Capítulo 6. CÓDIGO EN C++

6.1. main.cpp

```
/*!
 *      @file  main.cpp
 *    @author  Irene Santos Velázquez, isantos.ext@catec.aero
 *    @brief   Give positions of detected objects.
 *
 *  @internal
 *    Created  02/05/2013
 */

#define WIN32_LEAN_AND_MEAN

// Headers to access to LIDAR device
#include <IbeoAPI/Serializable.hpp>
#include <IbeoAPI/Mutexed.hpp>
#include <IbeoAPI/Scan.hpp>
#include <IbeoAPI/Laserscanner.hpp>
#include <IbeoAPI/ibeointern/IbeoLUX3.hpp>
#include <IbeoAPI/Manager.hpp>
#include <IbeoAPI/EventMonitor.hpp>
#include <IbeoAPI/SystemSignalHandler.hpp>
#include <IbeoAPI/ibeointern/ibeointernfactory.hpp>
#include <boost/bind.hpp>

#include <iostream>
#include <cassert>
#include "background.h"
#include <fstream>

using namespace std;

// #####   FUNCTION DEFINITIONS  -  LOCAL TO THIS SOURCE FILE
#######
bool messageHandler (const Serializable& msg, UINT8);
bool processScan();
```

```
////////////////////////////////
// IBEO global variables      //
////////////////////////////////
EventMonitor g_events; // object for thread synchronization

// Define the events used for synchronization
const EventMonitor::Mask g_newScanDataEvent = g_events.newEvent();
const EventMonitor::Mask g_cancelEvent =      g_events.newEvent();

// Global thread-safe Scan buffer with associated mutex
typedef Mutexed<Scan> MutexedScan;
static MutexedScan g_scan;

////////////////////////////////
// Other stuffs               //
////////////////////////////////
static background *bckModel(NULL);

int main()
{
  const string ipText="192.168.0.1";

  ////////////////////////////////
  // Connect to LIDAR
  ////////////////////////////////
  bool keep_running = true;
  bool first_time=true;

   // Ctrl-C handler
  SystemSignalHandler systemHandlers(&g_events, g_cancelEvent,
"connectscanner");

  // The IbeoAPI manager needs a configuration file from an input
stream
  std::string configstring(
      "BEGIN IbeoLUX \n"
      "    DeviceID = 120\n"
      "    IPAddress = " + ipText + "\n"
      "END\n");

  // Register the devices; needed for connecting to an IbeoLUX
  Manager myManager;
  ibeo::intern::registerFactory (myManager);

  std::istringstream config_istream(configstring);
  if (!myManager.initSystem (config_istream))
  {
    cerr << "Manager::initSystem() failed. Exiting." << endl;
    return EXIT_FAILURE;
  }

  // Set message handlers (callbacks)
  DataTypeFilter filter;
  filter.add(ibeo::DataTypeScanPointList);
  myManager.registerMessageHandler (messageHandler, filter);

  // Access the first Ibeo Laserscanner
  IbeoLUX3* scanner_p = myManager.getFirst<IbeoLUX3>();
  if (scanner_p == NULL)
  {
```

```
    cerr << "No Laserscanner device exists in the manager.
Exiting.\n";
    return EXIT_FAILURE;
  }

  cout << "Scan frequency: " << scanner_p->getScanFrequency() << endl;
  keep_running &= scanner_p->measure();
  cout << "Starting measures...\n";

  while(keep_running)
  {
    // Wait for any event signalled by our message handler.
    EventMonitor::Mask ev = g_events.wait
(boost::posix_time::seconds(10),
      EventMonitor::TimeOutEvent | g_newScanDataEvent |
g_cancelEvent);

    // Which event did occur?
    if (ev == EventMonitor::TimeOutEvent) // "==" instead of bit test
(zero!)
    {
      cout << "Timeout occurred. Terminating." << endl;
      keep_running = false; // terminate event loop
    }
    if (ev & g_newScanDataEvent)
    {
            // A scan has arrived, hence call the example processing
function
            if (first_time == true)
            {
                  cout << "Background model will be generated\n";
                  bckModel=new background(g_scan.data);  // Se
inicializa el background
                  first_time = false;
            }
            else
                  keep_running = processScan();  // Se escanean las
siguientes nubes de puntos
    }
    if (ev & g_cancelEvent)
    {
      cout << "Terminated by Ctrl+C event\n";
      keep_running = false; // terminate event loop
    }
  }

  cout << endl;

  if (scanner_p)
    scanner_p->idle();

  delete bckModel;

  return EXIT_SUCCESS;
}


// This message handler function is used to receive the messages from
// the serializer of the IbeoAPI
bool messageHandler (const Serializable& msg, UINT8 )
{
```

```
  ofstream outFile;
  outFile.open("prueba.out",std::ios_base::app);

  assert(msg.getDataType() == ibeo::DataTypeScanPointList);
  const Scan& scan = dynamic_cast<const Scan&>(msg);
  // Copy the received scan to the thread-safe local buffer
  bool copiedSuccessfully = g_scan.tryCopy (scan);
  if (copiedSuccessfully)
  {
    g_events.signal (g_newScanDataEvent);       // Send event
  }
  else
  {
    cout << "Scan buffer locked - cannot copy Laserscanner's Scan to
buffer\n";
      outFile << "Scan buffer locked - cannot copy Laserscanner's Scan
to buffer\n";
  }
  return true;
}


bool processScan()
{
  ofstream outFile;
  outFile.open("prueba.out",std::ios_base::app);
  // Lock the scan buffer to synchronize the access.
  MutexedScan::scoped_try_lock criticalSection (g_scan.mutex);
  bool lockObtained = criticalSection; // true if we acquired the lock

  if (lockObtained)
  {
        //cout << "Background model generated. Starting to detect
objects\n";
        cout << "Background model generated. Detecting objects\n";
        outFile << "Background model generated. Detecting
objects\n\n";
        bckModel->detect(g_scan.data);  // Se empieza a detectar
objetos
  }
  return true;
}
```

6.2. background.cpp

```
/*!
 *      @file  background.cpp
 *    @author  Irene Santos Velázquez, isantos.ext@catec.aero
 *
 *  @internal
 *    Created  02/05/2013
 */

#include "background.h"
#include "conversor.h"

#include <cmath>
#include <iostream>
```

```
#include <vector>
#include <fstream>
#include <IbeoAPI/Scan.hpp>

using namespace std;

#define PI 3.14159265
#define TAM_MAX 2000  // Se impone un tamaño máximo para que no haya
problemas de acceso a memoria fuera de rango
#define TH 0.4  // Umbral
#define ALFA 0.05
#define TAM_MIN 0.2 // Tamaño mínimo de objeto a detectar (20cm)
#define MAX_CEROS 10  // Una vez detectado el inicio de un obj,
MAX_CEROS es el número de máximo de pixeles de un canal que no forman
parte de un obj pero están entre su principio y fin
#define MAX_OBJECTS 10  // Es el número máximo de objetos que se
podrán detectar por canal
#define DIF_PIXEL 0.4  // Máxima diferencia entre pixeles de un mismo
objeto
#define REP_OBJ 3 // Número de veces seguidas que debe estar el objeto
presente para considerarse como objeto
#define MIN_PIX_OBJ 2 // Número mínimo de pixeles de que está formado
un objeto (si no son al menos 2, no se podría calcular el tamaño del
objeto)
#define DIF_OBJ 0.5  // Máxima diferencia de distancia entre el mismo
objeto en instantes consecutivos

// Constantes para la conversión de coordenadas
#define LAT_REF 37.0  // Punto de referencia del dispositivo en
coordenadas LLA (en grados)
#define LON_REF -5.0
#define H_REF 0
#define PITCH 0.0  // Pitch del dispositivo (en grados)
#define YAW 0.0  // Yaw del dispositivo (en grados)
#define ROLL 0.0  // Roll del dispositivos (en grados)
#define deg2rad PI/180.0

static conversor *Coord_LLA(NULL);


background::background(const Scan &scan)
{
      /*
      ofstream outFile;
      outFile.open("prueba.out",std::ios_base::app);
      */

      num_objects_ant.resize(4);  // Número de objetos en cada canal
en el instante anterior al actual

      // Indica los instantes de tiempo seguidos en que ha aparecido
un objeto en cada canal
      times_obj_c0.resize(MAX_OBJECTS);  // Se impone que no detecte
más de MAX_OBJECTS objetos en un mismo canal
      times_obj_c1.resize(MAX_OBJECTS);
      times_obj_c2.resize(MAX_OBJECTS);
      times_obj_c3.resize(MAX_OBJECTS);

      // Distancia media de los objetos en el instante anterior al
actual
```

```
	dist_obj_ant_c0.resize(MAX_OBJECTS);  // Se impone que no
detecte más de MAX_OBJECTS objetos en un mismo canal
	dist_obj_ant_c1.resize(MAX_OBJECTS);
	dist_obj_ant_c2.resize(MAX_OBJECTS);
	dist_obj_ant_c3.resize(MAX_OBJECTS);

	LLA.resize(3);  // LLA es un vector de 3 elementos que almacena
la posición del objeto en coordenadas LLA
	Coord_LLA=new conversor(LON_REF*deg2rad, LAT_REF*deg2rad, H_REF,
PITCH*deg2rad, YAW*deg2rad, ROLL*deg2rad);

	unsigned int points = static_cast<unsigned
int>(scan.getNumPoints());  // Número de puntos de la nube de puntos
inicial

	//outFile << "Background: (i,x,y,dist,hAngle_deg,channel) \n";
	//outFile << "Points: " << points << "\n";

	if (points>TAM_MAX)
		points=TAM_MAX;

	set_bg(points);  // Se inicializa con un tamaño points
	set_ob();  // guarda la nube de puntos del objeto

	const Scan::PointList& pl = scan.getPointList();  //
getPointList() returns the list of scan points (read only)

	for (unsigned int i=0; i<points; i++)
	{
		const ScanPoint& p = pl[i];
		add_bg(i, p.getX(), p.getY(), p.getZ(), p.getDist(),
p.getHAngle(), p.getChannel());
		add_bg_ini(i, p.getX(), p.getY(), p.getZ(), p.getDist(),
p.getHAngle(), p.getChannel());
		//outFile << "i: " << i << "     x: " << x[i] << "     y: "
<< y[i] << "     dist: " << dist[i] << "     hAngle_deg: " <<
hAngle_deg[i] << "     channel: " << channel[i] << "\n";
	}
	//outFile << "\n";
}


void background::set_bg(const unsigned int n)
{
	// Background
	x_bg.resize(n);
	y_bg.resize(n);
	z_bg.resize(n);
	dist_bg.resize(n);
	hAngle_bg_deg.resize(n);
	channel_bg.resize(n);
	// Background inicial
	x_bg_ini.resize(n);
	y_bg_ini.resize(n);
	z_bg_ini.resize(n);
