UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Departamento de Electrónica

Máster Oficial en Sistemas Electrónicos Avanzados.
Sistemas Inteligentes.

Tesis de Máster

“Seguimiento de Múltiples Objetos en Espacios Inteligentes”

Álvaro Marcos Ramiro

Julio 2009

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Departamento de Electrónica

Máster Oficial en Sistemas Electrónicos Avanzados.
Sistemas Inteligentes.

Tesis de Máster

"Seguimiento de Múltiples Objetos en Espacios Inteligentes"

Autor: Álvaro Marcos Ramiro

Director: Marta Marrón Romera

TRIBUNAL:

Presidente: Juan Carlos García García

Vocal 1º: Daniel Pizarro Pérez

Vocal 2º: Marta Marrón Romera

CALIFICACIÓN:.....

FECHA:.....

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO

Convenciones sobre la notación de símbolos

Los acrónimos se indican en letra mayúscula
Las matrices se representan en cursiva y mayúsculas
Las variables escalares o vectoriales se representan en cursiva y minúsculas
Los vectores llevan una flecha sobre el nombre de la variable

Acrónimos

3DPF	*3-Dimensional Particle Filter*	*Filtro de Partículas Tridimensional*
CP	*Clustering Process*	*Proceso de Clasificación*
CV	*Constant Velocity model*	*Modelo de Velocidad Constante*
FPS	*Frames Per Second*	*Imágenes Por Segundo*
JPDAF	*Joint Probabilistic Data Association Filter*	*Filtro de Asociación Probabilística Conjunta de Datos*
KF	*Kalman Filter*	*Filtro de Kalman*
NN	*Nearest Neighbour*	*Vecino Más Próximo*
PDF	*Probability Density Function*	*Función Densidad de Probabilidad*
PF	*Particle Filter*	*Filtro de Partículas*
SIS	*Sequential Important Sampling*	*Muestreo Secuencial Ponderado*
SIR	*Sequential Important Resampling*	*Muestreo Secuencial Ponderado con Selección*
XPF	*Extended Particle Filter*	*Filtro de Partículas Extendido*
XPFCP	*Extended Particle Filter with Clustering Process*	*Filtro de Partículas Extendido con Proceso de Segmentación*

Símbolos

XYZ	*Sistema de coordenadas tridimensional de componentes x, y, z que define el entorno de seguimiento.*
x	*Coordenada de lateralidad en sistema tridimensional que define el entorno*
y	*Coordenada de altura en sistema tridimensional que define el entorno*
z	*Coordenada de profundidad en sistema tridimensional que define el entorno*
vx	*Componente de velocidad en dirección del eje x en el sistema de coordenadas XYZ*
vy	*Componente de velocidad en dirección del eje y en el sistema de coordenadas XYZ*
vz	*Componente de velocidad en dirección del eje z en el sistema de coordenadas XYZ*
UV	*Sistema de coordenadas bidimensional de componentes u, v que define el plano imagen de la cámara.*
u	*Coordenada de abscisas en sistema bidimensional que define el plano imagen*
v	*Coordenada de ordenadas en sistema bidimensional que define el plano imagen*
I	*Matriz identidad*
$\vec{x}$	*Vector de estado de un sistema descrito en variables de estado*[1]
χ	*Voto asociado a una clase en el proceso de identificación de robots*
A	*Matriz de transición de un sistema lineal descrito en variables de estado*
b	*Altura discreta en el algoritmo Visual Hull*
$\vec{u}$	*Vector de entrada de un sistema descrito en variables de estado*
M	*Matriz de re-escalado en Visual Hull*
m	*Número total de medidas*
$\vec{y}$	*Vector de salida (o de medida) de un sistema descrito en variables de estado*
Y	*Matriz de medidas generado por el sistema de seguimiento. $Y = \{\vec{y}_i / i = 1:m\}$*
C	*Matriz de transición de un sistema lineal descrito en variables de estado*
c	*Identificador de una cámara*
$\mathrm{d_M}$	*Distancia de Mahalanobis*
$\mathrm{d_E}$	*Distancia Euclidea*
$\vec{v}$	*Vector de ruido asociado al de estado en un sistema descrito en variables de estado*
V	*Matriz de covarianza del ruido de estado*
$\vec{o}$	*Vector de ruido asociado al de medidas en un sistema descrito en variables de estado*
o	*Tamaño de la rejilla de ocupación en Visual Hull*
O	*Matriz de covarianza del ruido de medida*
$f(\vec{x}_t, \vec{u}_t, \vec{o}_t)$	*Función no lineal que caracteriza a la ecuación de transición del modelo de un sistema expresado en variables de estado*
$h(\vec{x}_t, \vec{u}_t, \vec{r}_t)$	*Función no lineal que caracteriza a la ecuación de salida del modelo de un sistema expresado en variables de estado*
λ_c	*Homogenizador de coordenadas para la cámara c*
λ	*Cada una de las dimensiones del espacio de características de la clasificación*
K_c	*Matriz de parámetros intrínsecos de la cámara c*
k	*Número de clases. Se añade el subíndice "in" para indicar clases de medidas y "out" para indicar clases de partículas, o número de objetos en la aplicación de seguimiento*

[1] La flecha indicativa de vector diferencia a los vectores $\vec{u}$, $\vec{v}$, $\vec{x}$ e $\vec{y}$ de los valores escalares de coordenadas u, v, x e y de los espacios UV y XYZ, el número total de miembros l_j de cada clase j de la variable que caracteriza a cada uno de sus i miembros $\vec{l}_{i,j} / i = 1:l_j$, y su centroide $\vec{g}_j$ con la función genérica $g()$.

G	*Matriz/estructura contenedora de la información de salida del proceso de clasificación para cada clase.* $G_j \ / \ j = 1:k$
$\vec{g}_j$	*Centroide de la clase.* $\vec{g}_j \ / \ j = 1:k$
$H_{c,b}$	*Matriz de homografía de la cámara c a una altura b*
L_j	*Conjunto de miembros de la clase.* $L_j = \left\{ \vec{l}_{i,j} \ / \ i = 1:l_j \right\}$, *donde* $\vec{l}_{i,j}$ *representa a cada miembro de la clase y* l_j *al número total de miembros de la misma*
τ_j	*Identificador de la clase.* $\tau_j \ / \ j = 1:k$
β_j	*Variable de validación de la clase.* $\beta_j \ / \ j = 1:k$
α	*Diversas connotaciones:* α *factor de ponderación superior con* $M_{max\ cubo}$ α_{new} *factor de ponderación al crear nuevas clases con el clasificador "subtractive"* α_{valid} *factor de olvido para validar las clases*
β	*Factor de ponderación inferior con* $M_{max\ cubo}$
Π_b	*Plano común en coordenadas espaciales al que se referencian las imágenes de cada cámara c mediante la matriz de homografía*
π	*Factor de histéresis de los límites de validación de clases:* π_d *para el límite de distancia* d_{valid} π_p *para el límite de verosimilitud* p_{valid}
n	*Número total de partículas*
n_{eff}	*Número total de partículas efectivo según [Liu98]*
R_c	*Matriz de rotación de la cámara c*
T_c	*Matriz de traslación de la cámara c*
$\vec{s}$	*Valor de una partícula.*
S	*Matriz de partículas.* $S = \left\{ \vec{s}_i \ / \ i = 1:n \right\}$
$\vec{w}$	*Vector de pesos del conjunto de partículas.* $\vec{w} = \left\{ w_i \ / \ i = 1:n \right\}$
$w()$	*Función de ponderación del conjunto de partículas*
n_m	*Número total de partículas introducidas en etapa de re-inicialización del XPFCP*
n_{div}	*Número de divisiones del histograma en la ecualización de medidas basada en histograma*
umbralHist	*Umbral de medidas aplicado en los histogramas obtenidos en la ecualización de medidas basada en histograma*
propRed	*Proporción de medidas a eliminar, de las consideradas redundantes en el algoritmo de ecualización de medidas basado en histograma*
distG	*Distancia de clasificación para el k-medias en tres dimensiones, en el algoritmo de ecualización de medidas basado en grupos*
H	*Grupo originado con el k-medias en tres dimensiones en el algoritmo de ecualización de medidas basado en grupos*
umbralRed	*Umbral a partir del cual se consideran redundantes los elementos de un grupo originado al aplicar el k-medias en tres dimensiones, en el algoritmo de ecualización de medidas basado en grupos*
umbralGr	*Umbral de medidas aplicado en los histogramas obtenidos en la ecualización de medidas basada en grupos*

$M_{max\ cubo}$	*Número de medidas máximo que cabe en la ventana de búsqueda de medidas en torno a una partícula.*
M_{min}	*Valor inferior de saturación de pesos antes de normalizar en el algoritmo de pesado del 3DPF*
M_{max}	*Valor superior de saturación de pesos antes de normalizar en el algoritmo de pesado del 3DPF*
P	*Número de planos con los que se divide el volumen, en la clasificación de partículas del 3DPF*
h	*Distancia entre dos planos P contiguos*
Ω	*Velocidad angular del robot*
Ω_P	*Plano intermedio entre dos planos P, a una distancia h/2 de cada uno de ellos*
$distC$	*Distancia de conectividad, en el algoritmo de clasificación de partículas del 3DPF*
M_q	*Estructura de salida del 3DPF:* $M_q = \{\vec{\mu}_{s,q} / s = 1{:}\,\mu_q\}$
$\vec{\mu}_{s,q}$	*Elemento de una estructura* M_q
ξ	*Ponderación en coordenadas XZ para calcular la distancia distC*
δ	*Ponderación en Y para calcular la distancia distC*
$TAMBUFFER$	*Número de iteraciones sobre las que se realizan distintos procesos en el algoritmo de identificación*
$ORBUFFER$	*Número de iteraciones sobre las que se filtra la orientación de una clase*
$VOTOPOS$	*Número de votos con los que se premia la clase considerada robot*
$VOTONEG$	*Número de votos con los que se penaliza la clase menos parecida a la del robot*
odo_x	*Coordenada X de la odometría proporcionada por el robot*
odo_z	*Coordenada Y de la odometría proporcionada por el robot (en el sistema de referencia utilizado en esta tesis, se corresponde con la coordenada Z)*
odo_ϕ	*Orientación proporcionada por la odometría*
ϕ	*Orientación de una clase*
vl	*Módulo de la velocidad de una clase*
χ	*Voto asociado a una clase en el proceso de identificación*
Σ	*Matriz de covarianza*
Σ_u	*Matriz de propagación del error*
J	*Matriz Jacobiano*
γ	*Relación entre* n_m *y* n
p	*Valor de probabilidad o verosimilitud*
$p(a\|b)$	*Probabilidad de* a *condicionada a* b
$p(a\|b,c)$	*Probabilidad de* a *condicionada a* b *y* c
$p(\vec{x}_t\|\vec{x}_{t-1})$	*PDF que caracteriza al modelo de actuación o de estado en un sistema estocástico*
$p(\vec{y}_t\|\vec{x}_t)$	*Verosimilitud del vector de medidas. Caracteriza al modelo de observación o de medida en un sistema estocástico*
$p(\vec{x}_t\|\vec{y}_{1:t-1})$	*"Prior distribution" o probabilidad a priori de la estimación del vector de estado*
$p(\vec{x}_t\|\vec{y}_{1:t})$	*"Posterior distribution" o probabilidad a posteriori de la estimación del vector de estado, también llamada creencia*
E()	*Función estadística de la esperanza o el valor medio*
μ , N	*Media de una variable aleatoria escalar y vectorial, respectivamente*
σ , Ω	*Desviación típica de una variable aleatoria escalar y vectorial, respectivamente*

$norm(\mu,\sigma)$	*PDF normal de una variable y vector de variables aleatorias, de media* μ *y* N*, y*
$norm(\mathrm{N},\Omega)$	*desviación típica* σ *y* Ω*, respectivamente*
$g()$	*Función genérica*
$\max$	*Función máximo*
$\min$	*Función mínimo*
η	*Factor de normalización*
d	*Distancia genérica*
i , j , a	*Variables índice genéricas*
t	*Instante de tiempo*
t_s	*Periodo de muestreo*
t_{exe}	*Tiempo de ejecución*

Superíndices

T	*Transpuesta de una matriz*
$\vec{\ }$	*Vector*
$\bar{\ }$	*Valor medio o promediado. También aparece expresado a través de la función estadística de la esperanza.* $\mathrm{E}(a)=\bar{a}$
(i)	*Referente a una muestra de la señal*
$\tilde{\ }$	*Variable normalizada*
$'$, $\breve{\ }$	*Variable secundaria o aproximada*

Subíndices

T	*Valor total*
$\max$	*Valor máximo*
$\min$	*Valor mínimo*
opt	*Valor óptimo*
eff	*Valor eficaz*
exe	*Valor de ejecución*
th	*Valor límite*
$valid$	*Valor de validación*
t	*Referente al instante temporal*
$iter$	*Referente a la iteración del algoritmo*
s	*Referente al periodo de muestreo*
m	*Referente a las medidas obtenidas con el modelo de observación*
g	*Referente al centroide del proceso de clasificación*
$filt$	*Filtrado*
r	*Referente a la clase del robot*

I. RESUMEN

Esta tesis trata dos métodos de seguimiento de múltiples objetos en un espacio inteligente: para ello se parte de una serie de cámaras sincronizadas y situadas estratégicamente en el entorno que permiten extraer toda la información referente al mismo e interaccionar con él. A partir esta información, el trabajo aquí descrito permite obtener, o bien la posición, trayectoria, velocidad de los elementos presentes en la escena ([Marrón08]) junto con la identidad de un robot, de forma bidimensional, o bien una reconstrucción tridimensional de las entidades presentes en el espacio inteligente.

