
```

```
        {
          if (a2>0 && fi<0)
          {
          fi=PI+fi;
          }
        }
      }
      float r;
      r=(-a2/m)*sin(fi);

      if (sqrt(sq(SegmentosInt[i][0]-
SegmentosInt[i][2])+sq(SegmentosInt[i][1]-SegmentosInt[i][3]))>100)
      {
        Segmentos1[iaux][0]=fi;
        Segmentos1[iaux][1]=r;
        Segmentos1[iaux][2]=SegmentosInt[i][0];
        Segmentos1[iaux][3]=SegmentosInt[i][1];
        Segmentos1[iaux][4]=SegmentosInt[i][2];
        Segmentos1[iaux][5]=SegmentosInt[i][3];
        iaux=iaux+1;
      }
    }
  }
}

//Limpia las variables usadas solo por cada ciclo

void Limpia1()
{
  Puntos.clear();
  Clust.clear();
  iaux=0;
}

//Vacia el vector Segmentos1 en cada ciclo

void Limpia2()
{
  for(int i=0;i<Segmentos1.length;i++)
  {
    Segmentos1[i][0]=0;
    Segmentos1[i][1]=0;
    Segmentos1[i][2]=0;
    Segmentos1[i][3]=0;
  }
}

//Amplia el mapa de segmentos en función de lo que ve en cada toma de
datos

void AmpliaMapa()
{
  float tolfi=0.1;
  float tolr=500;
  println(iaux);
  for (int i=0;i<iaux;i++)
  {
    for (int j=0;j<Extension;j++)
    {
```

```
      if (SegmentosMapa[j][0]<(Segmentos1[i][0]+tolfi) &&
SegmentosMapa[j][0]>(Segmentos1[i][0]-tolfi)  //Si coinciden ángulo y
término independiente
      && SegmentosMapa[j][1]<(Segmentos1[i][1]+tolr) &&
SegmentosMapa[j][1]>(Segmentos1[i][1]-tolr))
      {
       if (((Segmentos1[i][0]*180/PI)>45 &&
(Segmentos1[i][0]*180/PI)<135) ||  ((Segmentos1[i][0]*180/PI)>225 &&
(Segmentos1[i][0]*180/PI)<315)) //Trabajamos con las x
        {
          if ((Segmentos1[i][0]*180/PI)>225) //Si están en el 3 o 4
cuadrante invertimos los valores
          {
            Segmentos1[i][2]=-1*Segmentos1[i][2]; Segmentos1[i][4]=-
1*Segmentos1[i][4];
            SegmentosMapa[j][2]=-1*SegmentosMapa[j][2];
SegmentosMapa[j][4]=-1*SegmentosMapa[j][4];
          }
          if ((Segmentos1[i][4]+300000)<SegmentosMapa[j][2] ||
(Segmentos1[i][2]-300000)>SegmentosMapa[j][4])  //Si no son
coincidentes, segmento nuevo
          {
            SegmentosMapa[Extension-1][0]=Segmentos1[i][0];
            SegmentosMapa[Extension-1][1]=Segmentos1[i][1];
            SegmentosMapa[Extension-1][2]=Segmentos1[i][2];
            SegmentosMapa[Extension-1][3]=Segmentos1[i][3];
            SegmentosMapa[Extension-1][4]=Segmentos1[i][4];
            SegmentosMapa[Extension-1][5]=Segmentos1[i][5];
            Extension=Extension+1;
            j=j+1;
          }
          else
          {
            if (Segmentos1[i][2]<SegmentosMapa[j][2])
            {
              SegmentosMapa[j][2]=Segmentos1[i][2];
            }
            if (Segmentos1[i][4]>SegmentosMapa[j][4])
            {
              SegmentosMapa[j][4]=Segmentos1[i][4];
            }
          }
        }
        else
//Trabajamos con las y
        {
          if ((Segmentos1[i][0]*180/PI)<45) //Si están en el 1 y 4
cuadrante invertimos los valores
          {
            Segmentos1[i][3]=-1*Segmentos1[i][3]; Segmentos1[i][5]=-
1*Segmentos1[i][5];
            SegmentosMapa[j][3]=-1*SegmentosMapa[j][3];
SegmentosMapa[j][5]=-1*SegmentosMapa[j][5];
          }
          if ((Segmentos1[i][5]+300000)<SegmentosMapa[j][3] ||
(Segmentos1[i][3]-300000)>SegmentosMapa[j][5])  //Si no son
coincidentes, segmento nuevo
          {
            SegmentosMapa[Extension-1][0]=Segmentos1[i][0];
            SegmentosMapa[Extension-1][1]=Segmentos1[i][1];
            SegmentosMapa[Extension-1][2]=Segmentos1[i][2];
```