	dist_bg_ini.resize(n);
	hAngle_bg_deg_ini.resize(n);
	channel_bg_ini.resize(n);
}
```

```
void background::add_bg(const size_t i, const float _x, const float
_y,
  const float _z, const float d, const float ha, const unsigned int c)
{
      x_bg[i]=_x;
      y_bg[i]=_y;
      z_bg[i]=_z;
      dist_bg[i]=d;
      hAngle_bg_deg[i]=ha*180/PI;
      channel_bg[i]=c;
}


void background::add_bg_ini(const size_t i, const float _x, const
float _y,
  const float _z, const float d, const float ha, const unsigned int c)
{
      x_bg_ini[i]=_x;
      y_bg_ini[i]=_y;
      z_bg_ini[i]=_z;
      dist_bg_ini[i]=d;
      hAngle_bg_deg_ini[i]=ha*180/PI;
      channel_bg_ini[i]=c;
}


void background::set_ob()
{
      // Objeto final auxiliar
      x_obj_aux.empty();
      y_obj_aux.empty();
      z_obj_aux.empty();
      dist_obj_aux.empty();
      hAngle_obj_deg_aux.empty();

      // Objeto final
      x_obj.empty();
      y_obj.empty();
      z_obj.empty();
      dist_obj.empty();
      ha_obj.empty();
      tamaño_obj.empty();
}


void background::detect(const Scan &scan)
{
      ofstream outFile;
      outFile.open("prueba.out", std::ios_base::app);


      const Scan::PointList& frames = scan.getPointList();
      unsigned int puntos = static_cast<unsigned
int>(scan.getNumPoints());
      unsigned int points = x_bg.size();
      /*
      outFile << "Detecting \n";
      outFile << "Puntos: " << puntos << "\n";
      */

      if (puntos>TAM_MAX)
```

```
            puntos=TAM_MAX;

      clear_all_possible();  // Limpia los vectores de indices,
comparacion y los relacionados con los objetos

      /*
      // Para ver los resultados en el fichero
      for (unsigned int ii=0; ii<puntos; ii++)
            outFile << "i: " << ii << "     x: " << frames[ii].getX()
<< "     y: " << frames[ii].getY() << "     dist: " <<
frames[ii].getDist() << "     hAngle_deg: " <<
frames[ii].getHAngle()*180/PI << "     channel: " << unsigned
int(frames[ii].getChannel()) << "\n";
      outFile << "\n";
      */
/*
      // Para file
      outFile << "Background: (i,x,y,dist,hAngle_deg,channel) \n";
      for (unsigned int i=0; i<points; i++)
            outFile << "i: " << i << "     x: " << x[i] << "     y: "
<< y[i] << "     dist: " << dist[i] << "     hAngle_deg: " <<
hAngle_deg[i] << "     channel: " << channel[i] << "\n";
*/

      // Indices para recorrer bg y nube actual teniendo en cuenta que
ambas no tienen los mismos números de pixeles
      unsigned int ultii=0;
      unsigned int indexi=0;

      std::vector<unsigned int> num_objects;  // Indica el número de
objetos en cada canal. Cada elemento corresponde a un canal.
      num_objects.resize(4);

      for (unsigned int i=0; i<points; i++)  // Recorre el background
(x,y,z,dist, hAngle_deg, channel)
      {
            for (unsigned int ii=ultii; ii<puntos; ii++)  // Recorre la
nube de puntos actual
            {
                  const ScanPoint& pixel = frames[ii];  // pixel
contiene toda la información (x,y,z,dist,hAngle,channel...) de cada
elemento de frames
                  float pixel_hdeg=pixel.getHAngle()*180/PI;
                  unsigned int canal=pixel.getChannel();

                  unsigned int indexi=0;

                  if (pixel_hdeg == hAngle_bg_deg[i] && canal ==
channel_bg[i])  // Estamos en el mismo bin (mismo hAngle y mismo
canal)
                  {
                        bool bg=false;  // Para indicar si alguno de
esos puntos es background
                        for (unsigned int kk=0; kk<3; kk++)
                        {
                              indexi=0;
                              for (unsigned int k=0; k<3; k++)  //
Puede que haya varios pixeles para el mismo canal y hAngle (nunca son
mas de 3 echos)
                              {
```

```
                                    if (k==0 && abs(dist_bg[i]-
frames[ii+k].getDist())<=TH)  // Es background
                                    {

      x_bg[i]=ALFA*frames[ii].getX()+(1-ALFA)*x_bg[i];

      y_bg[i]=ALFA*frames[ii].getY()+(1-ALFA)*y_bg[i];

      z_bg[i]=ALFA*frames[ii].getZ()+(1-ALFA)*z_bg[i];

      dist_bg[i]=ALFA*frames[ii].getDist()+(1-ALFA)*dist_bg[i];
                                          if (bg==false) // Se guarda
un solo 0 en comparacion para el mismo hAngle y channel
                                          {
                                                switch (canal)
                                                {
                                                case 0:

      comparacion_c0.push_back(0);
                                                      break;
                                                case 1:

      comparacion_c1.push_back(0);
                                                      break;
                                                case 2:

      comparacion_c2.push_back(0);
                                                      break;
                                                default:

      comparacion_c3.push_back(0);
                                                      break;
                                                }
                                          }
                                          bg=true;
                                    }

                                    if (ii+k<puntos-1 &&
frames[ii].getChannel() == frames[ii+k+1].getChannel() &&
frames[ii].getHAngle()*180/PI == frames[ii+k+1].getHAngle()*180/PI)
// Hay otro pixel con el mismo hAngle y canal
                                    {
                                          indexi++;
                                          if (abs(dist_bg[i]-
frames[ii+k+1].getDist())<=TH)  // Es background
                                          {

      x_bg[i]=ALFA*frames[ii+k+1].getX()+(1-ALFA)*x_bg[i];

      y_bg[i]=ALFA*frames[ii+k+1].getY()+(1-ALFA)*y_bg[i];

      z_bg[i]=ALFA*frames[ii+k+1].getZ()+(1-ALFA)*z_bg[i];

      dist_bg[i]=ALFA*frames[ii+k+1].getDist()+(1-ALFA)*dist_bg[i];
                                                if (bg==false) // Se
guarda un solo 0 en comparacion para el mismo hAngle y channel
                                                {
                                                      switch (canal)
                                                      {
                                                      case 0:
```

```
comparacion_c0.push_back(0);
                                                        break;
                                                    case 1:

comparacion_c1.push_back(0);
                                                        break;
                                                    case 2:

comparacion_c2.push_back(0);
                                                        break;
                                                    default:

comparacion_c3.push_back(0);
                                                        break;
                                                    }
                                                }
                                                bg=true;
                                            }
                                        }
                                        else  // Los siguientes pixeles no
tienen el mismo hAngle y canal que el anterior
                                        {
                                            if (bg == false &&
(i>=points-1 || channel_bg[i]!=channel_bg[i+1]))  // Si algún pixel
(con mismo canal y hAngle) es bg, entonces se considera bg. Solo será
objeto, en caso de que todos los pixeles (con mismo canal y hAngle)
sean objeto (bg==false)
                                            {
                                                switch (canal)
                                                {
                                                case 0:

comparacion_c0.push_back(1);

index_ob_c0.push_back(ii);  // Indice del pixel actual

index_bg_c0.push_back(i);  // Indice del pixel bg
correspondiente
                                                    break;
                                                case 1:

comparacion_c1.push_back(1);

index_ob_c1.push_back(ii);

index_bg_c1.push_back(i);
                                                    break;
                                                case 2:

comparacion_c2.push_back(1);

index_ob_c2.push_back(ii);

index_bg_c2.push_back(i);
                                                    break;
                                                default:

comparacion_c3.push_back(1);

index_ob_c3.push_back(ii);
```

```
		index_bg_c3.push_back(i);
                                                                        break;
                                                                }
                                                        }
                                                        break;
                                                }
                                        }

                                        if (i<points-1 &&
channel_bg[i]==channel_bg[i+1] &&
hAngle_bg_deg[i]==hAngle_bg_deg[i+1])
                                                i++;
                                        else
                                                break;
                                }
                                ii+=indexi;
                                ultii=ii+1;

                                break;
                        }
                        else
                                if (frames[ii].getHAngle()*180/PI <