Además se realiza el seguimiento bidimensional y tridimensional de estas reconstrucciones mediante técnicas probabilísticas, para hacer robusto al sistema frente a perturbaciones.

A partir del seguimiento, se define de forma individual cada una de las entidades existentes en el espacio inteligente en cada instante, y, en el caso bidimensional, se diferencia un robot respecto del resto de elementos. En esta tesis se demuestra, además, la fiabilidad y robustez de los algoritmos mediante diversas pruebas reales.

Palabras clave: Espacio inteligente, algoritmos probabilísticos, anillo de cámaras, seguimiento bidimensional, seguimiento tridimensional, identificación.

ABSTRACT

This thesis goes through two multiple object tracking methods in intelligent spaces. The observation model is based in a ring of fully calibrated and synchronized cameras, strategically placed in the environment to extract all the information and interact with it. This work presents two ways of using that information. The first one is to obtain the position, trajectories and speed of the present objects in the scene ([Marrón08]), together with the identity of a robot, all in two dimensions. The second one is to get a three dimensional volume reconstruction of those objects.

Also, tracking with those two variants is done by using probabilistic techniques, in order to increase the robustness of the system against perturbations.

Each entity is defined individually from the tracking data. This way, a robot can be spotted from the rest of the elements, by using its odometry. The reliability and robustness of the proposal presented is finally demonstrated in this thesis, with different tests from real data.

Keywords: Intelligent space, probabilistic algorithms, camera ring, two-dimensional tracking, three-dimensional tracking, identification.

II. MEMORIA

CAPÍTULO 1

INTRODUCCIÓN

El seguimiento de entidades como personas y robots ha sido un campo de gran interés en los últimos tiempos. Así lo demuestra la extensa literatura publicada, en la que se definen diversas maneras de abordar la problemática que ello conlleva.

Asimismo, el concepto de espacio inteligente, caracterizado por contar con inteligencia distribuida en cámaras, robots y otros agentes, es objeto de continuas propuestas ([Lee99], [Jeni06]), y se muestra en la Figura 1.1.

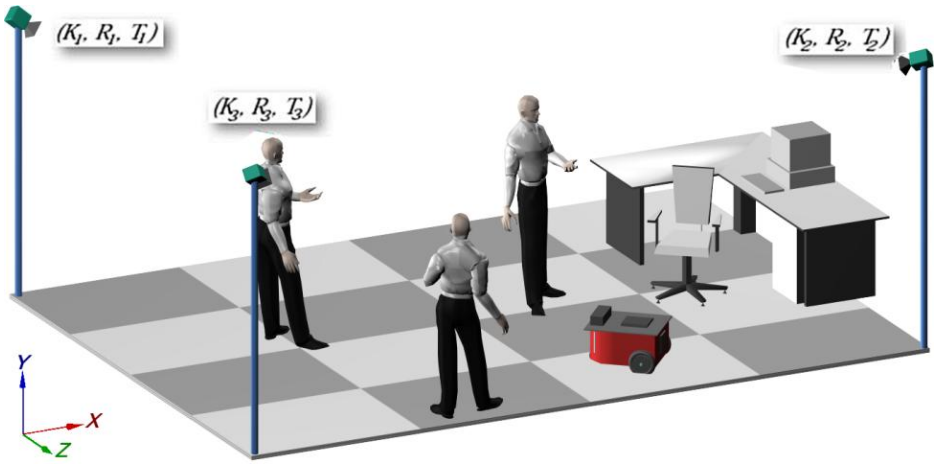


Figura 1.1. Renderizado tridimensional del espacio inteligente.

1.1 Contenido de la tesis y objetivos

En esta tesis se proponen dos formas distintas de realizar el seguimiento, siendo éste de distinta naturaleza: por una parte, se realiza un seguimiento bidimensional de las entidades presentes en el entorno mediante un XPFCP (eXtended Particle Filter with Clustering Process, [Marrón08]). Por otra, mediante una serie de modificaciones al algoritmo de seguimiento bidimensional, se pasa a realizar un seguimiento tridimensional mediante un 3DPF (3-Dimensional Particle Filter).

En ambos casos se consigue mediante la utilización de un único Filtro de Partículas (PF o Particle Filter). Esto aporta dos ventajas importantes: por una parte, la utilización de un algoritmo probabilístico robustece y aporta fiabilidad a la funcionalidad del seguidor frente a seguidores determinísticos como [Atyabi06]. Por otra parte, la resolución del problema de estimación con un solo algoritmo multimodal ([Marrón08]) constituye una importante mejora frente a otras propuestas como [Krumm01] y [López07], donde se utiliza un filtro por cada elemento a seguir.

El sistema de observación se basa en un anillo de cámaras calibradas, sincronizadas y comunicadas por red local con un conjunto de servidores (uno por cámara), que procesan cada imagen y envían la información elaborada a un equipo cliente mediante red local. La distribución de las cámaras es tal que se evitan en la medida de lo posible las oclusiones de cualquiera de los objetos a seguir. A partir de las imágenes obtenidas por las cámaras se realiza una segmentación basada en resta de fondo, por lo que los elementos presentes en la escena quedan diferenciados en una imagen binaria. Posteriormente, en el procesador cliente, mediante la técnica conocida como Visual Hull (descrita en [Yang03]), se obtienen una serie de proyecciones a planos paralelos al del suelo de los elementos binarizados. Estas proyecciones se utilizan como observaciones de entrada del seguidor. Esto se explica de forma más detallada en los capítulos 2, 3 y 4.

Los objetivos de la presente tesis son dos: el primero, analizar y corroborar la robustez del funcionamiento del XPFCP funcionando en tiempo real en un espacio inteligente, así como identificar un robot respecto al resto de elementos; el segundo, diseñar un algoritmo que permita realizar el seguimiento y reconstrucción morfológica tridimensional de los elementos presentes en el espacio inteligente, también a partir de datos reales.

1.2 Estado del arte

Como se ha indicado anteriormente, el seguimiento en imágenes, sobre todo de personas, es un tema muy tratado en los últimos tiempos. Esta tesis lo orienta a los espacios inteligentes.

En [Lee99] se trata el concepto de espacio inteligente. En ese artículo se hace referencia a DINDs (Distributed Intelligent Networked Devices, o Dispositivos Inteligentes Distribuidos mediante Red), que son los sensores que captan y procesan la información recogida en el espacio inteligente, además de comunicarla al resto de DINDs mediante una red. En la Figura 1.2 pueden verse dos esquemas incluidos en ese trabajo, que ejemplifican esta funcionalidad.

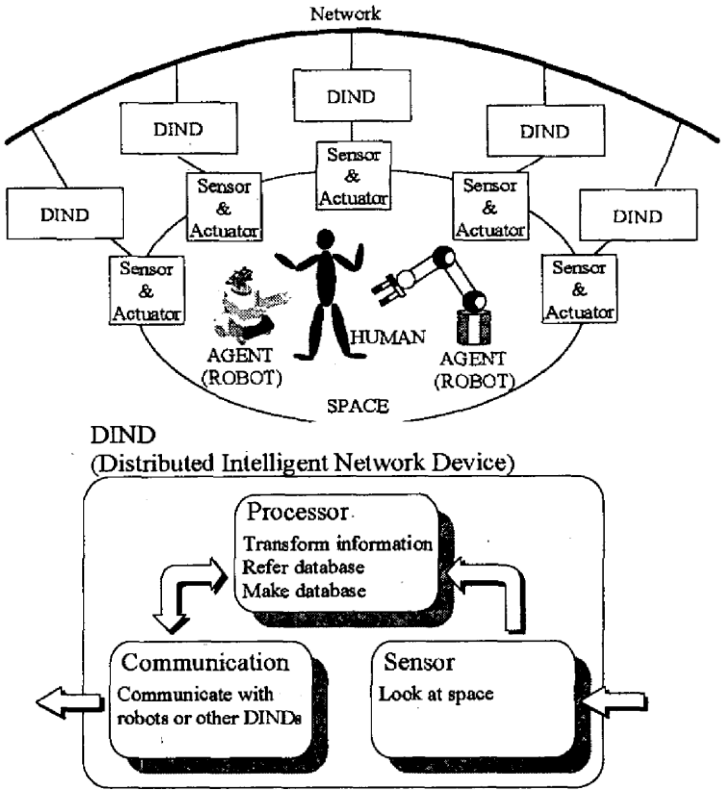


Figura 1.2. Concepto de espacio inteligente según [Lee99]. Arriba: esquema del sistema. Abajo: estructura de un DIND.

Como ya se ha mencionado, el seguimiento de personas es espacios inteligentes ha sido un campo de gran interés en los últimos tiempos.

Así, en [Krumm00] se realiza un seguimiento bidimensional de varias personas en un espacio inteligente, mediante múltiples cámaras para un proyecto conocido como EasyLiving ([Shafer98], en la Figura 1.3 puede verse el laboratorio). Se utiliza una resta de fondo para identificar los blobs (conjuntos de pixels) con forma de personas. Se mantienen los identificadores asociados a cada persona mediante un estimador e histogramas de color, que se guardan por zonas de la escena para conseguir inmunidad ante cambios de iluminación entre habitaciones.

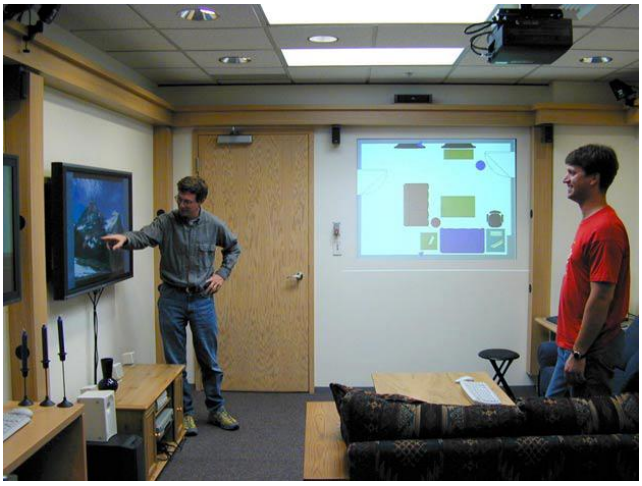


Figura 1.3. Laboratorio de EasyLiving para el seguimiento de múltiples personas.

En [Jeni06], como en [Lee99], se retorna la definición del concepto de espacio inteligente, orientándolo hacia la robótica móvil, mediante procedimientos demostrados en esos trabajos. Se realiza la observación y seguimiento bidimensional de personas en un espacio inteligente para planificar rutas de robots. Por ejemplo, las zonas en las que se observa que las personas andan frecuentemente se determinan como zonas seguras para el movimiento de robots, como puede verse en la Figura 1.4, extraída de [Jeni06].

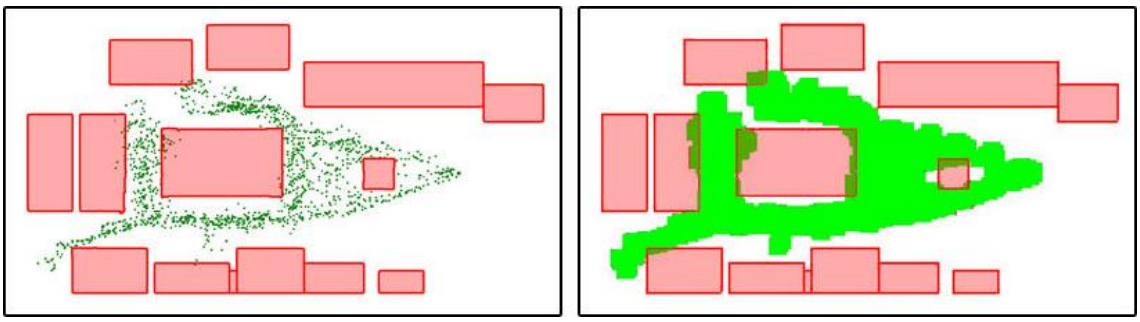


Figura 1.4. Determinación de zonas seguras para el movimiento de robots en [Jeni06]. Izquierda: zonas en las que se mueven las personas. Derecha: zona segura para robots.