```
            SegmentosMapa[Extension-1][3]=Segmentos1[i][3];
            SegmentosMapa[Extension-1][4]=Segmentos1[i][4];
            SegmentosMapa[Extension-1][5]=Segmentos1[i][5];
            Extension=Extension+1;
            j=j+1;
          }
          else
          {
            if (Segmentos1[i][3]<SegmentosMapa[j][3])
            {
              SegmentosMapa[j][3]=Segmentos1[i][3];
            }
            if (Segmentos1[i][5]>SegmentosMapa[j][5])
            {
              SegmentosMapa[j][5]=Segmentos1[i][5];
            }
          }
        }
        j=Extension;
       }
       else
       {
         if (j==(Extension-1))
//Si no coincide con ángulo y radio se añade como nuevo segmento
         {
           SegmentosMapa[Extension-1][0]=Segmentos1[i][0];
           SegmentosMapa[Extension-1][1]=Segmentos1[i][1];
           SegmentosMapa[Extension-1][2]=Segmentos1[i][2];
           SegmentosMapa[Extension-1][3]=Segmentos1[i][3];
           SegmentosMapa[Extension-1][4]=Segmentos1[i][4];
           SegmentosMapa[Extension-1][5]=Segmentos1[i][5];
           Extension=Extension+1;
           j=j+1;
         }
       }
     }
   }
   Extension=Extension-1;
}

void CorrigePos()
{
  float MapaAux[][]=new float[500][6];
  //Coge los segmentos del mapa y los deja de forma que se pueda
calcular la intersección con los puntos que debería ver
   for (int i=0;i<Extension;i++)
   {
     float x1=SegmentosMapa[i][2];
     float y1=SegmentosMapa[i][3];
     float x2=SegmentosMapa[i][4];
     float y2=SegmentosMapa[i][5];

     MapaAux[i][0]=sqrt(x1*x1+y1*y1);     //El radio es el mismo en
todas las locales
     MapaAux[i][1]=atan2(y1,x1)-Fi;       //El ángulo en las solidarias
al robot
     //Si el angulo es negativo pilla el equivalnte positivo
     if (MapaAux[i][1]<0) MapaAux[i][1]=2*PI+MapaAux[i][1];

     MapaAux[i][2]=sqrt(x2*x2+y2*y2);
     MapaAux[i][3]=atan2(y2,x2)-Fi;
```

```
      //Si el ángulo de fase es negativo pilla el equivalente positivo
      if (MapaAux[i][3]<0) MapaAux[i][3]=2*PI+MapaAux[i][3];

      //Si la recta es vertical hace cero los términos 4 y 5 (pendiente
y offset de la recta)

if(MapaAux[i][0]*cos(MapaAux[i][1])==MapaAux[i][2]*cos(MapaAux[i][3]))
      {
        MapaAux[i][4]=0;
        MapaAux[i][5]=0;
      }
      else
      {
        MapaAux[i][4]=(MapaAux[i][0]*sin(MapaAux[i][1])-
MapaAux[i][2]*sin(MapaAux[i][3]))/(MapaAux[i][0]*cos(MapaAux[i][1])-
MapaAux[i][2]*cos(MapaAux[i][3]));
        MapaAux[i][5]=MapaAux[i][0]*sin(MapaAux[i][1])-
MapaAux[i][4]*MapaAux[i][0]*cos(MapaAux[i][1]);
      }
    }

  //Calcula los puntos que deberíamos ver según la posición calculada
por la odometría
  float Puntos[][]=new float[360][3];
  float PuntosXY[][]=new float[360][2];
  float ang;
  float d,r,radio;
  boolean implicado=false;
  r=0;
  ang=0;
  d=PI/180;         //incremento de angulo 1°
  for (int i=0;i<360;i++)
  {
    radio=100000;
    for (int j=0;j<Extension;j++)
    {
      float t1,t2;     //Fases menor y mayor de los extremos
      t2=MapaAux[j][3];
      t1=MapaAux[j][1];
      if (MapaAux[j][1]>MapaAux[j][3])
        {
          t1=MapaAux[j][3];
          t2=MapaAux[j][1];
        }
      implicado=(((t2-t1)>PI) && ((0<=ang && ang<=t1)||(t2<=ang &&
ang<=2*PI))) || ((t2-t1)<=PI && t1<=ang && ang<=t2);