hAngle_bg_deg[i])  // No hay más pixeles con el mismo hA en el
background (así se evita tener que recorrer toda la nube actual entera
buscando un hA que no se va a encontrar)
                                        break;
                }
        }


        // Recorremos el vector de resultados para determinar si los
pixeles distintos a bg (que están a uno) son realmente objetos
        for (unsigned int c=0; c<4; c++)
        {
                float tam=0;
                unsigned int count_1=0;  // Contador de pixeles diferentes
                unsigned int count_0=0;  // Contador de huecos entre
pixeles diferentes
                unsigned int ind_ini; // Contiene la posición del indice
del primer pixel que puede ser objeto
                unsigned int j=0; // Lleva la cuenta de todos los unos
anteriores a la posición actual

                // Para detectar objetos cercanos (su separación es menor
de MAX_CEROS ceros)
                unsigned int count_1_2=0;  // Contador de unos
                unsigned int ind_ini_2;  // Contiene la posición del indice
del primer pixel que puede ser objeto

                std::vector<bool> imprimir_ob;  // Indica si un determinado
objeto ha cumplido todos los requisitos como para imprimir su posición
media (es decir, considerarse como objeto)
                imprimir_ob.resize(MAX_OBJECTS);

                switch (c)
                {
                case 0:

                        for (unsigned int k=0; k<comparacion_c0.size(); k++)
                        {
```

```
                        if (comparacion_c0[k]==1)  // el pixel actual
es distinto a bg
                        {
                              if (count_1==0) // Si es el primer pixel,
guardo la posición de su indice para identificarlo después
                                    ind_ini=j;
                              count_1++;  // Se aumenta el número de
pixeles diferentes
                              count_0=0;
                        }
                        else
                              if (count_1>0)  // Solo se cuentan los
huecos a partir de encontrar un pixel a 1 (es decir, diferente a bg)
                                    count_0++;

                        if (count_0>MAX_CEROS ||
k==comparacion_c0.size()-1)  // Cuando se encuentran MAX_CEROS huecos
seguidos, se da por terminado el objeto anterior
(k==comparacion_c0.size()-1 se pone por si el vector comparacion acaba
en 1 y por tanto los pixeles anteriores pueden ser objetos)
                        {
                              j+=count_1;
                              if (count_1>MIN_PIX_OBJ && calc_tamaño(c,
frames, ind_ini, count_1)>=TAM_MIN)  // Se considerará objeto si está
formado por al menos por MIN_PIX_OBJ pixeles y tiene un tamaño de al
menos TAM_MIN
                              {
                                    while (count_1>MIN_PIX_OBJ)
                                    {
                                          // Se comprueba si el objeto
sigue estando, o por el contrario sus pixeles son similares a los del
bg inicial
                                          if (check_obj_retirado(c,
frames, ind_ini, count_1, ind_ini_2, count_1_2)==false)
                                          {
                                                tam=calc_tamaño_fin();
                                                if (tam>=TAM_MIN &&
x_obj_aux.size()>MIN_PIX_OBJ)  // Si tras haber comprobado todos los
pixeles objeto, su tamaño es mayor que TAM_MIN y está formado por más
de MIN_PIX_OBJ, se calcula su posición
                                                {
                                                      mean();
                                                      num_objects[c]++;

      times_obj_c0[num_objects[c]-1]++;

                                                      if
(times_obj_c0[num_objects[c]-1]!=1 && abs(dist_obj.back()-
dist_obj_ant_c0[num_objects[c]-1])<=DIF_OBJ)
                                                      {
                                                            if
(times_obj_c0[num_objects[c]-1]>=REP_OBJ)
                                                            {

      imprimir_ob[num_objects[c]-1]=true;

      tamaño_obj.push_back(tam);
                                                            }
                                                      }
                                                      else
```

```
      times_obj_c0[num_objects[c]-1]=1;


      dist_obj_ant_c0[num_objects[c]-1]=dist_obj.back();

                                                      if
(imprimir_ob[num_objects[c]-1]==false)

      delete_calc_ob();  // Borra el ultimo calculo de la media del
objeto, porque no sirve
                                                            }
                                                            clear_ob_aux();
                                                      }
                                                      count_1=count_1_2;
                                                      ind_ini=ind_ini_2;
                                                      count_1_2=0;
                                                      if
(num_objects[c]>=MAX_OBJECTS)  // Si se alcazan el número máximo de
objetos detectados, se sale de todos los bucles que sirven para
almacenar y detectar el objeto y ya no detectará más objetos
                                                            break;
                                                }
                                          }
                                          count_1=0;
                                          count_0=0;
                                    }
                                    if (num_objects[c]>=MAX_OBJECTS)  // Si se
alcazan el número máximo de objetos detectados, se sale de todos los
bucles que sirven para almacenar y detectar el objeto y ya no
detectará más objetos
                                          break;
                        }

                        if (x_obj.size()!=0)
                        {
                              print_ob(c);
                              clear_ob();
                        }

                        break;

                  case 1:

                        for (unsigned int k=0; k<comparacion_c1.size(); k++)
                        {
                              if (comparacion_c1[k]==1)  // el pixel actual
es distinto a bg
                              {
                                    if (count_1==0) // Si es el primer pixel,
guardo la posición de su indice para identificarlo después
                                          ind_ini=j;
                                    count_1++;  // Se aumenta el número de
pixeles diferentes
                                    count_0=0;
                              }
                              else
                                    if (count_1>0)  // Solo se cuentan los
huecos a partir de encontrar un pixel a 1 (es decir, diferente a bg)
                                          count_0++;
```

```
                        if (count_0>MAX_CEROS ||
k==comparacion_c1.size()-1)  // Cuando se encuentran MAX_CEROS huecos
seguidos, se da por terminado el objeto anterior
(k==comparacion_c1.size()-1 se pone por si el vector comparacion acaba
en 1 y por tanto los pixeles anteriores pueden ser objetos)
                        {
                              j+=count_1;
                              if (count_1>MIN_PIX_OBJ && calc_tamaño(c,
frames, ind_ini, count_1)>=TAM_MIN)  // Se considerará objeto si está
formado por al menos por MIN_PIX_OBJ pixeles y tiene un tamaño de al
menos TAM_MIN
                              {
                                    while(count_1>MIN_PIX_OBJ)
                                    {
                                          // Se comprueba si el objeto
sigue estando, o por el contrario sus pixeles son similares a los del
bg inicial
                                          if (check_obj_retirado(c,
frames, ind_ini, count_1, ind_ini_2, count_1_2)==false)
                                          {
                                                tam=calc_tamaño_fin();
                                                if (tam>=TAM_MIN &&
x_obj_aux.size()>MIN_PIX_OBJ)
                                                {
                                                      mean();
                                                      num_objects[c]++;

      times_obj_c1[num_objects[c]-1]++;

                                                      if
(times_obj_c1[num_objects[c]-1]!=1 && abs(dist_obj.back()-
dist_obj_ant_c1[num_objects[c]-1])<=DIF_OBJ)
                                                      {
                                                            if
(times_obj_c1[num_objects[c]-1]>=REP_OBJ)
                                                            {

      imprimir_ob[num_objects[c]-1]=true;

      tamaño_obj.push_back(tam);
                                                            }
                                                      }
                                                      else

      times_obj_c1[num_objects[c]-1]=1;


      dist_obj_ant_c1[num_objects[c]-1]=dist_obj.back();

                                                      if
(imprimir_ob[num_objects[c]-1]==false)

      delete_calc_ob();  // Borra el ultimo calculo de la media del
objeto, porque no sirve
                                                }