En [Khan06] se realiza un seguimiento determinístico de múltiples personas, a partir de múltiples cámaras para evitar oclusiones. Se utilizan las homografías a nivel del suelo para llevar a cabo la tarea de detección de las personas. Se mantienen las identidades de las mismas a lo largo del tiempo mediante un sistema de clasificación. En la Figura 1.5 se muestra el efecto de combinar información relativa a múltiples cámaras (en azul claro), para determinar la situación de las personas en el plano del suelo (en rojo).

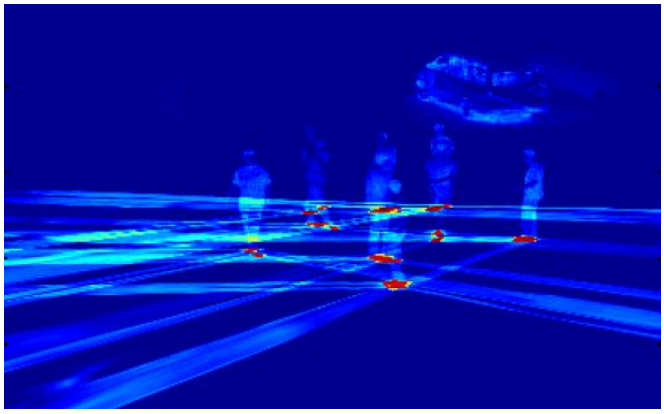


Figura 1.5. Sistema de seguimiento propuesto por [Khan06].

En [López07] se propone seguimiento tridimensional de múltiples personas mediante filtros de partículas (uno por objeto a seguir), con un sistema de observación que explota la redundancia espacial de un mismo volumen visto por distintas cámaras. Es por tanto muy parecido al seguimiento tridimensional propuesto en la presente tesis, con algunas diferencias, mencionadas en el punto 1.1. En la Figura 1.6 se muestra un ejemplo de funcionamiento del sistema de seguimiento propuesto, extraído de [Lopez07]:

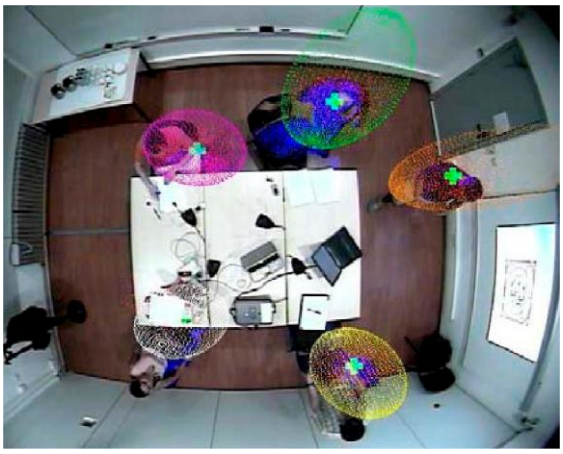


Figura 1.6. Seguimiento tridimensional de múltiples personas propuesto por [López07].

En [Kang05] aparece otra variante algorítmica de seguimiento en imagen: en este caso se utilizan JPDAFs (Joint Probabilistic Data Association Filters, o Filtros de Asociación Conjunta de Datos Probabilísticos, [Zhou93]). En este caso se amplía el vector de estado del sistema para poder realizar el seguimiento de múltiples objetos. Alternativas a esto, ya mencionadas, consisten en la utilización de un único filtro para seguir múltiples objetos, o la utilización de un filtro por objeto. La Figura 1.7 muestra un ejemplo de seguimiento extraído de [Kang05].

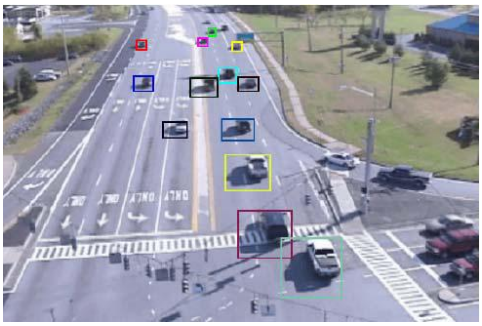


Figura 1.7. Seguimiento de tráfico realizado por [Kang05]

Puede concluirse que las tareas de seguimiento en espacios inteligentes es un tema ampliamente tratado en la actualidad. El sistema de observación suele estar basado en la utilización de múltiples cámaras, ya que esto reduce las posibles oclusiones. Existen múltiples alternativas distintas a las tratadas en la presente tesis para realizar el seguimiento de múltiples objetos, siendo el objetivo común la posibilidad de ser utilizadas en tiempo real.

1.3 Estructura del sistema

Los diagramas funcionales de los sistemas bidimensional y tridimensional se muestran en las Figura 1.8 y Figura 1.9, respectivamente.

La implementación de partida del XPFCP en tiempo real ha sido llevada a cabo por [Marrón08]. En el presente trabajo se ha llevado a cabo la implementación del sistema de identificación en tiempo real, así como del sistema de seguimiento tridimensional, implementado en el entorno de simulación Matlab.

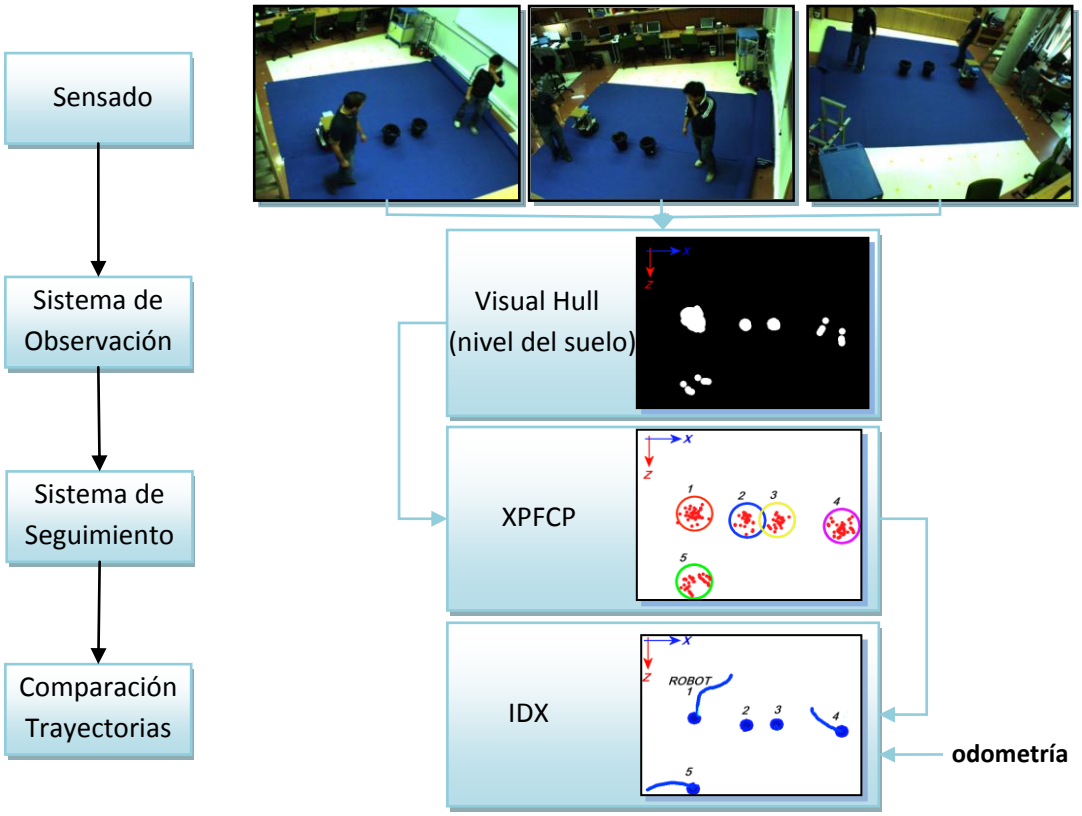


Figura 1.8. Diagrama funcional del sistema de seguimiento bidimensional.

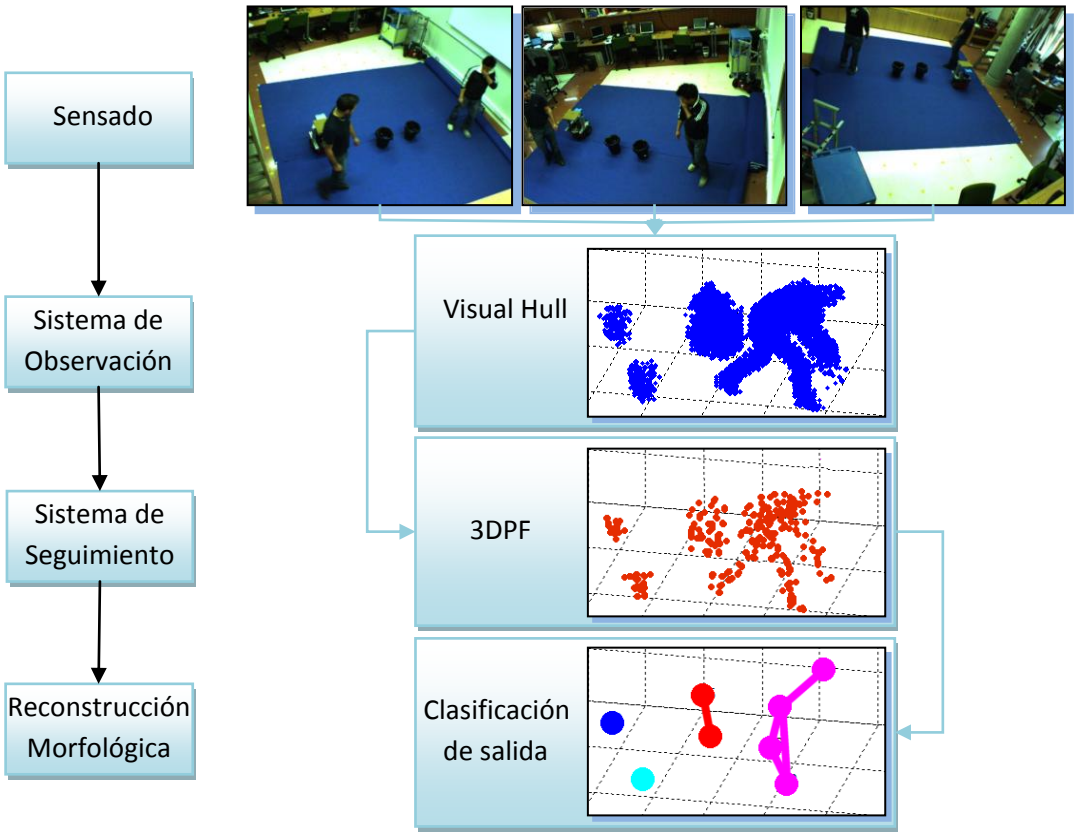


Figura 1.9. Diagrama funcional del sistema de seguimiento tridimensional.

1.4 Estructura de la memoria

La memoria está compuesta por 10 capítulos y se organiza de la siguiente forma:

- **Capítulo 1. Introducción:** Es el capítulo actual, en el que el trabajo realizado se pone dentro de un contexto, se introduce y se definen sus objetivos.
- **Capítulo 2. Base Teórica:** Se exponen las bases teóricas sobre las que se apoyan los sistemas de seguimiento implicados en la tesis, que se analizan con más profundidad en los capítulos 3 y 4. Además se muestra la teoría del sistema de observación utilizado.
- **Capítulo 3. Seguimiento Bidimensional:** Se trata el algoritmo de seguimiento bidimensional utilizado, así como el proceso añadido en este trabajo a su salida que permite identificar un robot del resto de los elementos del entorno.
- **Capítulo 4. Seguimiento Tridimensional:** Análogo al capítulo anterior, pero aplicado al seguidor tridimensional propuesto.
- **Capítulo 5. Resultados:** Este capítulo está dividido en dos partes. En la primera se describen las pruebas y condiciones en las que se ha demostrado el funcionamiento del seguidor bidimensional (Figura 1.8), para posteriormente mostrar los resultados obtenidos. En la segunda se hace lo propio con el 3DPF.
- **Capítulo 6. Conclusiones y Trabajos Futuros:** Se exponen las conclusiones obtenidas a partir del trabajo realizado, y se proporciona una visión de las posibles líneas de trabajo futuras relacionadas con el mismo.
- **Capítulo 7. Manual de Usuario:** Este capítulo detalla el modo reproducir la aplicación y reutilizar los códigos.
- **Capítulo 8. Pliego de Condiciones:** Incluye las necesidades hardware y software para la realización del sistema tratado.
- **Capítulo 9. Presupuesto:** Necesidades económicas para la realización del sistema tratado.
- **Capítulo 10. Planos:** Alberga el código de las partes de la aplicación de seguimiento bidimensional más relacionadas con el trabajo realizado (ya que la totalidad del código de la aplicación sería demasiado extenso), y de los ficheros necesarios para simular el sistema de seguimiento tridimensional en Matlab.