      //Si el radio corta con el segmento.
      if (implicado==true)
      Puntos[i][2]=1;
      {
        //Si el segmento está alineado con el radio
        if (ang==MapaAux[j][1] && ang==MapaAux[j][3])
          {
            r=MapaAux[j][0];
            if (MapaAux[j][2]<r)
              r=MapaAux[j][2];
          }
        else
          {
            //Si la recta es vertical
```

```
            if ((MapaAux[j][4]==MapaAux[j][5]) && (MapaAux[j][4]==0))
                r=MapaAux[j][0]*cos(MapaAux[j][1])/cos(ang);
            else
                r=MapaAux[j][5]/(sin(ang)-MapaAux[j][4]*cos(ang));
          }
      }
      if (r<0) r=-r;
      if (r<radio)
        radio=r;
    }

    Puntos[i][0]=ang;
    Puntos[i][1]=radio;

    PuntosXY[i][0]= Puntos[i][1]*cos(Puntos[i][0]);
    PuntosXY[i][1]= Puntos[i][1]*sin(Puntos[i][0]);
    ang=ang+d;
  }

  n=0; sumX=0; sumY=0; sumx=0; sumy=0; sumXY=0; sumxX=0; sumyY=0;
sumxY=0; sumyX=0;
  for (int i=0;i<360;i++)  //Calulamos los sumatorios
  {
    if (Puntos[i][2]>0.1)  //Solo comparamos en los ángulos que
veriamos punto en el mapa calculado
    {
      n=n+1;
      sumx=sumx+LIDARXYr[i][0];
      sumy=sumy+LIDARXYr[i][1];
      sumX=sumX+PuntosXY[i][0];
      sumY=sumY+PuntosXY[i][1];
      sumXY=sumXY+(PuntosXY[i][0]*PuntosXY[i][1]);
      sumxX=sumxX+(LIDARXYr[i][0]*PuntosXY[i][0]);
      sumyY=sumyY+(LIDARXYr[i][1]*PuntosXY[i][1]);
      sumxY=sumxY+(LIDARXYr[i][0]*PuntosXY[i][1]);
      sumyX=sumyX+(LIDARXYr[i][1]*PuntosXY[i][0]);

      int ntrial=400;
      float tolx=0.0001;
      float tolf=0.001;
      float x[]=new float[3];
      x[0]=Xr;
      x[1]=Yr;
      x[2]=Fir;

      mnewt(ntrial,x,3,tolx,tolf);

      for (int ir=0;ir<n;ir++)
      {
        println (ir, x[ir]);
      }
      if (n==3) println (x[2]*180/PI);     //OJO AL SIGNO DEL ANGULO
      if (n==4) println( atan2(x[2],x[3])*180/PI);
    }
  }
}

void mnewt(int ntrial, float x[], int n, float tolx, float tolf)
{
  int k,i;
  int indx[] = new int[n];
```

```
  float errx=0,errf=0,d=0;
  float p[]=new float[n];
  float fvec[]=new float[n];
  float fjac[][]=new float[n][n];
  for (k=0;k<ntrial;k++)
  {
    usrfun3(x,n,fvec,fjac);                    //User function
supplies function values at x in fvec and Jacobian matrix in fjac.
    errf=0;
    for (i=0;i<n;i++) errf += abs(fvec[i]);   //Check function
convergence.
    if (errf <= tolf)
    {
      println(k," intentos SALE POR errf",errx, errf);
      return;
    }
    for (i=0;i<n;i++)
      p[i] = -fvec[i];                    //Right-hand side of linear
equations.

    ludcmp(fjac,n,indx,d);                //Solve linear equations
using LU decomposition.

    lubksb(fjac,n,indx,p);

    errx=0.0;                            //Check root convergence.
    for (i=0;i<n;i++)                    //Update solution.
    {
      errx += abs(p[i]);
      x[i] += p[i];
    }
    if (errx <= tolx)
    {
      println(k," intentos SALE POR errx",errx, errf);
      for(i=0;i<n;i++) println (fvec[i], p[i]);
      return;
    }
  }
  println(k," intentos",errx, errf);
  return;
}

void usrfun3(float x[],int n,float fvec[],float fjac[][])
{
float A,B,C,Ce,D,E,A1,B1,C1,Ce1,D1,E1,
A2,B2,C2,D2,A3,B3,C3,D3,A4,B4,C4,D4,A5,B5,C5,D5;
//Definición de los coeficientes