                                                clear_ob_aux();
                                          }
                                          count_1=count_1_2;
                                          ind_ini=ind_ini_2;
                                          count_1_2=0;
```

```
                                    if
(num_objects[c]>=MAX_OBJECTS)  // Si se alcazan el número máximo de
objetos detectados, se sale de todos los bucles que sirven para
almacenar y detectar el objeto y ya no detectará más objetos
                                    break;
                              }
                        }
                        count_1=0;
                        count_0=0;
                  }
                  if (num_objects[c]>=MAX_OBJECTS)  // Si se
alcazan el número máximo de objetos detectados, se sale de todos los
bucles que sirven para almacenar y detectar el objeto y ya no
detectará más objetos
                        break;
            }

            if (x_obj.size()!=0)
            {
                  print_ob(c);
                  clear_ob();
            }

            break;

      case 2:

            for (unsigned int k=0; k<comparacion_c2.size(); k++)
            {
                  if (comparacion_c2[k]==1)  // el pixel actual
es distinto a bg
                  {
                        if (count_1==0) // Si es el primer pixel,
guardo la posición de su indice para identificarlo después
                              ind_ini=j;
                        count_1++;  // Se aumenta el número de
pixeles diferentes
                        count_0=0;
                  }
                  else
                        if (count_1>0)  // Solo se cuentan los
huecos a partir de encontrar un pixel a 1 (es decir, diferente a bg)
                              count_0++;

                  if (count_0>MAX_CEROS ||
k==comparacion_c2.size()-1)  // Cuando se encuentran MAX_CEROS huecos
seguidos, se da por terminado el objeto anterior
(k==comparacion_c2.size()-1 se pone por si el vector comparacion acaba
en 1 y por tanto los pixeles anteriores pueden ser objetos)
                  {
                        j+=count_1;
                        if (count_1>MIN_PIX_OBJ && calc_tamaño(c,
frames, ind_ini, count_1)>=TAM_MIN)  // Se considerará objeto si está
formado por al menos por MIN_PIX_OBJ pixeles y tiene un tamaño de al
menos TAM_MIN
                        {
                              while (count_1>MIN_PIX_OBJ)
                              {
                                    // Se comprueba si el objeto
sigue estando, o por el contrario sus pixeles son similares a los del
bg inicial
```

```
                                        if (check_obj_retirado(c,
frames, ind_ini, count_1, ind_ini_2, count_1_2)==false)
                                        {
                                              tam=calc_tamaño_fin();
                                              if (tam>=TAM_MIN &&
x_obj_aux.size()>MIN_PIX_OBJ)
                                              {
                                                    mean();
                                                    num_objects[c]++;

      times_obj_c2[num_objects[c]-1]++;

                                                    if
(times_obj_c2[num_objects[c]-1]!=1 && abs(dist_obj.back()-
dist_obj_ant_c2[num_objects[c]-1])<=DIF_OBJ)
                                                    {
                                                          if
(times_obj_c2[num_objects[c]-1]>=REP_OBJ)
                                                          {

      imprimir_ob[num_objects[c]-1]=true;

      tamaño_obj.push_back(tam);
                                                          }
                                                    }
                                                    else

      times_obj_c2[num_objects[c]-1]=1;


      dist_obj_ant_c2[num_objects[c]-1]=dist_obj.back();

                                                    if
(imprimir_ob[num_objects[c]-1]==false)

      delete_calc_ob();  // Borra el ultimo calculo de la media del
objeto, porque no sirve
                                              }
                                              clear_ob_aux();
                                        }
                                        count_1=count_1_2;
                                        ind_ini=ind_ini_2;
                                        count_1_2=0;
                                        if
(num_objects[c]>=MAX_OBJECTS)  // Si se alcazan el número máximo de
objetos detectados, se sale de todos los bucles que sirven para
almacenar y detectar el objeto y ya no detectará más objetos
                                        break;
                                  }
                            }
                            count_1=0;
                            count_0=0;
                      }
                      if (num_objects[c]>=MAX_OBJECTS)  // Si se
alcazan el número máximo de objetos detectados, se sale de todos los
bucles que sirven para almacenar y detectar el objeto y ya no
detectará más objetos
                            break;
                }

                if (x_obj.size()!=0)
```

```
                  {
                        print_ob(c);
                        clear_ob();
                  }

                  break;

            default:

                  for (unsigned int k=0; k<comparacion_c3.size(); k++)
                  {
                        if (comparacion_c3[k]==1)  // el pixel actual
es distinto a bg
                        {
                              if (count_1==0) // Si es el primer pixel,
guardo la posición de su indice para identificarlo después
                                    ind_ini=j;
                              count_1++;  // Se aumenta el número de
pixeles diferentes
                              count_0=0;
                        }
                        else
                              if (count_1>0)  // Solo se cuentan los
huecos a partir de encontrar un pixel a 1 (es decir, diferente a bg)
                                    count_0++;

                        if (count_0>MAX_CEROS ||
k==comparacion_c3.size()-1)  // Cuando se encuentran MAX_CEROS huecos
seguidos, se da por terminado el objeto anterior
(k==comparacion_c3.size()-1 se pone por si el vector comparacion acaba
en 1 y por tanto los pixeles anteriores pueden ser objetos)
                        {
                              j+=count_1;
                              if (count_1>MIN_PIX_OBJ && calc_tamaño(c,
frames, ind_ini, count_1)>=TAM_MIN)  // Se considerará objeto si está
formado por al menos por MIN_PIX_OBJ pixeles y tiene un tamaño de al
menos TAM_MIN
                              {
                                    while (count_1>MIN_PIX_OBJ)
                                    {
                                          // Se comprueba si el objeto
sigue estando, o por el contrario sus pixeles son similares a los del
bg inicial
                                          if (check_obj_retirado(c,
frames, ind_ini, count_1, ind_ini_2, count_1_2)==false)
                                          {
                                                tam=calc_tamaño_fin();
                                                if (tam>=TAM_MIN &&
x_obj_aux.size()>MIN_PIX_OBJ)
                                                {
                                                      mean();
                                                      num_objects[c]++;

      times_obj_c3[num_objects[c]-1]++;

                                                      if
(times_obj_c3[num_objects[c]-1]!=1 && abs(dist_obj.back()-
dist_obj_ant_c3[num_objects[c]-1])<=DIF_OBJ)
                                                      {
                                                            if
(times_obj_c3[num_objects[c]-1]>=REP_OBJ)
```

```
                                                                {
      imprimir_ob[num_objects[c]-1]=true;

      tamaño_obj.push_back(tam);
                                                                }
                                                          }
                                                          else

      times_obj_c3[num_objects[c]-1]=1;


      dist_obj_ant_c3[num_objects[c]-1]=dist_obj.back();

                                                          if
(imprimir_ob[num_objects[c]-1]==false)

      delete_calc_ob();  // Borra el ultimo calculo de la media del
objeto, porque no sirve
                                                               }
                                                               clear_ob_aux();
                                                     }
                                                     count_1=count_1_2;
                                                     ind_ini=ind_ini_2;
                                                     count_1_2=0;
                                                     if
(num_objects[c]>=MAX_OBJECTS)  // Si se alcazan el número máximo de
objetos detectados, se sale de todos los bucles que sirven para
almacenar y detectar el objeto y ya no detectará más objetos
                                                          break;
                                               }
                                          }
                                          count_1=0;
                                          count_0=0;
                                     }