CAPÍTULO 2

BASE TEÓRICA

En este capítulo se exponen los fundamentos teóricos básicos sobre los que se apoyan tanto el algoritmo de observación como el de seguimiento, que son las partes por las que está formado el sistema diseñado en este trabajo, como se explicó en el capítulo anterior.

2.1 Algoritmo de observación: Visual Hull

Para hacer posible el seguimiento tanto tridimensional como bidimensional de uno o varios objetos, como se persigue en este trabajo, es necesario contar con un sistema de observación que extraiga información tridimensional del entorno. Para resolver este problema se propone la utilización de una técnica conocida como Visual Hull.

2.1.1. Introducción

El "Visual Hull" se define como el mayor volumen encerrado por la intersección de los conos visuales de las siluetas obtenidas a partir de diferentes cámaras. El volumen que se obtiene no se corresponde con el volumen real que ocupan los objetos en la escena. Sin embargo, puede asegurarse que lo contiene. El funcionamiento del proceso de "Visual Hull" se muestra gráficamente en la Figura 2.1.

Este volumen se obtiene siguiendo el proceso descrito a continuación.

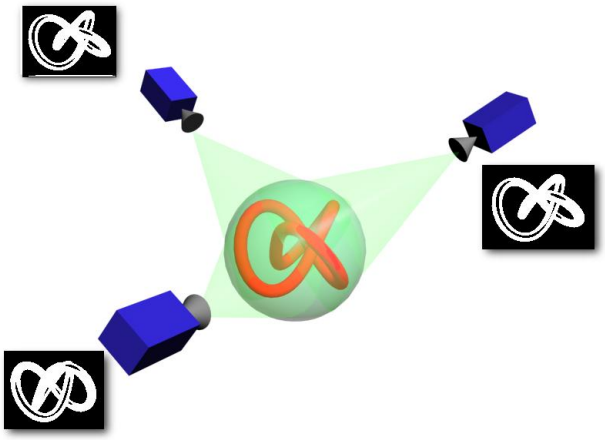


Figura 2.1. Representación gráfica del concepto de Visual Hull.

2.1.2. Descripción de funcionamiento

A continuación se detalla la descripción de funcionamiento del Visual Hull. Todo esto puede verse con más detalle en [Rodríguez08], [Espejo08], [Jalvo09] y [Marrón09].

a) Se parte de un anillo de cámaras calibradas, asumiendo un modelo pin-hole para cada una de ellas. El modelo pin-hole está definido por una matriz K_c que contiene características intrínsecas a la cámara *c* (distancia focal y distorsiones), y dos matrices R_c y T_c que definen las propiedades extrínsecas de dicha cámara *c*: rotación y traslación, respectivamente.

b) Los objetos detectados en cada cámara son referenciados a un plano común Π_b (b es el valor discreto de la altura). Esto se realiza mediante la matriz $H_{c,b}$ de homográfia, según la siguiente expresión:

$$\begin{bmatrix} x_{c,b} \\ y_{c,b} \\ 1 \end{bmatrix} = {\lambda_c}^{-1} {H_{c,b}}^{-1} \begin{bmatrix} u_c \\ v_c \\ 1 \end{bmatrix} \qquad (2.1)$$

Transformando (2.1) es posible obtener una relación entre K_c, R_c y T_c con la matriz $H_{c,b}$, como puede verse en la siguiente expresión, equivalente a (2.1):

$$\begin{bmatrix} x_{c,b} \\ y_{c,b} \\ 1 \end{bmatrix} = \lambda_c K_c \left(R_c \begin{bmatrix} u_c \\ v_c \\ 1 \end{bmatrix} + T_c \right) \qquad (2.2)$$

Donde λ_c y K_c serán iguales si las cámaras son iguales, como es el caso. Multiplicando entre sí las imágenes binarizadas de cada cámara, una vez referidas al mismo plano Π_b, se obtiene la imagen de intersección en el Visual Hull de los objetos de la escena con ese plano.

c) Repitiendo este proceso para una serie de planos Π_b paralelos, es posible reconstruir el volumen ocupado por los objetos presentes en el entorno, como se ilustra en la Figura 2.2. Esto da origen a una serie de puntos $[x_t \; y_t \; b]^T$ en el espacio, que es la salida del algoritmo.

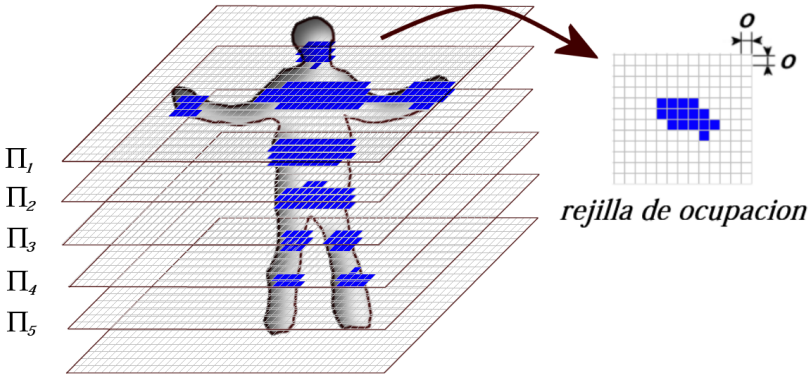


Figura 2.2. Esquema del concepto de Visual Hull. Se utilizan distintos planos Π_b a diferentes alturas *b* sobre las que se proyectan las siluetas del volumen a esas alturas. Como puede verse en el detalle del plano superior, el espacio de cada plano está discretizado, constituyendo lo que se conoce como rejilla de ocupación. El tamaño del lado de cada celda es *o*.

En la aplicación propuesta se utilizan tres procesos para que el proceso pueda ser implementando en tiempo real:

1. *Resta de fondo*. Mediante una resta de fondo por distancia de Mahalanobis se segmentan las imágenes recibidas por cada cámara, consiguiendo separar así los objetos presentes en el entorno del fondo del mismo. Los elementos de la escena quedan binarizados, como puede verse en la Figura 2.3.

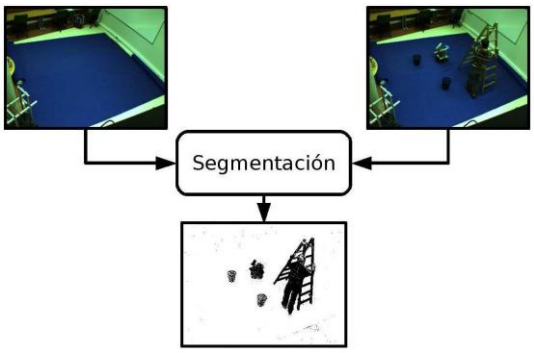


Figura 2.3. Diagrama del proceso de segmentación.

2. *Discretización en rejilla*. El plano Π_b es discretizado mediante una rejilla de ocupación. Cada celda de la misma, de lado *o*, tiene dos posibles estados: lleno, si forma parte del volumen definido por Visual Hull, o vacío, si no forma parte.
3. *Transformación de escala*. La transformación originada al aplicar $H_{c,b}$ crea una escala en milímetros. Se hace necesario el uso de una matriz *M* que re-escale la imagen a pixeles, para obtener una imagen representable. El procedimiento se muestra en la figura:

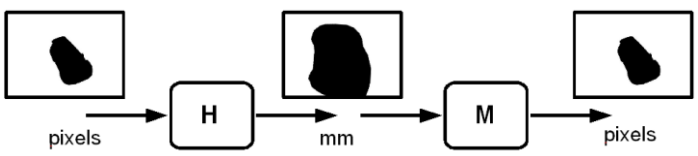


Figura 2.4. Proceso seguido para obtener la intersección del volumen de la escena con un plano Π_b determinado.

2.1.3. Aplicación al sistema de seguimiento

Como se ha visto en el primer capítulo, el seguimiento se realiza a partir de los datos proporcionados por el algoritmo de observación. Por lo tanto, es necesario detallar la forma de los mismos, como se hace a continuación.

En el caso del XPFCP y proceso de identificación (es decir, del sistema de seguimiento e identificación bidimensionales), las medidas provenientes del sistema de observación son las correspondientes a la intersección entre los volúmenes de la escena y el plano del suelo. Teniendo en cuenta el sistema de referencia utilizado, mostrado en la Figura 1.1, los ejes que forman el plano del suelo son el *X* y el *Z*, mientras que la coordenada de altura, *Y*, es constante en este caso, al estar todas las observaciones, como se ha dicho, en el plano del suelo, por lo que no se utiliza. El vector de medidas $\vec{y}_t$ es el siguiente:

$$\vec{y}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix} \tag{2.3}$$

En el caso del 3DPF se realiza un seguimiento tridimensional, por lo que los datos proporcionados por el Visual Hull no se limitan al plano del suelo. A la expresión del vector de medidas de la ecuación 2.3 se le añade la coordenada de altura z_t (es la *b* del apartado 2.1.2):

$$\vec{y}_t = \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \tag{2.4}$$

En el primer caso, el vector de medidas es de *2xN*, siendo *N* el número de puntos ocupados en la correspondiente rejilla de ocupación y 2 el número de dimensiones. En el segundo caso, es de *3xN*, aplicando un razonamiento análogo.

2.2 Algoritmo de estimación: Filtro de Partículas

Como ya se ha comentado, el sistema hace uso de un Filtro de Partículas Extendido con Proceso de Clustering (XPFCP), o de una versión significativamente modificada del mismo que da lugar al Filtro de Partículas Tridimensional (3DPF). Por lo tanto, se hace necesario un repaso del principio de funcionamiento del Filtro de Partículas (PF) básico, a partir de [Marrón08], donde este repaso se realiza de forma más extensa y profunda.

2.2.1 Principio de funcionamiento del PF

El PF es un estimador recursivo probabilístico ([Marrón08]). Su funcionamiento está basado en una representación discreta de la función de creencia a estimar $p(\vec{x}_t|\vec{y}_{1:t})$, mediante un conjunto de *n* partículas $S_t = \{\vec{s}_{i,t}\}$. La probabilidad de certidumbre de cada partícula dentro de la función de densidad de probabilidad (PDF, cuyo propósito es conocer cómo se distribuyen las probabilidades de un suceso) global se define mediante un peso normalizado asociado $\vec{w}_t = \{\tilde{w}_t^{(i)} \,/\, i = 1{:}\,n\}$. La función de "creencia" discreta es la salida del algoritmo de estimación ([Arulampalam02]):

$$p\left(\vec{x}_t^{(1:n)}\middle|\vec{y}_{1:t}\right) \cong S_t = \left\{\vec{x}_t^{(i)}, \widetilde{w}_t^{(i)}\right\}_{i=1}^{n} \tag{2.5}$$

El valor final de la estimación se extrae de esta función de creencia mediante distintas técnicas, como la envolvente de la PDF discreta o el análisis estadístico de la misma.

El PF utiliza una de las fórmulas de implementación en tiempo real del filtro de Bayes. Esta fórmula evita tanto la discretización del espacio de estado como la linealización del modelo del sistema (esto último es lo que ocurre en el Filtro de Kalman, por ejemplo). Por lo tanto la estimación es más fiable en todo tipo de situaciones, ya que no cuenta con restricciones como las mencionadas. La discretización de la creencia se realiza mediante el muestreo de Monte-Carlo. La función de creencia se calcula a partir de los pesos ([Arulampalam02], [Doucet97]):

$$p\left(\vec{x}_t^{(1:n)}\middle|\vec{y}_{1:t}\right) = \sum_{i=1}^{n} \widetilde{w}_t^{(i)} \cdot \delta(\vec{x}_t^{(i)}) \tag{2.6}$$

Los pesos se calculan de forma recursiva mediante ([Arulampalam02]):

$$w_t^{(i)} \to w(\vec{x}_{0:t}) = w(\vec{x}_{0:t-1}) \frac{p(\vec{y}_t|\vec{x}_t) \cdot p(\vec{x}_t|\vec{x}_{t-1})}{q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})} \tag{2.7}$$

Donde $p(\vec{y}_t|\vec{x}_t)$ es la probabilidad de tener $\vec{y}_t$ dado $\vec{x}_t$ (es decir, la verosimilitud), $p(\vec{x}_t|\vec{x}_{t-1})$ es el modelo probabilístico de actuación, y $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ es la función de aproximación a la creencia. Los pesos son normalizados a continuación ([Doucet97]):

$$\widetilde{w}_t^{(i)} = \frac{w_t^{(i)}}{\sum_{i=1}^{n} w_t^{(i)}} \tag{2.8}$$

2.2.2 Algoritmo SIS (Sequential Importance Sampling)

El algoritmo SIS es la primera versión del PF ([Marrón08]). Permite obtener las expresiones anteriores mediante las dos etapas usuales en estimadores recursivos: "predicción" y "corrección".