//Las ecuaciones son:
//A+Bx+Csc+Dc*Es=0
//A1+B1y+C1sc+D1c+E1S=0
//A2*s^2+B2*s^2*y+C2*s^2*x+D2*s^2*x*y +
A3*c^2+B3*c^2*y+C3*c^2*x+D3*c^2*x*y + A4*s+B4*s*x+C4*s+D4*s*y +
A5*c+B5*c*y+C5*c+D5*c*x=0

//Damos valores a las constantes de las ecuaciones

  A=-sumX;    B=n; C=2*sumY; Ce=2*n; D=sumx;   E=sumy;
  A1=sumY; B1=n;C1=2*sumx; Ce1=2*n; D1=sumy; E1=sumx;
  A2=2*sumXY; B2=2*sumX; C2=2*sumY; D2=2*n;
  A3=A2;B3=B2; C3=C2; D3=D2;
```

```
  A4=sumxX; B4=sumx; C4=sumyY; D4=sumy;
  A5=sumxY; B5=sumx; C5=sumyX ;D5=sumy;
  float s=sin(x[2]);
  float c=cos(x[2]);

//Definimos las funciones

  fvec[0]=A+B*x[0]+C*s*c-Ce*s*c*x[1]+D*c+E*s;
  fvec[1]=A1+B1*x[1]+C1*s*c-Ce1*s*c*x[0]+D1*c+E1*s;
  fvec[2]=s*s*(A2+B2*x[1]+C2*x[0]+D2*x[0]*x[1]);
  fvec[2]=fvec[2]+c*c*(A3+B3*x[1]+C3*x[0]+D3*x[0]*x[1]);
  fvec[2]=fvec[2]+s*(A4+B4*x[0]+C4+D4*x[1]);
  fvec[2]=fvec[2]+c*(A5+B5*x[1]+C5+D5*x[0]);

//Definicion del Jacoviano

fjac[0][0]=B;
fjac[0][1]=0;
fjac[0][2]=C*c*c-C*s*s-D*s+E*c;

fjac[1][0]=0;
fjac[1][1]=B1;
fjac[1][2]=C1*c*c-C1*s*s-D1*s+E1*c;

fjac[2][0]=s*s*(C2+D2*x[1])+ c*c*(C3+D3*x[1]) +B4*s+ D5*c;
fjac[2][1]=s*s*(B2+D2*x[0])+ c*c*(B3+D3*x[0]) +D4*s+ B5*c;
fjac[2][2]=2*s*c*(A2+B2*x[1]+C2*x[0]+D2*x[0]*x[1]  -A3-B3*x[1]-
C3*x[0]-D3*x[0]*x[1]) + c*(A4+B4*x[0]+C4+D4*x[1]) -
s*(A5+B5*x[1]+C5+D5*x[0]);
}


void ludcmp(float a[][], int n, int indx[], float d)
{
  int i,imax=n,j,k;
  float big=0,dum,sum,temp;
  float vv[]=new float[n];     //vv stores the implicit scaling of
each row.
  d=1.0;                       //No row interchanges yet.
  for (i=0;i<n;i++)            //Loop over rows to get the implicit
scaling informabig=0.0; tion.
  {
    big=0;
    for (j=0;j<n;j++)
      if ((temp=abs(a[i][j])) > big) big=temp;
    if (big == 0.0)
      println("Singular matrix in routine ludcmp");
    else                  //No nonzero largest element.
    vv[i]=1.0/big;        //Save the scaling.
  }
  for (j=0;j<n;j++)       //This is the loop over columns of Crout's
method.
  {
    for (i=0;i<j;i++)     //This is equation (2.3.12) except for i =
j.
    {
      sum=a[i][j];
      for (k=0;k<i;k++)
        sum -= a[i][k]*a[k][j];
      a[i][j]=sum;
    }
```