                                     if (num_objects[c]>=MAX_OBJECTS)  // Si se
alcazan el número máximo de objetos detectados, se sale de todos los
bucles que sirven para almacenar y detectar el objeto y ya no
detectará más objetos
                                          break;
                           }

                           if (x_obj.size()!=0)
                           {
                                print_ob(c);
                                clear_ob();
                           }

                           break;
                     }
      }

      // Se actualiza el número de instantes seguidos que ha aparecido
el objeto actual (en caso de no haber objeto o de haber menos objetos
que en el instante anterior se pone a cero)
      for (unsigned int k=0; k<4; k++)
      {
            bool possible_object=false;

            switch (k)
            {
```

```
            case 0:
                  for (unsigned int kk=0; kk<MAX_OBJECTS; kk++)
                        if (times_obj_c0[kk]>=REP_OBJ)
                        {
                              possible_object=true;  // Si alguno de
los elementos de possible_object_c0 es mayor o igual a REP_OBJ,
entonces la variable possible_object valdrá true
                              break;
                        }

                  if (num_objects[k]<num_objects_ant[k] ||
possible_object==true || num_objects[k]==0)
                  {
                        // Se vuelve a poner todos los elementos de
possible_object_c0 a cero
                        times_obj_c0.clear();
                        times_obj_c0.resize(MAX_OBJECTS);
                        dist_obj_ant_c0.clear();
                        dist_obj_ant_c0.resize(MAX_OBJECTS);
                  }
                  break;
            case 1:
                  for (unsigned int kk=0; kk<times_obj_c1.size(); kk++)
                        if (times_obj_c1[kk]>=REP_OBJ)
                        {
                              possible_object=true;  // Si alguno de
los elementos de possible_object_c1 es mayor o igual a REP_OBJ,
entonces la variable possible_object valdrá true
                              break;
                        }

                  if (num_objects[k]<num_objects_ant[k] ||
possible_object==true || num_objects[k]==0)
                  {
                        times_obj_c1.clear();
                        times_obj_c1.resize(MAX_OBJECTS);
                        dist_obj_ant_c1.clear();
                        dist_obj_ant_c1.resize(MAX_OBJECTS);
                  }
                  break;
            case 2:
                  for (unsigned int kk=0; kk<times_obj_c2.size(); kk++)
                        if (times_obj_c2[kk]>=REP_OBJ)
                        {
                              possible_object=true;  // Si alguno de
los elementos de possible_object_c2 es mayor o igual a REP_OBJ,
entonces la variable possible_object valdrá true
                              break;
                        }

                  if (num_objects[k]<num_objects_ant[k] ||
possible_object==true || num_objects[k]==0)
                  {
                        times_obj_c2.clear();
                        times_obj_c2.resize(MAX_OBJECTS);
                        dist_obj_ant_c2.clear();
                        dist_obj_ant_c2.resize(MAX_OBJECTS);
                  }
                  break;
            default:
                  for (unsigned int kk=0; kk<times_obj_c3.size(); kk++)
```

```
                        if (times_obj_c3[kk]>=REP_OBJ)
                        {
                              possible_object=true;  // Si alguno de
los elementos de possible_object_c3 es mayor o igual a REP_OBJ,
entonces la variable possible_object valdrá true
                              break;
                        }

                  if (num_objects[k]<num_objects_ant[k] ||
possible_object==true || num_objects[k]==0)
                  {
                        times_obj_c3.clear();
                        times_obj_c3.resize(MAX_OBJECTS);
                        dist_obj_ant_c3.clear();
                        dist_obj_ant_c3.resize(MAX_OBJECTS);
                  }
                  break;
            }
            num_objects_ant[k]=num_objects[k];
      }

      clear_all_possible();  // Limpia los vectores de indices y
comparacion
}


void background::print_ob(const unsigned int c)
{
      ofstream outFile;
      outFile.open("prueba.out", std::ios_base::app);

      cout << "Se han detectado " << x_obj.size() << " objetos en el
canal " << c << " \n";
      outFile << "Se han detectado " << x_obj.size() << " objetos en
el canal " << c << " \n";

      for (unsigned int k=0; k<x_obj.size(); k++)
      {
            cout << "Objeto " << k+1 << " detectado a X: " << x_obj[k]
<< "    e Y: " << y_obj[k] << "      Dist: " << dist_obj[k] << "
ÁnguloH: " << ha_obj[k] << "     Su tamaño es: " << tamaño_obj[k] <<
"\n";
            outFile << "Objeto " << k+1 << " detectado a X: " <<
x_obj[k] << "     Y: " << y_obj[k] << "      Dist: " << dist_obj[k] <<
"     ÁnguloH: " << ha_obj[k] << "     tamaño: " << tamaño_obj[k] <<
"\n";

            // Convirtiendo a coordenadas LLA
            LLA=Coord_LLA->newConversion(x_obj[k], y_obj[k], z_obj[k]);
// LLA=[lambda, phi, h]=[longitud, latitud, altura]  (en grados)
            cout << "LLA: (lon, lat, alt) = ( " << LLA[0] << ", " <<
LLA[1] << ", " << LLA[2] << ")\n ";
            outFile << "LLA: (lon, lat, alt) = ( " << LLA[0] << ", " <<
LLA[1] << ", " << LLA[2] << ")\n ";
      }

      outFile << "\n";
}
```

```
void background::keep_object(const float _x, const float _y, const
float _z, const float _d, const float _ha)
{
      x_obj_aux.push_back(_x);
    y_obj_aux.push_back(_y);
    z_obj_aux.push_back(_z);
    dist_obj_aux.push_back(_d);
      hAngle_obj_deg_aux.push_back(_ha);
}


bool background::check_obj_retirado(const unsigned int c, const
Scan::PointList& nube_act, const unsigned int i, const unsigned int
size, unsigned int& ind_ob_cerc, unsigned int& count1_ob_cerc)
{
      unsigned int count=0;
      bool obj_retirado=false;
      int ult_pixel_ob=-1;  // Si ult_pixel_ob ==-1, entonces todavía
no hay ningún pixel que se haya añadido como objeto

      switch (c)
      {
      case 0:
            for (unsigned int k=i; k<i+size; k++)
            {
                  if (abs(dist_bg_ini[index_bg_c0[k]]-
nube_act[index_ob_c0[k]].getDist())<=TH)
                  {
                        count++;
                        // Añadimos el pixel a bg

      x_bg[index_bg_c0[k]]=nube_act[index_ob_c0[k]].getX();

      y_bg[index_bg_c0[k]]=nube_act[index_ob_c0[k]].getY();

      z_bg[index_bg_c0[k]]=nube_act[index_ob_c0[k]].getZ();

      dist_bg[index_bg_c0[k]]=nube_act[index_ob_c0[k]].getDist();
                  }
                  else
                  {
                        // Si el primer pixel se sale de la media
respecto al segundo, no se tendrá en cuenta en el objeto
                        if (k==i &&
abs(nube_act[index_ob_c0[i]].getDist()-
nube_act[index_ob_c0[i+1]].getDist())>DIF_PIXEL)
                              count++;
                        else
                        {
                              // Si hay mucha diferencia entre la
distancia de los pixeles objeto, tampoco se añaden al objeto (esta
comprobación solo se hace en el caso de que algún pixel anterior se
haya considerado como objeto, es decir, se ha pasado por keep_object,
por eso se pone la condición ult_pixel_ob != -1)
                              if (k>i+1 && ult_pixel_ob!=-1 &&
abs(nube_act[index_ob_c0[k]].getDist()-
nube_act[index_ob_c0[ult_pixel_ob]].getDist())>DIF_PIXEL)
                              {
                                    count++;
                                    if (count1_ob_cerc==0)
                                    {
```

```
                                                count1_ob_cerc=size-(k-i);
                                                ind_ob_cerc=k;
                                          }
                                    }

                                    else
                                    {

      keep_object(nube_act[index_ob_c0[k]].getX(),
nube_act[index_ob_c0[k]].getY(), nube_act[index_ob_c0[k]].getZ(),
nube_act[index_ob_c0[k]].getDist(),
nube_act[index_ob_c0[k]].getHAngle()*180/PI);  // Se guardan los
pixeles restantes en x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux
para posteriormente calcular su tamaño y posición media
                                          ult_pixel_ob=k;
                                    }
                              }
                        }
                  }
                  if (count==size)
                        obj_retirado=true;  // El objeto se ha retirado
porque los pixeles ob son parecidos al bg inicial

                  break;

      case 1:
                  for (unsigned int k=i; k<i+size; k++)
                  {
                        if (abs(dist_bg_ini[index_bg_c1[k]]-
nube_act[index_ob_c1[k]].getDist())<=TH)
                        {
                              count++;
                              // Añadimos el pixel a bg

      x_bg[index_bg_c1[k]]=nube_act[index_ob_c1[k]].getX();

      y_bg[index_bg_c1[k]]=nube_act[index_ob_c1[k]].getY();

      z_bg[index_bg_c1[k]]=nube_act[index_ob_c1[k]].getZ();

      dist_bg[index_bg_c1[k]]=nube_act[index_ob_c1[k]].getDist();
                        }
                        else
                        {