1. *Predicción*: Se propaga cada partícula al siguiente estado temporal mediante el modelo de estado no necesariamente lineal que caracteriza al sistema a estimar, y viene dado por la siguiente expresión, que es la forma determinística del modelo probabilístico de actuación $p(\vec{x}_t|\vec{x}_{t-1})$:

   $$\vec{x}_t = f(\vec{x}_{t-1}, \vec{u}_{t-1}, \vec{v}_{t-1}) \tag{2.9}$$

   Así se obtiene un valor a priori de la creencia $p(\vec{x}_t|\vec{y}_{t-1})$, que también está formada por un conjunto de muestras $S_{t|t-1} = \left\{\vec{x}_{t|t-1}^{(i)}, \widetilde{w}_{t-1}^{(i)}\right\}_{i=1}^{n}$.

2. *Corrección:* También conocida como ponderación de muestras o "importance sampling". Partiendo de la función de creencia a priori obtenida en la etapa de predicción, se obtiene la función de creencia a posteriori $p\left(\vec{x}_t^{(1:n)}\middle|\vec{y}_{1:t}\right)$, haciendo el

cálculo de los pesos, como ya se ha explicado anteriormente (ecuación 2.7). Lo que se pretende en este paso es evaluar la probabilidad de acierto de cada partícula en el paso de predicción del proceso de estimación.

Este algoritmo sin embargo deja de funcionar correctamente transcurrida una serie de iteraciones, debido al problema de degeneración del set, que se resuelve, como se explica en el apartado siguiente, añadiendo un paso al funcionamiento del PF. La degeneración del set de partículas es consecuencia del proceso de ponderación de las partículas, que provoca que la varianza de los pesos, $\Omega^2(w(\vec{x}_{0:t}))$ se incremente, llegando un momento en el que la mayoría adquiere un peso nulo, mientras que unas pocas terminan con probabilidades altas. En [Liu98] se trata este problema, y se define el concepto de número de partículas eficaces n_{eff} mediante la siguiente expresión:

$$n_{eff} = \frac{n}{1 + \Omega^2\left(w_{opt}\left(\vec{x}_{0:t}\right)\right)} \tag{2.10}$$

Donde $\Omega^2(w_{opt}(\vec{x}_{0:t}))$ hace referencia a la varianza de la función de ponderación óptima, que será, a partir de la ecuación 2.10:

$$w_{opt}(\vec{x}_{0:t}) = w_{opt}(\vec{x}_{0:t-1}) \frac{p(\vec{y}_t|\vec{x}_t) \cdot p(\vec{x}_t|\vec{x}_{t-1})}{q_{opt}(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})} \tag{2.11}$$

Estudios ([Doucet97]) demuestran que el mejor valor para la función de aproximación a la creencia $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ es $p(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$. Sin embargo, esta función es precisamente a la que se pretende llegar mediante el algoritmo de estimación, por lo que su cálculo no es posible. Así, el cálculo de n_{eff} mediante las ecuaciones (2.10 y 2.11) tampoco será posible, ya que aparece $q_{opt}(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$. Teniendo esto en cuenta, en [Doucet97] se introduce la expresión del valor aproximado del número de partículas eficaces $\hat{n}_{eff}$:

$$\hat{n}_{eff} = \frac{1}{\sum_{i=1}^{n}\left(\tilde{w}_t^{(i)}\right)^2} \tag{2.12}$$

El valor de $\hat{n}_{eff}$ es por tanto una medida de la eficiencia del PF, y debido a ello se utiliza como factor de calidad comparativo ([Marrón08]). En esta tesis también se utilizará como factor de calidad en el capítulo de resultados. Como puede verse en las ecuaciones 2.7, 2.8 y 2.12, esta eficiencia depende de la función $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ que se utilice en la implementación, lo que da lugar a un amplio número de propuestas respecto a la forma de esta función.

2.2.3 Algoritmo SIR (Sequential Importance Resampling)

Para solucionar el problema de degeneración del set de partículas, se añade una tercera etapa de selección o "resampling" al final del algoritmo SIS básico. Esta etapa se utiliza para eliminar las partículas con peso bajo y reproducir las que cuentan con un peso elevado. Esta modificación del algoritmo SIS da lugar al algoritmo SIR.

La etapa de selección del PF consiste en ([Marrón08]):

1. A partir del set obtenido a la salida del paso de corrección, se repite n_i veces cada partícula $\vec{x}_t^{(i)}$, de modo que:

$$E\{n_i\} = n \cdot \widetilde{w}_t^{(i)} \;\Rightarrow\; \sum_{i=1}^{n} n_i = n \tag{2.13}$$

   Siendo $E\{n_i\}$ la esperanza de n_i.
2. Se obtiene un conjunto de partículas con pesos $w_t^{(i)}$ con valor n^{-1}:

$$S_t = \left\{\vec{x}_t^{(i)}, \frac{1}{n}\right\}_{i=1}^{n} \tag{2.14}$$

El nuevo paso de selección todavía es problemático, ya que al final de los tres pasos del PF puede seguir ocurriendo que todas las partículas se concentren en la misma hipótesis. Es decir, $n_{i1} = n$ para una partícula y $n_{i2} = 0$ para el resto. Este problema, a la salida del paso de corrección, se conoce como "empobrecimiento del set de partículas", y depende de la función $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ elegida en la etapa de corrección y del algoritmo de resampling. Debido a ello, la definición de estos dos pasos es objeto de gran cantidad de estudios y propuestas como las expuestas en [Gordon93], [Liu98], [Merwe01], [Doucet01], [Arulampalam02], [Ristic04], [Douc05] o [Hol06]. Algunas de estas dan lugar al resampling residual, multinomial y sistemático, detallados en [Marrón08] e implementados en el código de la presente tesis.

De hecho, las modificaciones propuestas por el XPFCP tienen como finalidad asegurar que el número de partículas eficaces sea los suficientemente alto, es decir, que no aparezca el problema del empobrecimiento del set de partículas.

Debido al ruido de estado, las partículas se dispersan de forma natural, lo que justifica la no aparición del problema de empobrecimiento del set de partículas.

Es necesario comentar además que la salida de este algoritmo no aporta mucho, puesto que es un conjunto de hipótesis de posición, pero no hay una por objeto. Para resolver esto, como se verá en puntos posteriores, se incluye un proceso de clasificación que será diferente para cada uno de los dos algoritmos propuestos en esta tesis.

2.2.4 Algoritmo Bootstrap

El algoritmo Bootstrap es la versión más utilizada del algoritmo SIR. Como se ha comentado anteriormente, la función $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ elegida juega un papel importante en la eficiencia del PF. La función óptima no puede implementarse en la práctica, por lo que la opción más generalizada, debido a su sencillez, es utilizar como $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ la PDF (Probability Density Function o Función Densidad de Probabilidad) que caracteriza al modelo de actuación del sistema, que como se vio en el paso de predicción descrito anteriormente, es $p(\vec{x}_t|\vec{x}_{t-1})$.

La expresión para el cálculo de pesos, una vez simplificada, queda, sustituyendo $q(\vec{x}_t|\vec{x}_{0:t-1}, \vec{y}_{1:t})$ por $p(\vec{x}_t|\vec{x}_{t-1})$ en (2.7):

$$w_t^{(i)} \rightarrow w(\vec{x}_{0:t}) = w(\vec{x}_{0:t-1}) \cdot p(\vec{y}_t|\vec{x}_t) \tag{2.15}$$

La utilización de $p(\vec{x}_t|\vec{x}_{t-1})$ como $q(\vec{x}_t|\vec{x}_{0:t-1},\vec{y}_{1:t})$ tiene como inconveniente la eliminación de la información acerca de la verosimilitud $p(\vec{y}_t|\vec{x}_t)$ del modelo implicado en el proceso de estimación, por lo que pueden generarse pesos bajos para la mayor parte de partículas que representan la PDF a la salida del paso de corrección cuando la PDF de verosimilitud se sitúa lejos de la media de la creencia a priori $p(\vec{y}_t|\vec{x}_{t-1})$, como puede observarse en la Figura 2.5.

Esta situación es típica en procesos con bajo ruido de medidas, y basados en modelos de actuación inexactos, como los implicados en tareas de seguimiento como la propuesta.

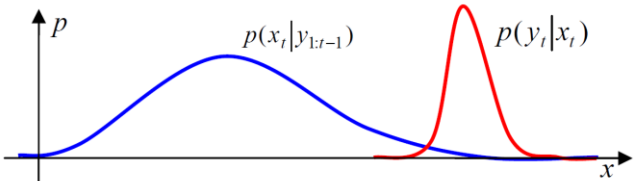


Figura 2.5. Representación del problema asociado a la elección de $p(\vec{x}_t|\vec{x}_{t-1})$ como función de aproximación a la creencia. La creencia a priori se representa en azul, y la verosimilitud en rojo.

El algoritmo resultante se conoce como "Bootstrap Filter" o Filtro Autónomo.

Este algoritmo Bootstrap incluye, además de los pasos mencionados, una etapa de inicialización para la primera iteración del algoritmo, ya que la función de creencia inicial es desconocida. En ella se genera un set de partículas con pesos idénticos y distribución $p(\vec{x}_0)$ uniforme en el espacio de estado.

A continuación puede verse el diagrama funcional del algoritmo Bootstrap así completado.

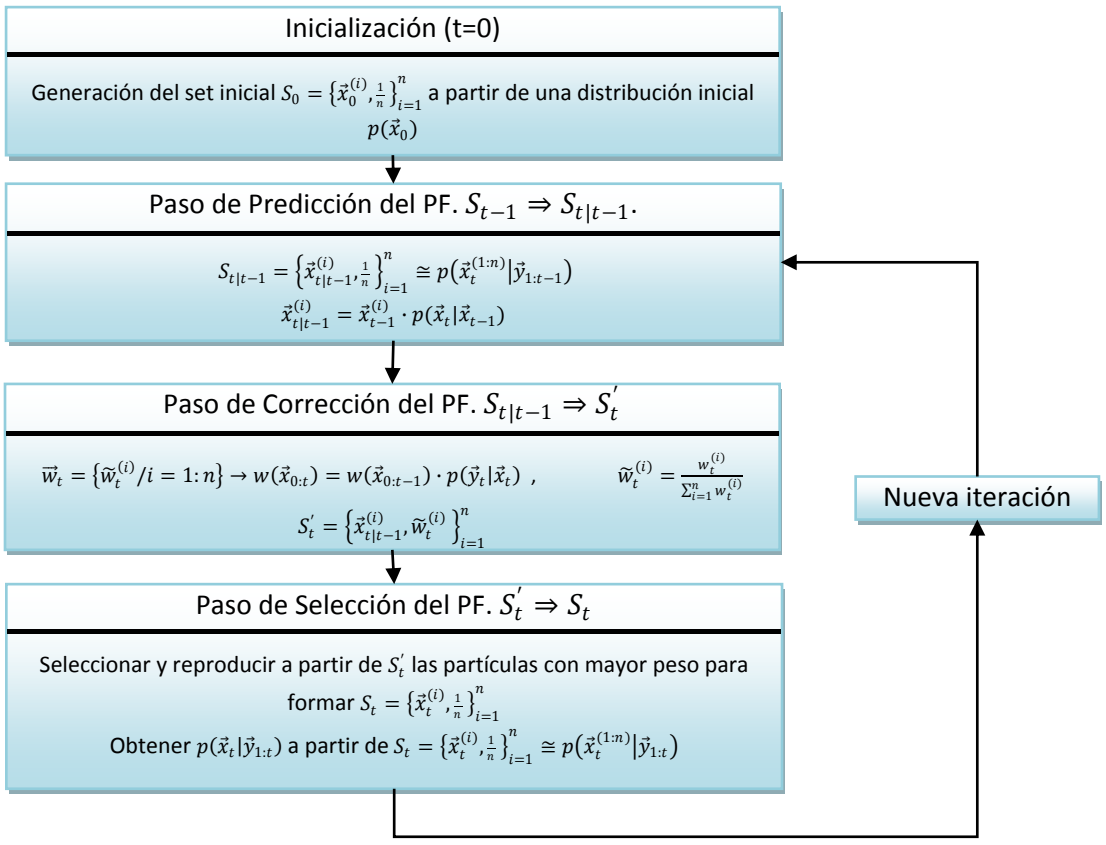


Figura 2.6. Diagrama funcional del algoritmo Bootstrap.

2.2.5. Datos proporcionados por el filtro

En el apartado 2.1.3 se ha visto cómo el sistema de observación proporciona al sistema de seguimiento los datos necesarios para que éste realice su tarea. A su vez, el seguidor proporciona información del entorno a su salida: la posición, velocidad y trayectoria de cada objeto presente en la escena. Esto se verá con más detalle en capítulos posteriores.

CAPÍTULO 3

SEGUIMIENTO BIDIMENSIONAL

En este capítulo se trata el proceso de seguimiento en dos dimensiones. Este proceso está formado por un seguidor basado en el Filtro de Partículas Extendido con Proceso de Clasificación (XPFCP o eXtended Particle Filter with Clustering Process) propuesto por [Marrón08] y una rutina para la identificación de robots en el espacio inteligente implementada en la presente tesis.