```
    big=0.0;                //Initialize for the search for largest
pivot element.
    for (i=j;i<n;i++)       //This is i = j of equation (2.3.12) and i
= j+1. . .N
    {
      sum=a[i][j];          //of equation (2.3.13).
      for (k=0;k<j;k++)
        sum -= a[i][k]*a[k][j];
      a[i][j]=sum;
      if ( (dum=vv[i]*abs(sum)) >= big) //Is the figure of merit for
the pivot better than the best so far?
      {
        big=dum;
        imax=i;
      }
    }
    if (j != imax) //Do we need to interchange rows?
    {
      for (k=0;k<n;k++)  //Yes, do so...
      {
        dum=a[imax][k];
        a[imax][k]=a[j][k];
        a[j][k]=dum;
      }
      d = -d;           //...and change the parity of d.
      vv[imax]=vv[j]; //Also interchange the scale factor.
    }
    indx[j]=imax;
    if (a[j][j] == 0.0)
      a[j][j]= 1.0e-20;  //If the pivot element is zero the matrix is
singular (at least to the precision of the
                         //algorithm). For some applications on
singular matrices, it is desirable to substitute
                         //TINY for zero.
    if (j != n) //Now, finally, divide by the pivot element.
    {
      dum=1.0/(a[j][j]);
      for (i=j+1;i<n;i++) a[i][j] *= dum;
    }
  } //Go back for the next column in the reduction.
}

void lubksb(float a[][], int n, int indx[], float b[])
{
  int i,ii=0,ip,j;
  float sum;
  for (i=0;i<n;i++)
  {
    ip=indx[i];
    sum=b[ip];
    b[ip]=b[i];
    if (ii>0)
    for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
    else if (sum>0) ii=i; //A nonzero element was encountered, so from
now on we
                          //will b[i]=sum; have to do the sums in the
loop above.
  }
  for (i=n-1;i>=0;i--) //Now we do the backsubstitution, equation
(2.3.7).
  {
```

```
    sum=b[i];
    for (j=i+1;j<n;j++) sum -= a[i][j]*b[j];
    b[i]=sum/a[i][i]; //Store a component of the solution vector X.
  }
}
```

Código de las funciones para la selección de trayectorias

```
// Calcula la recta que une la posición del robot y el destino, la
dibuja y recalcula la m y la a de la recta y=mx+a que los une

void CalculaUnion()
{
  //line(Esc*X,Esc*Y,Esc*Xd,Esc*Yd);
  m=(Yd-Yr)/(Xd-Xr);
  a=(Yd-m*Xd);
}

void GiraDerecha(float Grados)
{
  Fi=Fi+Grados;
}

void GiraIzquierda(float Grados)
{
  Fi=Fi-Grados;
}

void Recto()
{
  Vd=200;
  Vi=200;
}

void Orientarse()
{
  orient=atan(abs(Xd-X)/abs(Yd-Y));
  if (Xd>X && Yd>Y)
  {
    orient=2*PI-orient;
  }
  else
  {
    if (Xd<X && Yd<Y)
    {
      orient=PI-orient;
    }
    else
    if (Xd>X && Yd<Y)
    {
      orient=PI+orient;
    }
  }
  if (Fi<(orient+0.1) && Fi>(orient-0.1))
  {
    Recto();
  }
```

```
  else
  {
    Fi=orient;
  }
}


void EvitarObstaculo()
{
  float DisMin=LIDAR[1][1];
  float Ind=0;
  for (int i=1;i<360;i++)
  {
    if (DisMin>LIDAR[i][1])
    {
      DisMin=LIDAR[i][1];
      Ind=i;
    }
  }

  GiraDerecha((Ind+2)*PI/180);
  dibuja_casa();
  dibuja_robot();
  calcula_casa();
  calcula_lidar();
  CalculaPos();
  CalculaUnion();
  BordearObstaculo();
}

void BordearObstaculo()
{
  float DisUmbral=300+250;
  float DisMin=200+250;
  while (LIDAR[1][1]<DisUmbral)
  {
    if (LIDAR[90][1]>DisMin)
    {
      Recto();
    }
    else
    {
      GiraDerecha(PI/2);
    }
    dibuja_casa();
    dibuja_robot();
    calcula_casa();
    calcula_lidar();
    CalculaPos();
    CalculaUnion();
  }

  for (int i=0;i<31;i++)
  {
    Recto();
    dibuja_casa();
    dibuja_robot();
    calcula_casa();
    calcula_lidar();
    CalculaPos();
    CalculaUnion();
```

```
  }

  GiraIzquierda(PI/2);
  dibuja_casa();
  dibuja_robot();
  calcula_casa();
  calcula_lidar();
  CalculaPos();
  CalculaUnion();

  for (int i=0;i<76;i++)
  {
    Recto();
    dibuja_casa();
    dibuja_robot();
    calcula_casa();
    calcula_lidar();
    CalculaPos();
    CalculaUnion();
  }
}

void CalculaTrayectoria()
{
  int DisMin=200+250;
  if (LIDAR[90][1]>DisMin)
  {
    Orientarse();
  }
  else
  {
    EvitarObstaculo();
  }
}
```