                              // Si el primer pixel se sale de la media
respecto al segundo, no se tendrá en cuenta en el objeto
                              if (k==i &&
abs(nube_act[index_ob_c1[i]].getDist()-
nube_act[index_ob_c1[i+1]].getDist())>DIF_PIXEL)
                                    count++;
                              else
                              {
                                    // Si hay mucha diferencia entre la
distancia de los pixeles objeto, tampoco se añaden al objeto
                                    if (k>i+1 && ult_pixel_ob!=-1 &&
abs(nube_act[index_ob_c1[k]].getDist()-
nube_act[index_ob_c1[ult_pixel_ob]].getDist())>DIF_PIXEL)
                                    {
                                          count++;
                                          if (count1_ob_cerc==0)
                                          {
```

```
                                        count1_ob_cerc=size-(k-i);
                                        ind_ob_cerc=k;
                                  }
                            }
                            else
                            {

      keep_object(nube_act[index_ob_c1[k]].getX(),
nube_act[index_ob_c1[k]].getY(), nube_act[index_ob_c1[k]].getZ(),
nube_act[index_ob_c1[k]].getDist(),
nube_act[index_ob_c1[k]].getHAngle()*180/PI);  // Se guardan los
pixeles restantes en x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux
para posteriormente calcular su tamaño y posición media
                                  ult_pixel_ob=k;
                            }
                      }
                }
          }
          if (count==size)
                obj_retirado=true;  // El objeto se ha retirado
porque los pixeles ob son parecidos al bg inicial

          break;

      case 2:
          for (unsigned int k=i; k<i+size; k++)
          {
                if (abs(dist_bg_ini[index_bg_c2[k]]-
nube_act[index_ob_c2[k]].getDist())<=TH)
                {
                      count++;
                      // Añadimos el pixel a bg

      x_bg[index_bg_c2[k]]=nube_act[index_ob_c2[k]].getX();

      y_bg[index_bg_c2[k]]=nube_act[index_ob_c2[k]].getY();

      z_bg[index_bg_c2[k]]=nube_act[index_ob_c2[k]].getZ();

      dist_bg[index_bg_c2[k]]=nube_act[index_ob_c2[k]].getDist();
                }
                else
                {
                      // Si el primer pixel se sale de la media
respecto al segundo, no se tendrá en cuenta en el objeto
                      if (k==i &&
abs(nube_act[index_ob_c2[i]].getDist()-
nube_act[index_ob_c2[i+1]].getDist())>DIF_PIXEL)
                            count++;
                      else
                      {
                            // Si hay mucha diferencia entre la
distancia de los pixeles objeto, tampoco se añaden al objeto
                            if (k>i+1 && ult_pixel_ob!=-1 &&
abs(nube_act[index_ob_c2[k]].getDist()-
nube_act[index_ob_c2[ult_pixel_ob]].getDist())>DIF_PIXEL)
                            {
                                  count++;
                                  if (count1_ob_cerc==0)
                                  {
                                        count1_ob_cerc=size-(k-i);
```

```
                                        ind_ob_cerc=k;
                                  }
                            }
                            else
                            {

      keep_object(nube_act[index_ob_c2[k]].getX(),
nube_act[index_ob_c2[k]].getY(), nube_act[index_ob_c2[k]].getZ(),
nube_act[index_ob_c2[k]].getDist(),
nube_act[index_ob_c2[k]].getHAngle()*180/PI);  // Se guardan los
pixeles restantes en x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux
para posteriormente calcular su tamaño y posición media
                                  ult_pixel_ob=k;
                            }
                      }
                }
          }
          if (count==size)
                obj_retirado=true;  // El objeto se ha retirado
porque los pixeles ob son parecidos al bg inicial

          break;

      default:
          for (unsigned int k=i; k<i+size; k++)
          {
                if (abs(dist_bg_ini[index_bg_c3[k]]-
nube_act[index_ob_c3[k]].getDist())<=TH)
                {
                      count++;
                      // Añadimos el pixel a bg

      x_bg[index_bg_c3[k]]=nube_act[index_ob_c3[k]].getX();

      y_bg[index_bg_c3[k]]=nube_act[index_ob_c3[k]].getY();

      z_bg[index_bg_c3[k]]=nube_act[index_ob_c3[k]].getZ();

      dist_bg[index_bg_c3[k]]=nube_act[index_ob_c3[k]].getDist();
                }
                else
                {
                      // Si el primer pixel se sale de la media
respecto al segundo, no se tendrá en cuenta en el objeto
                      if (k==i &&
abs(nube_act[index_ob_c3[i]].getDist()-
nube_act[index_ob_c3[i+1]].getDist())>DIF_PIXEL)
                            count++;
                      else
                      {
                            // Si hay mucha diferencia entre la
distancia de los pixeles objeto, tampoco se añaden al objeto
                            if (k>i+1 && ult_pixel_ob!=-1 &&
abs(nube_act[index_ob_c3[k]].getDist()-
nube_act[index_ob_c3[ult_pixel_ob]].getDist())>DIF_PIXEL)
                            {
                                  count++;
                                  if (count1_ob_cerc==0)
                                  {
                                        count1_ob_cerc=size-(k-i);
                                        ind_ob_cerc=k;
```

```
                                        }
                                    }
                                    else
                                    {

      keep_object(nube_act[index_ob_c3[k]].getX(),
nube_act[index_ob_c3[k]].getY(), nube_act[index_ob_c3[k]].getZ(),
nube_act[index_ob_c3[k]].getDist(),
nube_act[index_ob_c3[k]].getHAngle()*180/PI);  // Se guardan los
pixeles restantes en x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux
para posteriormente calcular su tamaño y posición media
                                        ult_pixel_ob=k;
                                    }
                                }
                            }
                        }
                        if (count==size)
                              obj_retirado=true;  // El objeto se ha retirado
porque los pixeles ob son parecidos al bg inicial

                        break;
      }
      return obj_retirado;
}


float background::calc_tamaño(const unsigned int c, const
Scan::PointList& nube_act, const unsigned int i, const unsigned int
size)
{
      unsigned int ultimo;
      unsigned int primero = 0;
      float tam;

      switch (c)
      {
      case 0:
            ultimo = index_ob_c0[i+size-1];
            primero = index_ob_c0[i];


      tam=calc_hip(nube_act[primero].getX(),nube_act[ultimo].getX(),nu
be_act[primero].getY(),nube_act[ultimo].getY());
            break;

      case 1:
            ultimo = index_ob_c1[i+size-1];
            primero = index_ob_c1[i];


      tam=calc_hip(nube_act[primero].getX(),nube_act[ultimo].getX(),nu
be_act[primero].getY(),nube_act[ultimo].getY());
            break;

      case 2:
            ultimo = index_ob_c2[i+size-1];
            primero = index_ob_c2[i];


      tam=calc_hip(nube_act[primero].getX(),nube_act[ultimo].getX(),nu
be_act[primero].getY(),nube_act[ultimo].getY());
```

```
            break;

      default:
            ultimo = index_ob_c3[i+size-1];
            primero = index_ob_c3[i];


      tam=calc_hip(nube_act[primero].getX(),nube_act[ultimo].getX(),nu
be_act[primero].getY(),nube_act[ultimo].getY());
            break;
      }

      return tam;
}


float background::calc_tamaño_fin()
{
      float tam=0;
      unsigned int ultimo = x_obj_aux.size()-1;
      tam=calc_hip(x_obj_aux[0],x_obj_aux[ultimo],y_obj_aux[0],y_obj_a
ux[ultimo]);
      return tam;
}


float background::calc_hip(float x1, float x2, float y1, float y2)
{
      float cat1, cat2, h;
      cat1=abs(x1-x2);
      cat2=abs(y1-y2);
      h=sqrt((cat1*cat1)+(cat2*cat2));

      return h;
}


void background::clear_ob_aux()
{
      x_obj_aux.clear();
      y_obj_aux.clear();
      z_obj_aux.clear();
      dist_obj_aux.clear();
      hAngle_obj_deg_aux.clear();
}


void background::clear_ob()
{
      x_obj.clear();
      y_obj.clear();
      z_obj.clear();
      dist_obj.clear();
      ha_obj.clear();
      tamaño_obj.clear();
}


void background::mean()
{
      float med_x=0;
```

```
	float med_y=0;
	float med_z=0;
	float med_d=0;
	float med_ha=0;

	unsigned int tam_obj = x_obj_aux.size();

	for (unsigned int j=0;j<tam_obj;j++)
	{
		med_x+=x_obj_aux[j]/tam_obj;
		med_y+=y_obj_aux[j]/tam_obj;
		med_z+=z_obj_aux[j]/tam_obj;
		med_d+=dist_obj_aux[j]/tam_obj;
		med_ha+=hAngle_obj_deg_aux[j]/tam_obj;
	}

	x_obj.push_back(med_x);
	y_obj.push_back(med_y);
	z_obj.push_back(med_z);
	dist_obj.push_back(med_d);
	ha_obj.push_back(med_ha);
}


void background::clear_all_possible()
{
	// Indices
	index_ob_c0.clear();
	index_ob_c1.clear();
	index_ob_c2.clear();
	index_ob_c3.clear();

	index_bg_c0.clear();
	index_bg_c1.clear();
	index_bg_c2.clear();
	index_bg_c3.clear();

	// Vector para comparar bg con nube actual
	comparacion_c0.clear();
	comparacion_c1.clear();
	comparacion_c2.clear();
	comparacion_c3.clear();

	clear_ob_aux();
	clear_ob();
}


void background::delete_calc_ob()
{
	x_obj.pop_back();
	y_obj.pop_back();
	z_obj.pop_back();
	dist_obj.pop_back();
	ha_obj.pop_back();
}
```

6.3. background.h

```
/*!