3.1 Filtro de Partículas Extendido con Proceso de Clasificación (XPFCP)

El XPFCP es la versión del PF implementada en el sistema como estimador multimodal. El XPF fue propuesto en [Koller-Meier01]. No cumple, sin embargo, los requerimientos de robustez hasta que en [Marrón08] se añade un proceso clasificación (CP), dando lugar al XPFCP que se utiliza en esta tesis de máster.

Debido a ello, a continuación se habla primero del XPF y sus problemas de robustez, para posteriormente describir el proceso de clasificación como solución a los mismos.

3.1.1 Filtro de Partículas Extendido (XPF)

El XPF añade una serie de modificaciones al algoritmo Bootstrap ya visto en el capítulo anterior, característico por su baja carga computacional y gran sencillez funcional. La finalidad es adaptar el filtro para que sea capaz de estimar el estado de varios sistemas. Para hacer esto no

es necesario extender el vector de estado, como se vio en algunos trabajos del estado del arte en el primer capítulo de la memoria. El XPF es además una de las pocas alternativas presentadas por la comunidad que cumple la especificación de multimodalidad ([Marrón08]).

3.1.1.1. Funcionalidad del XPF

El diagrama de funcionamiento del XPF se muestra en la Figura 3.1. Los pasos que han sido modificados respecto al PF básico (Figura 2.6) aparecen representados en color naranja. A continuación se hace un repaso de todos los pasos.

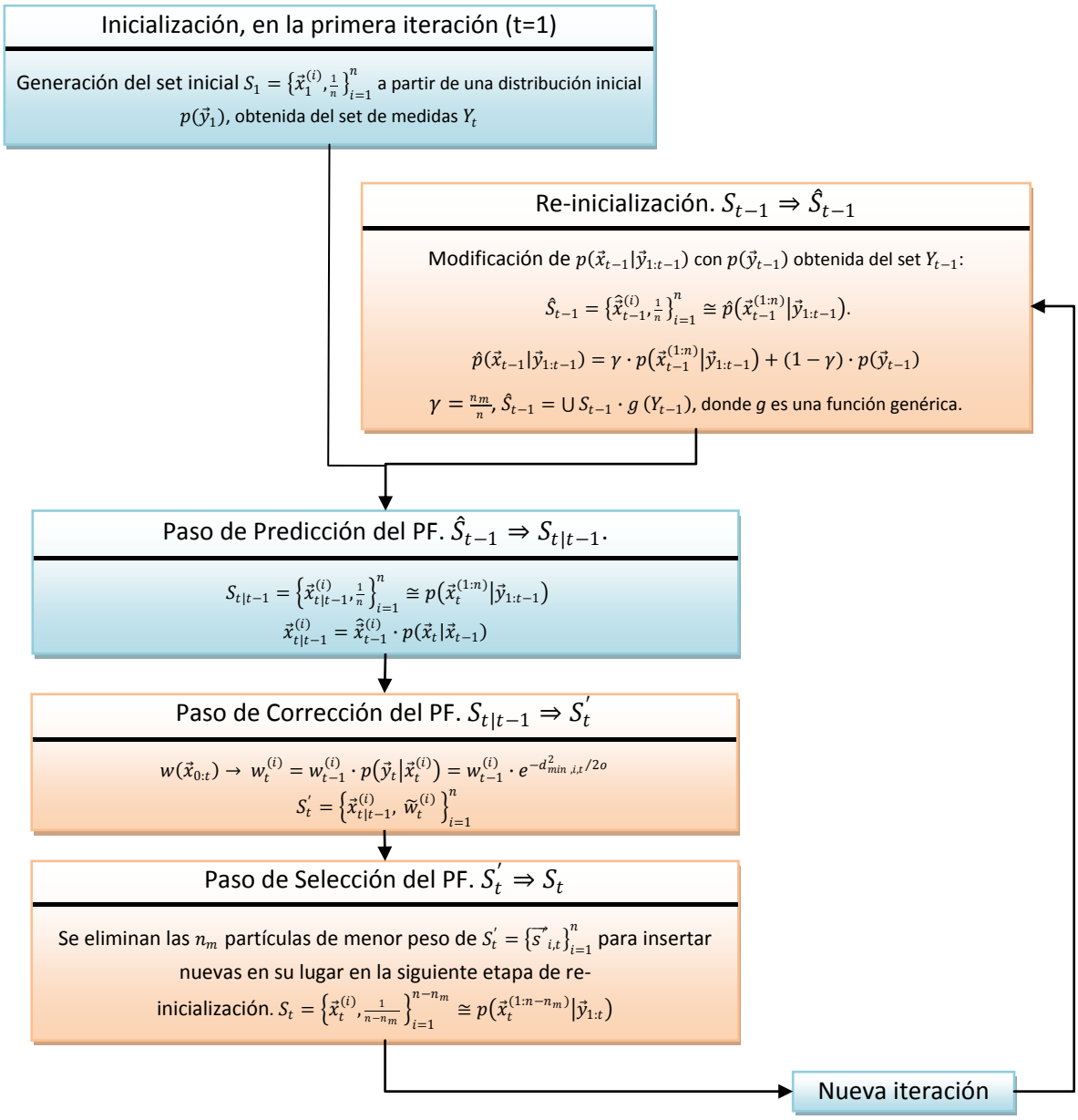


Figura 3.1. Flujograma del XPF.

a) Paso de re-inicialización

Para que la creencia $p(\vec{x}_t|\vec{y}_{1:t})$, y por lo tanto, el proceso de estimación, se adapte dinámicamente al número variable de objetos a seguir es necesario introducir una etapa de re-inicialización de la creencia en el lazo de ejecución original del PF. De esta forma se incorporan en cada instante picos de probabilidad en la creencia, los cuales representan a todas las entidades presentes en la escena. Así, se modifica $p(\vec{x}_{t-1}|\vec{y}_{1:t-1})$, que pasa a denominarse $\hat{p}(\vec{x}_{t-1}|\vec{y}_{1:t-1})$.

Este es el único método conocido ([Marrón08]) que hace posible la utilización de un único PF basado en vector de estado no extendido para realizar la estimación de un número variable de sistemas $k_{out,t}$.

Esto se consigue insertando n_m partículas en el set $S_{t-1} = \{\vec{s}_{i,t-1}\}_{i=1}^{n-n_m}$. Por lo tanto el set de salida de la etapa de re-inicialización, denotado por $\hat{S}_{t-1}$, cuenta con las *n* partículas. Las n_m partículas insertadas provienen de las medidas Y_{t-1}, es decir, de la densidad $p(\vec{y}_{t-1})$. Así, la primera iteración del XPF sólo se ejecuta una vez que se tiene el primer set de medidas Y_1, por lo que se corresponde con el instante de tiempo *t=1* y no *t=0*, como ocurre en el PF original, como puede verse en la Figura 2.6.

Según se explica en [Koller2000] y en [Marrón08], para la estimación se utiliza un modelo de velocidad constante (CV). Su definición matemática es la siguiente:

$$\vec{x}_t = \begin{bmatrix} x_t \\ z_t \\ vx_t \\ vz_t \end{bmatrix} = f(\vec{x}_{t-1,}\vec{v}_{t-1}) = A \cdot \vec{x}_{t-1} + \vec{v}_{t-1} = \begin{bmatrix} 1 & 0 & t_s & 0 \\ 0 & 1 & 0 & t_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ z_t \\ vx_t \\ vz_t \end{bmatrix} + \begin{bmatrix} v_{x,t-1} \\ v_{z,t-1} \\ v_{vx,t-1} \\ v_{vz,t-1} \end{bmatrix} \quad (3.1)$$

$$\vec{y}_t = \begin{bmatrix} x_t \\ z_t \end{bmatrix} = h(\vec{x}_{t,}\vec{o}_t) = C \cdot \vec{x}_t + \vec{o}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ z_t \\ vx_t \\ vz_t \end{bmatrix} + \begin{bmatrix} o_{x,t} \\ o_{z,t} \end{bmatrix} \quad (3.2)$$

Las componentes de velocidad vx_t y vz_t se inicializan con valor nulo en cada partícula. Este valor evoluciona con el tiempo automáticamente, hasta ajustarse a la velocidad del objeto que representa la partícula.

La re-inicialización permite que la creencia a priori $p(\vec{x}_t|\vec{y}_{1:t})$ obtenida después del paso de predicción incluya información de la verosimilitud $p(\vec{y}_{t-1}|\vec{x}_{t-1})$. Como se ha comentado en el capítulo anterior, dicha información está ausente en la función de aproximación de creencia $q(\vec{x}_t|\vec{x}_{0:t-1},\vec{y}_{1:t})$ del algoritmo Bootstrap, y es el hecho que generalmente produce los problemas de empobrecimiento del set. Como también se dijo en el capítulo 2 y se demuestra en el capítulo de resultados, esta modificación asegura que este filtro permita evitar el problema de empobrecimiento, aun utilizando una versión del filtro de partículas tan sencilla como la Bootstrap.

Matemáticamente, con la re-inicialización se sustituye la $p(\vec{x}_{t-1}|\vec{y}_{1:t-1})$ de entrada al paso de predicción del PF estándar, por la mostrada a continuación:

$$\hat{p}(\vec{x}_{t-1}|\vec{y}_{1:t-1}) = \gamma \cdot p\left(\vec{x}_{t-1}^{(1:n)}\middle|\vec{y}_{1:t-1}\right) + (1-\gamma)\cdot p(\vec{y}_{t-1}) \tag{3.3}$$

$$\gamma = \frac{n_m}{n} \tag{3.4}$$

Donde γ es el porcentaje de partículas que se insertan en el paso de re-inicialización (n_m) con respecto al número total de partículas (n). Este parámetro ha de ser suficientemente bajo para mantener el rigor teórico de la función de creencia. En [Marrón08] se explican los efectos de poner un valor alto en este parámetro.

b) Paso de predicción

El paso de predicción del XPF es idéntico al del algoritmo Bootstrap. Las partículas procedentes del paso de re-inicialización $\hat{S}_{t-1} = \left\{\hat{\vec{x}}_{t-1}^{(i)}, \frac{1}{n}\right\}_{i=1}^{n} \cong \hat{p}\left(\vec{x}_{t-1}^{(1:n)}\middle|\vec{y}_{1:t-1}\right)$ se propagan con el modelo de estado del sistema $p(\vec{x}_t|\vec{x}_{t-1}) \cong f\left(\vec{x}_{t-1,}\vec{v}_{t-1}\right)$. Como se ha comentado anteriormente, se utiliza un modelo de velocidad constante (CV, ecuación 3.1) como modelo de actuación, el cual es lineal y muy sencillo.

c) Paso de corrección

El paso de corrección del XPF también coincide con el del Bootstrap, con los inconvenientes de empobrecimiento del set asociados, comentados en el capítulo 2. En [Koller-Meier01] se propone utilizar como verosimilitud, para cada partícula, una gausiana cuya media es la proyección de la partícula en el espacio de medidas $\vec{y}_{1:t-1} \rightarrow h\left(\vec{x}_{t|t-1}\middle|\vec{o}_t\right)$, y como matriz de covarianza la del modelo de observación.

Con todo ello, el valor específico que se asigna a cada peso se corresponde con la siguiente expresión:

$$w_t^{(i)} = w_{t-1}^{(i)} \cdot p\left(\vec{y}_t\middle|\vec{x}_t^{(i)}\right) = w_{t-1}^{(i)} \cdot e^{-d_{min\,,i,t}^2/2o} \tag{3.5}$$

Donde $d_{min\,,i,t}$ es la mínima distancia entre la media de la gausiana anteriormente mencionada y la medida más próxima a ella de Y_t.

Los pesos son normalizados posteriormente mediante la expresión ya vista en el capítulo anterior:

$$\widetilde{w}_t^{(i)} = \frac{w_t^{(i)}}{\sum_{i=1}^{n} w_t^{(i)}} \tag{3.6}$$

d) Paso de selección

El paso de selección es idéntico al del Bootstrap, pero el número de partículas seleccionadas se reduce a $n - n_m$, para mantener n constante a lo largo de todo el proceso. Para conseguir esto hay que modificar el algoritmo de selección utilizado, como se explica en [Marrón08]. La salida final del estimador es por lo tanto es $S_t = \left\{\vec{x}_t^{(i)}, \frac{1}{n-n_m}\right\}_{i=1}^{n-n_m}$.