 *      @file  background.h
 *    @author  Irene Santos Velázquez, isantos.ext@catec.aero
 *
 *  @internal
 *    Created  02/05/2013
 */

#ifndef BACKGROUND_DEF
#define BACKGROUND_DEF

#include <vector>
#include <IbeoAPI/Scan.hpp>

class background
{
private:

      ////////////////////////////////////
      ////////////// VARIABLES ///////////
      ////////////////////////////////////

      std::vector<unsigned int> num_objects_ant;  // Número de objetos
en cada canal en el instante anterior al actual

      // Indica los instantes de tiempo seguidos en que ha aparecido
un objeto en cada canal
      std::vector <unsigned int> times_obj_c0,
                                          times_obj_c1,
                                          times_obj_c2,
                                          times_obj_c3;

      // Background
      std::vector<float>    x_bg,
                        y_bg,
                        z_bg,
                        dist_bg,
                        hAngle_bg_deg;
      std::vector<unsigned int>    channel_bg;

      // Background inicial
      std::vector<float>    x_bg_ini,
                        y_bg_ini,
                        z_bg_ini,
                        dist_bg_ini,
                        hAngle_bg_deg_ini;
      std::vector<unsigned int>    channel_bg_ini;

      std::vector <double> LLA;   // Guarda [lambda, phi,
h]=[longitud, latitud, altura] que son las coordenadas en LLA (en
grados)

      // Se crea un vector resultado y que contiene 0s o 1s (0 indica
que no ha habido cambios respecto a bg y 1 que si ha habido cambios)
      std::vector<bool> comparacion_c0,
                              comparacion_c1,
                              comparacion_c2,
                              comparacion_c3;
```

```
	// Almacena los indices de los pixeles distintos a bg (respecto
a la nube de puntos actual)
	std::vector<unsigned int> index_ob_c0,
                                          index_ob_c1,
                                          index_ob_c2,
                                          index_ob_c3;

	// Almacena los indices de los pixeles distintos a bg (respecto
a bg)
	std::vector<unsigned int> index_bg_c0,
                                          index_bg_c1,
                                          index_bg_c2,
                                          index_bg_c3;

	// Objeto final auxiliar (almacena todos los puntos
correspondientes a un mismo objeto)
	std::vector<float>    x_obj_aux,
                        y_obj_aux,
                        z_obj_aux,
                        dist_obj_aux,
                                  hAngle_obj_deg_aux;

	// Objeto final (almacena las posiciones medias y el tamaño de
los objetos de cada canal)
	std::vector<float>    x_obj,
                        y_obj,
                        z_obj,
                        dist_obj,
                                  ha_obj,
                                  tamaño_obj;

	// Distancia media de los objetos en el instante anterior al
actual
	std::vector<float>    dist_obj_ant_c0,
                                  dist_obj_ant_c1,
                                  dist_obj_ant_c2,
                                  dist_obj_ant_c3;


	////////////////////////////////////
	////////////// FUNCIONES //////////
	////////////////////////////////////

	// Inicializa el tamaño (n) de los vectores correspondientes a
bg y bg inicial (x_bg, y_bg, z_bg, dist_bg, hAngle_bg_deg, x_bg_ini,
y_bg_ini, z_bg_ini, dist_bg_ini, hAngle_bg_deg_ini)
	void background::set_bg(const unsigned int n);

	// Almacena la nube de puntos inicial en los vectores
correspondientes a bg (x_bg, y_bg, z_bg, dist_bg, hAngle_bg_deg)
	void background::add_bg(const size_t i, const float _x, const
float _y, const float _z, const float d, const float ha, const
unsigned int c);

	// Almacena la nube de puntos inicial en los vectores
correspondientes a bg inicial (x_bg_ini, y_bg_ini, z_bg_ini,
dist_bg_ini, hAngle_bg_deg_ini)
	void background::add_bg_ini(const size_t i, const float _x,
const float _y, const float _z, const float d, const float ha, const
unsigned int c);
```

```
	// Inicializa como vectores vacios a los vectores
correspondientes al objeto (x_obj_aux, y_obj_aux, z_obj_aux,
dist_obj_aux, hAngle_obj_deg_aux, x_obj, y_obj, z_obj, dist_obj,
tamaño_obj)
	void background::set_ob();

	// Imprime por pantalla (y en un log) el número de objetos
detectados por canal, sus posiciones medias y su tamaño.
	void background::print_ob(const unsigned int c);

	// Almacena los elementos dados en los vectores correspondientes
al objeto auxiliar (x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux,
hAngle_obj_deg_aux), es decir, va añadiendo
	// los pixeles correspondientes a un objeto.
	void background::keep_object(const float _x, const float _y,
const float _z, const float _d, const float _ha);

	// Comprueba que los pixeles objetos sean realmente objeto, es
decir, que no se parecen al bg inicial y que sean parecidos entre
ellos. Se añadirán a los vectores del objeto auxiliar
	// (mediante la función keep_object) solo aquellos que cumplan
todas esas condiciones.
	// Si todos los pixeles objeto son parecidos al bg inicial, es
que el objeto ha sido retirado y la función devolvería true. En caso
contrario, devuelve false.
	bool background::check_obj_retirado(const unsigned int c, const
Scan::PointList& nube_act, const unsigned int i, const unsigned int
size, unsigned int& ind_ob_cerc, unsigned int& count1_ob_cerc);

	// Calcula el tamaño (según el canal) de la nube de puntos
indicada.
	// Para ello, unicamente se necesitan la x e y del primer y
último pixel de esa nube de puntos y calcular su hipotenusa (mediante
la función calc_hip)
	float background::calc_tamaño(const unsigned int c, const
Scan::PointList& nube_act, const unsigned int i, const unsigned int
size);

	// Calcula el tamaño final del objeto (con check_obj_retirado
puede que algunos pixeles que al principio se consideraron objetos ya
no lo sean y por tanto hay que recalcular el tamaño
	// del objeto, antes de imprimirlo por pantalla o guardarlo en
el log).
	// La nube de puntos a partir de la que calcula el tamaño es la
del objeto auxiliar (x_obj_aux e y_obj_aux), en concreto, utiliza el
primer y último pixel de estos vectores.
	float background::calc_tamaño_fin();

	// Calcula la hipotenusa de un triangulo, en el que sus vertices
son (0,0), (x1,y1), (x2,y2)
	float background::calc_hip(float x1, float x2, float y1, float
y2);

	// Limpia los vectores correspondientes al objeto auxiliar
(x_obj_aux, y_obj_aux, z_obj_aux, dist_obj_aux, hAngle_obj_deg_aux)
	void background::clear_ob_aux();

	// Limpia los vectores correspondientes al objeto final (x_obj,
y_obj, z_obj, dist_obj, tamaño_obj)
	void background::clear_ob();
```

```
	// Calcula la media de los elementos de los vectores
correspondientes al objeto auxiliar (x_obj_aux, y_obj_aux, z_obj_aux,
dist_obj_aux) y los almacena en los vectores del objeto
	// final (x_obj, y_obj, z_obj, dist_obj, tamaño_obj)
	void background::mean();

	// Limpia los vectores correspondientes a todos los indices que
se necesitan durante la ejecución de detect (index_ob_c0, index_ob_c1,
index_ob_c2, index_ob_c3, index_bg_c0, index_bg_c1,
	// index_bg_c2, index_bg_c3), a los vectores comparacion
(comparacion_c0, comparacion_c1, comparacion_c2, comparacion_c3) y a
los correspondientes a objeto (para ello llama a las
	// funciones clear_ob_aux() y clear_ob())
	void background::clear_all_possible();

	// Elimina el último elemento de los vectores correspondientes
al objeto final (x_obj, y_obj, z_obj, dist_obj)
	void background::delete_calc_ob();

public:
	// Constructor
	background(const Scan &scan);

	// Función principal que detecta por canales (por la técnica de
background) si hay o no objetos.
	// Para ello, compara los pixeles del bg con la nube de puntos
actual. Si su diferencia es menor que un umbral TH, se almacena un 0
en el vector comparacion y se añade el pixel actual
	// al bg, mediante una ponderación con ALFA. En caso contrario,
se almacena un 1. Si el número de 1s seguidos (o con huecos de 0s
menores de MAX_CEROS) es mayor que MIN_PIX_OBJ, se
	// calcula el tamaño del objeto. Si éste es mayor que TAM_MIN,
se comprueba si el objeto sigue estando (la diferencia con los pixeles
del bg inicial es pequeña) y si los pixeles son
	// parecidos entre ellos. En ese caso, se calcula la posición
media del objeto. Si ese mismo objeto sigue estando en una posición
parecido durante REP_OBJ instantes de tiempo seguidos,
	// entonces se imprime la posición del objeto.
	void background::detect(const Scan &scan);
};

#endif
```

6.4. conversor.cpp

```
/*!