3.1.1.2. Puntos débiles

Como se demuestra en [Marrón08] y se resume a continuación, no se puede asegurar la fiabilidad del XPF tal cual se ha expuesto. El paso de re-inicialización tiene dos funciones: adaptar en *t* la creencia procedente de *t-1* a un número variable de hipótesis, y eliminar el problema de empobrecimiento del set asociado al algoritmo Bootstrap. Estas funciones dan los resultados esperados sólo si las diversas entidades presentes en el entorno tienen una calidad de sensado similar, ya que las medidas que se transforman en partículas en la re-inicialización se toman muestreando aleatoriamente el set completo de medidas. Si la densidad de medidas $p(\vec{y}_{t-1})$ se distribuye de forma desigual entre los distintos objetos, los pobremente sensados tendrán una presencia menor en la función de creencia.

Para resolver este problema, como se explicará a continuación, la solución propuesta en [Marrón08] añade un proceso de clasificación de medidas para evitar el empobrecimiento del set ante cualquier distribución o modelo de observación.

Por otro lado, el paso de corrección del XPF también incluye varios puntos débiles: el tiempo de cómputo depende del número total de partículas *n* y del número de medidas m_t, por lo que cuanto mejor se desee modelar la función de creencia, y mejor sensados estén los objetos presentes en la escena, más tardará el algoritmo en ejecutarse. Además, la función de verosimilitud utilizada para el cálculo de los pesos no garantiza el funcionamiento robusto del XPF, como pudo verse en la Figura 2.5. Utilizar el proceso de clasificación de medidas en la etapa de pesado, como se explicará posteriormente y se propone en [Marrón08], es una forma de abordar el problema satisfactoriamente.

3.1.2 Proceso de Clasificación (CP)

El proceso de clasificación (CP o Clustering Process) añadido al XPF completa el algoritmo propuesto en [Marrón08] y es utilizado en esta tesis para realizar tanto el seguimiento bidimensional (XPFCP) como el tridimensional (3DPF), ya que permite clasificar datos utilizando el número de dimensiones que se desee.

El clasificador utilizado (de los dos propuestos en [Marrón08]) es un derivado del algoritmo lineal de clasificación k-medias, denominado k-medias recursivo.

De forma general, la clasificación busca organizar un conjunto de datos en función de una serie de características, en un espacio de definición fijado por las mismas. A cada agrupación resultante se le denomina “grupo” o “clase”. Los datos asociados a cada clase son los miembros de la misma.

En la clasificación, la organización de las clases resulta en un conjunto k de grupos $G_{1:k}$ definidos de la siguiente manera:

$$G_{j=1:k} = \left\{\vec{g}_j, \vec{l}_{1:l_j,j} \;/\; j = 1{:}\,k\right\} = \left\{\vec{g}_j, L_j \,/\, j = 1{:}\,k\right\} \qquad (3.7)$$

En la expresión anterior, $L_j = \left\{\vec{l}_{i,j} \;/\; i = 1{:}\,l_j\right\}$ se refiere al conjunto de l_j miembros asociados a la clase j. El set de datos a clasificar se denota por $Y = \{\vec{y}_i / \; i = 1{:}\,m\}$. En la aplicación tratada en este trabajo es $\vec{y}_i = [x_t \; y_t]$. El vector característico que identifica a la clase j es $\vec{g}_j$, que suele ser el centroide de L_j. Así, tanto los datos a clasificar como los grupos

creados se definen en el espacio bidimensional cartesiano XZ. Es importante destacar que en el XPFCP el clasificador funciona a dos niveles: por uno realiza una agrupación de medidas en XZ (también conocido como clasificación de entrada, necesario para las etapas de re-inicialización y cálculo de pesos), y por otro realiza una agrupación de partículas (utilizando no sólo sus coordenadas XZ, sino también sus velocidades en cada una de esas coordenadas), dando lugar a las clases finales $G_{1:k,t|out}$, que son la salida del algoritmo.

3.1.2.1. K-medias secuencial

En este trabajo se propone la utilización de un clasificador basado en el algoritmo k-medias: el k-medias secuencial ([Marrón08]). Se trata de un algoritmo no supervisado, recursivo, enlazado, booleano, y basado en la distancia euclidea.

El k-medias básico se caracteriza por un bajo tiempo de ejecución. El algoritmo utilizado se basa en él, por lo que encaja en el objetivo de implementar el algoritmo de seguimiento en tiempo real. Las reglas que sigue para obtener *k* clases son las siguientes:

1. Elegir *k* datos del set *Y* para utilizarlos como valor inicial de los centroides de los grupos a crear.
2. Calcular la distancia entre cada dato y todos los centroides. Asignar el $\vec{y}_i$ al cluster $\vec{g}_{iter,j}$ situado a la menor distancia.
3. Una vez asignado todo *Y*, se re-calculan los centroides de cada clase.
4. Se analiza si ha habido algún cambio en el valor de los centroides.
5. Si los ha habido, se repite el proceso desde el paso 2, hasta llegar a un número de iteraciones máximo, definido por el diseñador. En caso contrario, se eliminan los conjuntos sin miembros asociados, y el algoritmo concluye.

Hay que tener en cuenta que el k-medias básico necesita como parámetro en número de clases *k* a clasificar, por lo que es necesario conocerlo de antemano. Esto no es posible en el sistema aquí tratado. El algoritmo k-medias extendido resuelve este problema. En él se crea un nuevo grupo en el proceso de segmentación si alguno de los datos (ya sean medidas o partículas) tiene una característica de distancia muy diferente a la de los grupos existentes. Esto hace necesario la inclusión de un parámetro de distancia límite: *distM*, haciendo al k-medias extendido sensible ante variaciones de su valor. Además, en esta versión del algoritmo el ruido no es filtrado correctamente, y la covergencia del algoritmo no puede asegurarse, debido a la naturaleza iterativa del mismo (ver las demostraciones correspondientes en [Marrón08]).

Para resolver estos problemas, en [Marrón08] se realizan una serie de modificaciones al k-medias extendido, dando lugar a un nuevo algoritmo llamado k-medias secuencial, y se crea un proceso de validación que se añade al segmentador.

En el k-medias secuencial se realiza la predicción de los centroides $\vec{g}_{i:k,t}$ mediante el mismo modelo de velocidad constante ya comentado CV (ecuaciones 3.1 y 3.2), para utilizar estos centroides como valores iniciales en los grupos $\vec{g}_{0,1:k}$, y así asegurar la convergencia de la clasificación en un número reducido de iteraciones, como se demuestra en [Marrón08].

Para poder implementar este proceso de predicción, es necesario mantener una identificación inequívoca a lo largo del tiempo de cada una de las clases. Esto se consigue asignando al grupo $\tau_j : \{\vec{y}_{i,t} \ / \ i = 1: l_j\} \in L_j$ las medidas extraídas del objeto τ_j. Así, el identificador de grupo τ_j se

convierte en identificador de objetos. Esto es lo que permite, mediante un algoritmo de asociación NN (Nearest Neighbour, Vecino Más Cercano), relacionar grupos de medidas con grupos de partículas, algo fundamental en este proceso de seguimiento bidimensional. En la Figura 3.2 se muestra esquemáticamente el k-medias secuencial.

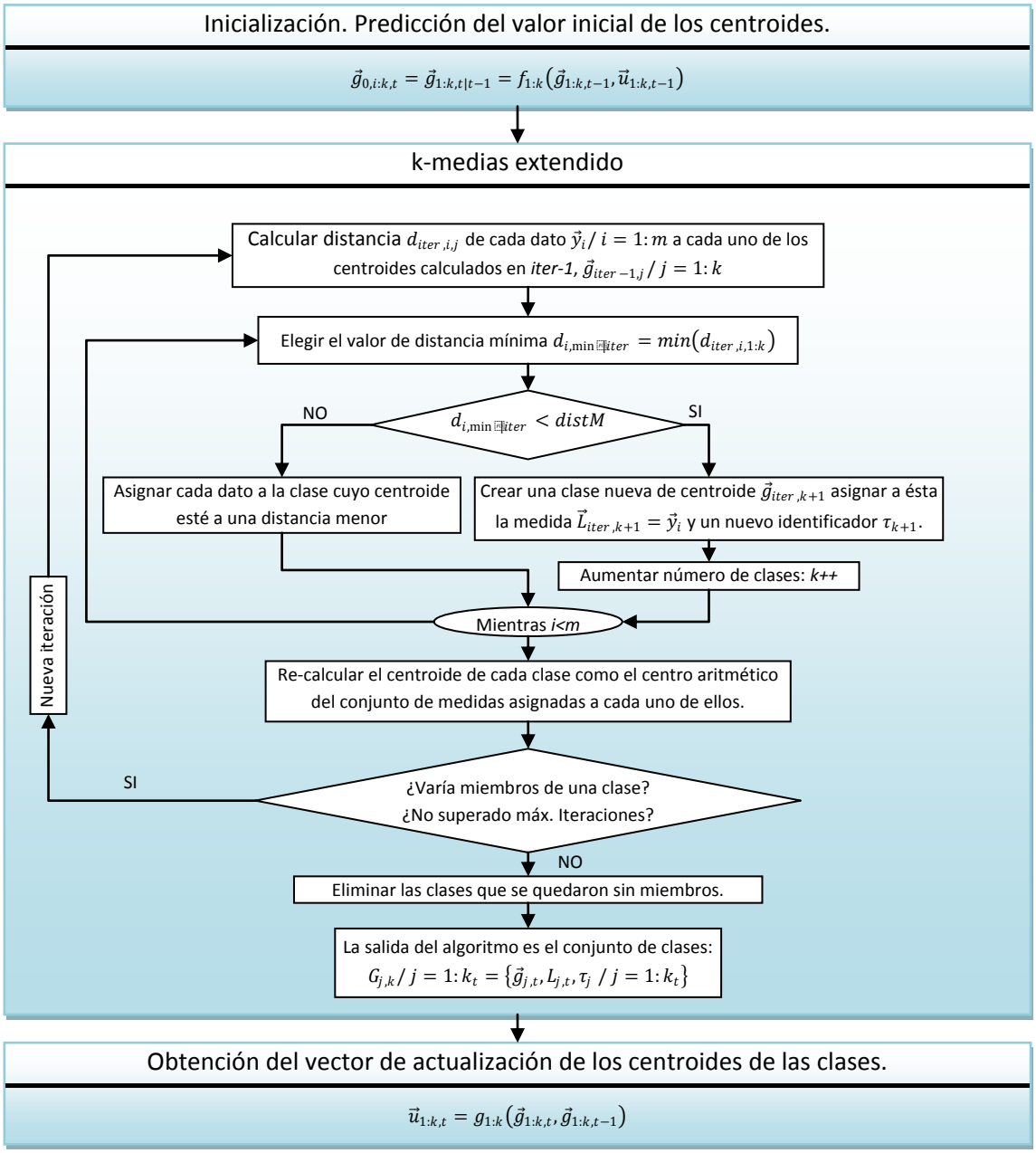


Figura 3.2. Diagrama funcional del k-medias secuencial.

En la Figura 3.2 puede verse que el k-medias secuencial añade dos etapas al k-medias extendido: la de predicción de centroides y la de actualización de los mismos. Además, se añade a las clases $G_{j,k}$ el identificador de clase τ_j antes mencionado.

3.1.2.2. Proceso de validación

La otra etapa añadida en [Marrón08] para resolver los problemas del k-medias extendido, el proceso de validación de clases, se inserta al final del segmentador, con el objetivo de

incrementar la robustez del mismo frente al ruido (si bien este ruido afectaría sólo a la clasificación de medidas y no a la de partículas), y reducir su sensibilidad a parámetros como *distM*. El objetivo del proceso es mantener una variable de validación $\beta_{j,t}$ asociada a cada grupo $G_{j,t} / j = 1{:}\, k$, indicando cómo de fiable es la existencia y permanencia de este grupo en función de dos características del grupo:

1. Su proximidad $d_{t,j,jold}$ en el espacio de clasificación a la clase $G_{jold,t-1}$ más semejante de las obtenidas en el instante anterior *t-1*. Es decir, se busca la clase en *t* más cercana a la obtenida en *t-1*.
2. Su verosimilitud $p_{j,1:t}$. Para suavizar el efecto de la verosimilitud instantánea $p_{j,t}$ se utiliza un factor de olvido normalizado α_{valid}, dando lugar a la siguiente expresión:

$$p_{j,1:t} = \alpha_{valid} \cdot p_{j,t} + (1 - \alpha_{valid}) \cdot p_{j,1:t-1} \; / \; j = 1{:}\, k_t \qquad (3.8)$$

   Este suavizado persigue robustecer el algoritmo ante variaciones instantáneas de la distribución que caracteriza a *Y*.

En función de esas dos características (si se superan los umbrales con histéresis de distancia y probabilidad) se actualiza linealmente el contador de validación $\beta_{j,t}$, como se muestra en la Figura 3.3:

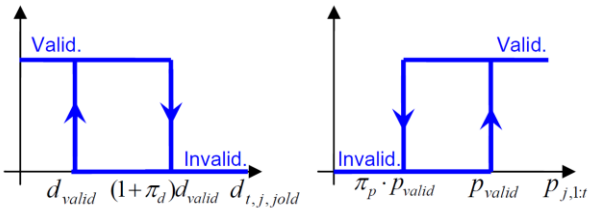


Figura 3.3. Representación del proceso de validación. A la izquierda se muestra la parte referida a la condición de distancia, y a la derecha, a la de verosimilitud.