 *      @file  conversor.cpp
 *    @author  Irene Santos Velázquez, isantos.ext@catec.aero
 *    @brief   Convert from BODY to LLA Coordinate System
 *
 *  @internal
 *    Created  10/05/2013
 */

#include "conversor.h"
#include <iostream>
#include <vector>
```

```
#include <cmath>
using namespace std;

#define PI 3.14159265
#define a 6378137.0 // Semieje mayor de la tierra (en metros)
#define f 1/298.257223563  // Factor de achatamiento

conversor::conversor(const double refLon, const double refLat, const
double refH, const double pitch, const double yaw, const double
roll):_refLon(refLon), _refLat(refLat), _refH(refH), _pitch(pitch),
_yaw(yaw), _roll(roll)
{
	// Las medidas angulares vienen dadas en radianes (refLon,
refLat, refH, pitch, yaw y roll)

	ned.resize(3);  // [xn, yn, zn]
	ecef.resize(3);  // [Xe, Ye, Ze]
	lla.resize(3);   // [lambda, phi, h]=[longitud, latitud, altura]
	body.resize(3);
}


std::vector <double> conversor::newConversion(const double x, const
double y, const double z)
{
	cout << "Body: (x,y,z)=(" << x << "," << y << "," << z << ")\n";
	// Se pasa de coordenadas body a NED
	body2ned(x,y,z);
	cout << "NED: (xn,yn,zn)=(" << ned[0] << "," << ned[1] << "," <<
ned[2] << ")\n";
	// Se pasa de coordenadas NED respecto al lidar a coordenadas
NED respecto al suelo
	//lidar2floor();
	// Se pasa de coordenadas NED a ECEF
	ned2ecef();
	cout << "ECEF: (xe,ye,ze)=(" << ecef[0] << "," << ecef[1] << ","
<< ecef[2] << ")\n";
	// Se pasa de coordenadas ECEF a LLA
	ecef2lla();
	cout << "LLA: (lon,lat,alt)=(" << lla[0] << "," << lla[1] << ","
<< lla[2] << ")\n";

	// Para comprobar si el cambio es correcto
	lla2ecef(lla[0]*PI/180,lla[1]*PI/180,lla[2]*PI/180);
	cout << "ECEF: (xe,ye,ze)=(" << ecef[0] << "," << ecef[1] << ","
<< ecef[2] << ")\n";
	ecef2ned(ecef[0],ecef[1],ecef[2]);
	cout << "NED: (xn,yn,zn)=(" << ned[0] << "," << ned[1] << "," <<
ned[2] << ")\n";
	ned2body();
	cout << "Body: (x,y,z)=(" << body[0] << "," << body[1] << "," <<
body[2] << ")\n";

	return lla;
}


void conversor::body2ned(const double x, const double y, const double
z)
{
	ned[0] = cos(_pitch)*cos(_yaw) * x +
```

```
                  (cos(_yaw)*sin(_pitch)*sin(_roll)-
sin(_yaw)*cos(_roll)) * y +

      (cos(_yaw)*sin(_pitch)*cos(_roll)+sin(_yaw)*sin(_roll)) * z;
      ned[1] = cos(_pitch)*sin(_yaw) * x +

      (cos(_yaw)*cos(_roll)+sin(_yaw)*sin(_pitch)*sin(_roll)) * y +
                  (sin(_yaw)*sin(_pitch)*cos(_roll)-
cos(_yaw)*sin(_roll)) * z;
      ned[2] = -sin(_pitch) * x +
                  (cos(_pitch)*sin(_roll)) * y +
                  (cos(_pitch)*cos(_roll)) * z;
}


void conversor::ned2ecef()
{
      // refLat y refLon están dadas en radianes
      lla2ecef(_refLon,_refLat, _refH); // Localización del punto de
referencia expresado en coordenadas ECEF

      // refLat y refLon están dadas en radianes
      ecef[0]=-ned[0]*sin(_refLat)*cos(_refLon)-ned[1]*sin(_refLon)-
ned[2]*cos(_refLat)*cos(_refLon)+ecef[0];
      ecef[1]=-ned[0]*sin(_refLat)*sin(_refLon)+ned[1]*cos(_refLon)-
ned[2]*cos(_refLat)*sin(_refLon)+ecef[1];
      ecef[2]=ned[0]*cos(_refLat)-ned[2]*sin(_refLat)+ecef[2];
}


void conversor::ecef2lla()
{
      const double b=a*(1-f);  // Semieje menor de la tierra (en
metros)
      const double e2=2*f-f*f;  // first eccentricity squared
      const double ep2=f*(2-f)/((1-f)*(1-f));  // second eccentricity
squared

      double r2, r, E2, F, G, c, s, P, Q, ro, tmp, U, V, zo;

      r2=ecef[0]*ecef[0]+ecef[1]*ecef[1];
      r=sqrt(r2);
      E2=a*a-b*b;
      F=54*b*b*ecef[2]*ecef[2];
      G=r2+(1-e2)*ecef[2]*ecef[2]-e2*E2;
      c=(e2*e2*F*r2)/(G*G*G);
      s=pow(1+c+sqrt(c*c+2*c),1/3);
      P=F/(3*(s+1/s+1)*(s+1/s+1)*G*G);
      Q=sqrt(1+2*e2*e2*P);
      ro=-(e2*P*r)/(1+Q)+sqrt((a*a/2)*(1+1/Q)-((1-
e2)*P*ecef[2]*ecef[2])/(Q*(1+Q))-P*r2/2);
      tmp=(r-e2*ro)*(r-e2*ro);
      U=sqrt(tmp+ecef[2]*ecef[2]);
      V=sqrt(tmp+(1-e2)*ecef[2]*ecef[2]);
      zo=(b*b*ecef[2])/(a*V);

      lla[0]=atan2(ecef[1],ecef[0])*180/PI;  // lambda = longitud (en
grados)
      lla[1]=atan((ecef[2]+ep2*zo)/r)*180/PI;  // phi = latitud (en
grados)
      lla[2]=U*(1-b*b/(a*V));  // h
```

```
}


void conversor::lla2ecef(const double lon, const double lat, const
double h)
{
	const double e2=2*f-f*f;  // first eccentricity squared

	double chi=sqrt(1-e2*sin(lat)*sin(lat));

	ecef[0]=(a/chi+h)*cos(lat)*cos(lon);
	ecef[1]=(a/chi+h)*cos(lat)*sin(lon);
	ecef[2]=(a*(1-e2)/chi+h)*sin(lat);
}


void conversor::ecef2ned(const double Xe, const double Ye, const
double Ze)
{
	lla2ecef(_refLon,_refLat, _refH); // Localización del punto de
referencia expresado en coordenadas ECEF

	ned[0]=-sin(_refLat)*cos(_refLon)*(Xe-ecef[0])-
sin(_refLat)*sin(_refLon)*(Ye-ecef[1])+cos(_refLat)*(Ze-ecef[2]);
	ned[1]=-sin(_refLon)*(Xe-ecef[0])+cos(_refLon)*(Ye-ecef[1]);
	ned[2]=-cos(_refLat)*cos(_refLon)*(Xe-ecef[0])-
cos(_refLat)*sin(_refLon)*(Ye-ecef[1])-sin(_refLat)*(Ze-ecef[2]);
}


void conversor::ned2body()
{
	body[0] = cos(_pitch)*cos(_yaw) * ned[0] +
			cos(_pitch)*sin(_yaw) * ned[1] +
			-sin(_pitch) * ned[2];
	body[1] = (cos(_yaw)*sin(_pitch)*sin(_roll)-
sin(_yaw)*cos(_roll)) * ned[0] +

(cos(_yaw)*cos(_roll)+sin(_yaw)*sin(_pitch)*sin(_roll)) * ned[1] +
			cos(_pitch)*sin(_roll) * ned[2];
	body[2] =
(cos(_yaw)*sin(_pitch)*cos(_roll)+sin(_yaw)*sin(_roll)) * ned[0] +
			(sin(_yaw)*sin(_pitch)*cos(_roll)-
cos(_yaw)*sin(_roll)) * ned[1] +
			cos(_pitch)*cos(_roll) * ned[2];
}
```

6.5. conversor.h

```
/*!
 *      @file  conversor.h
 *    @author  Irene Santos Velázquez, isantos.ext@catec.aero
 *
 *  @internal
 *    Created  10/05/2013
 */
```

```
#ifndef CONVERSOR_DEF
#define CONVERSOR_DEF

#include <vector>

class conversor
{
private:

	std::vector <double> ned;  // Guarda [xn, yn, zn] que son las
coordenadas en NED
	std::vector <double> ecef;  // Guarda [Xe, Ye, Ze] que son las
coordenadas en ECEF
	std::vector <double> lla;   // Guarda [lambda, phi,
h]=[longitud, latitud, altura] que son las coordenadas en LLA (en
grados)
	std::vector <double> body;

	const double _refLon;
	const double _refLat;
	const double _refH;
	const double _pitch;
	const double _yaw;
	const double _roll;

	// Convierte de coordenadas body (coordenadas en el sistema de
referencia del LIDAR) a coordenadas NED respecto del LIDAR
	void conversor::body2ned(const double x, const double y, const
double z);
	// Convierte de coordenadas NED (xn,yn,zn) a coordenadas ECEF
(Xe,Ye,Ze). Se necesita un punto de referencia dado en coordenadas LLA
(refLon, refLat,refH)
	void conversor::ned2ecef();
	// Convierte de coordenadas ECEF (Xe,Ye,Ze) a coordenadas LLA
(longitud,latitud,altura)=(lambda,phi,h)
	void conversor::ecef2lla();
	// Convierte de coordenadas LLA
(longitud,latitud,altura)=(lambda,phi,h) a coordenadas ECEF (Xe,Ye,Ze)
	void conversor::lla2ecef(const double lon, const double lat,
const double h);
	// Para comprobar que las coordenadas están bien
	void conversor::ecef2ned(const double Xe, const double Ye, const
double Ze);
	void conversor::ned2body();

public:
	// Constructor que convierte las medidas tomadas desde el LIDAR
a coordenadas LLA
	conversor(const double refLon, const double refLat, const double
refH, const double pitch, const double yaw, const double roll);
	std::vector <double> conversor::newConversion(const double x,
const double y, const double z);
};

#endif
```