Los factores de histéresis normalizados para las condiciones de verosimilitud y proximidad se denotan, respectivamente, por π_p y π_d. El límite de condición de verosimilitud se define linealmente, en función del número de clases generadas por el clasificador. Se satura a un mínimo:

$$p_{valid} = \frac{p_{valid\,|max}}{k_{valid}} \qquad (3.9)$$

La aplicación del proceso de validación al k-medias secuencial, conlleva que ([Marrón08]):

1. Gracias al identificador τ_j, el cálculo de la distancia $d_{t,j,jold}$ se realiza sólo respecto al centroide de la clase con su mismo identificador.
2. El valor de verosimilitud asignado a cada grupo se calcula en función del número de datos asociados a este, $l_{j,t} \; / \; L_j = \{\vec{l}_{i,j,t} \; / \; i = 1{:}\, l_{j,t}\}$, llegando a:

$$p_{j,t} = \frac{l_{j,t}}{m_t} \; / \; j = 1{:}\, k_t \qquad (3.10)$$

   Ponderando mediante el factor de olvido α_{valid}:

$$p_{j,1:t} = \alpha_{valid} \cdot \frac{l_{j,t}}{m_t} + (1 - \alpha_{valid}) \cdot p_{j,1:t-1} \;/\; j = 1{:}\,k_t \qquad (3.11)$$

Finalmente, una clase será válida cuando, siendo inicialmente no válida, el contador de validación $\beta_{j,t}$ supere el umbral determinado. Una clase se invalida si, siendo inicialmente válida, el contador de validación $\beta_{j,t}$ se hace menor que cero.

Con todo ello, el diagrama funcional del proceso de validación aplicado al k-medias secuencial se muestra en la Figura 3.4.

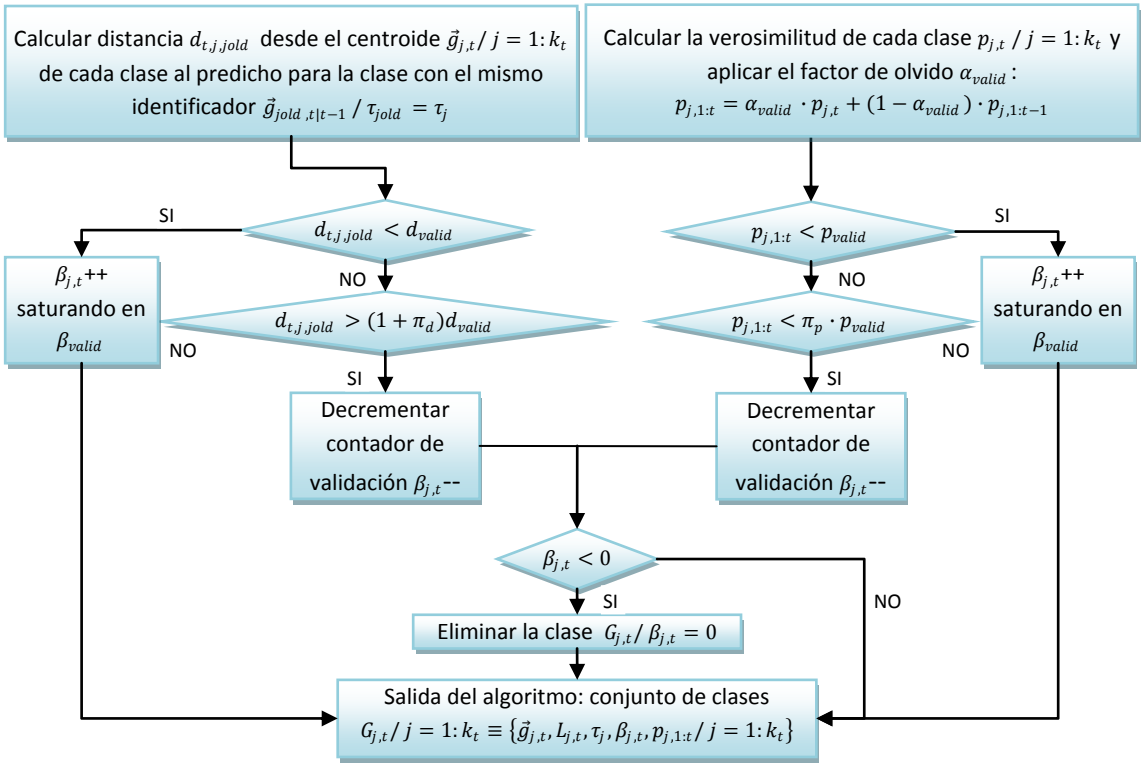


Figura 3.4. Diagrama funcional del proceso de validación aplicado al k-medias secuencial.

El aspecto final del algoritmo de clasificación utilizado, k-medias secuencial con proceso de validación, puede verse en la Figura 3.5.

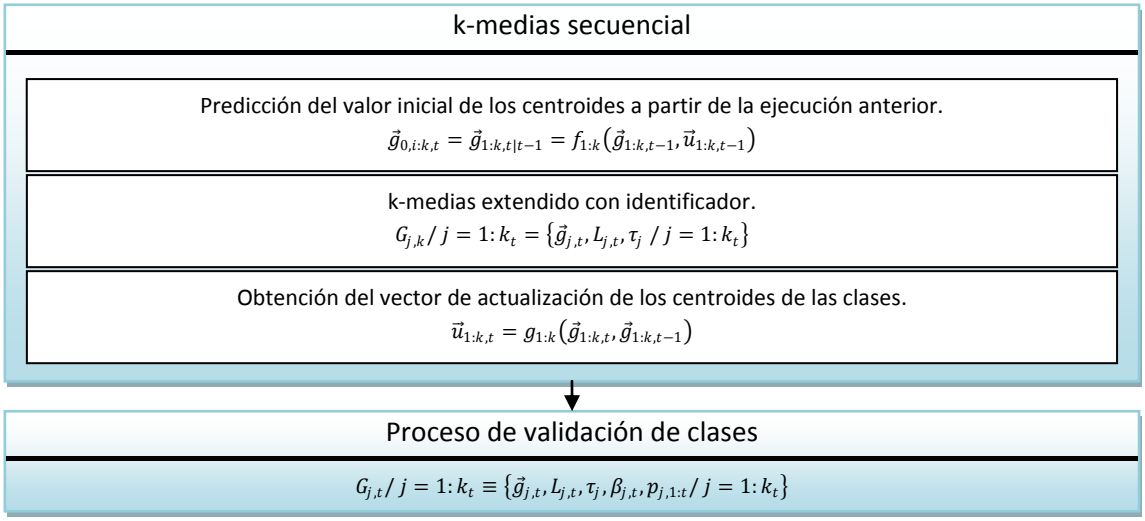


Figura 3.5 Diagrama funcional del algoritmo final de clasificación utilizado.

El ajuste y la selección de todos los parámetros del clasificador así obtenido ($distM$, π_d, π_p, d_{valid}, p_{valid} y α_{valid}) se estudia en [Marrón08].

3.1.3 Algoritmo final de seguimiento en dos dimensiones

La unión del XPF con el proceso de clasificación da lugar al XPFCP, el estimador utilizado para realizar el seguimiento bidimensional. El esquema final del mismo se muestra en la Figura 3.6:

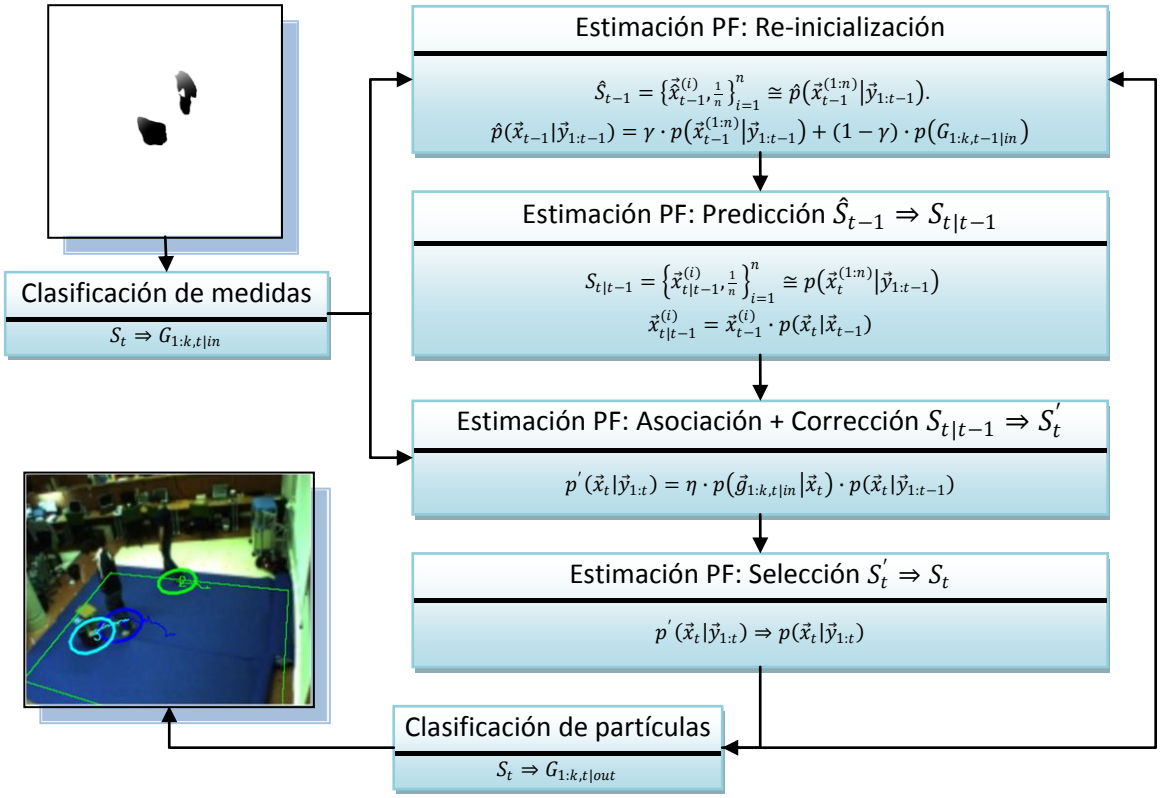


Figura 3.6. Diagrama funcional del XPFCP.

3.2 Proceso de identificación

A partir de [Pizarro08], en esta tesis se propone un método para identificar qué grupos de partículas se corresponden con objetos de tipo robot (uno en este caso) controlado por el espacio inteligente, con respecto al resto de obstáculos aparecidos en la rejilla de ocupación.

Para alcanzar este objetivo de identificación, normalmente se utiliza un modelo de apariencia combinado con el hecho de que un mismo objeto no cambia su posición de forma significativa en dos iteraciones consecutivas, suponiendo una frecuencia de muestreo lo suficientemente alta. También se suele utilizar una costosa identificación de trayectorias globales ([Pizarro08]).

El uso de cualquiera de estas dos soluciones se debe a que en la mayor parte de los trabajos previos no está disponible un modelo de movimiento lo suficientemente bueno para definir el objeto seguido. Sin embargo, la odometría proporciona uno muy preciso en trayectorias pequeñas. Los encoders proporcionan una trayectoria cuyo error de estimación crece indefinidamente con la longitud de la misma.

Utilizando la odometría es fácil obtener la probabilidad con que un grupo de partículas a la salida del XPFCP de una clase determinada está asociado con el robot. Además, esta probabilidad debe incluir las mismas propiedades de incertidumbre que muestran los encoders.

3.2.1 Principio de funcionamiento

Como se ha visto en puntos anteriores, el sistema de seguimiento bidimensional proporciona a su salida, entre otros datos, la posición estimada en coordenadas absolutas de cada uno de los objetos presentes en la escena, con lo que es posible reconstruir la trayectoria seguida por estos objetos utilizando las posiciones de instantes de tiempo pasados.

Por otra parte, como ya se ha comentado, la odometría del robot proporciona información en coordenadas relativas sobre la trayectoria seguida por el mismo, si bien el error de posición asociado aumenta indefinidamente.

En [Pizarro08] se propone comparar la trayectoria del robot, obtenida mediante una reconstrucción a partir de la odometría, con las trayectorias de cada uno de los elementos presentes en el entorno, a lo largo de los últimos *TAMBUFFER* instantes de tiempo.

Dado que, como anteriormente se ha mencionado, el error de odometría crece permanentemente, se propone como solución hacer coincidir, cada *TAMBUFFER* iteraciones, la posición y orientación estimadas mediante el XPFCP de cada elemento presente en la escena con la trayectoria del robot reconstruida a partir de la odometría. Con esto se consigue eliminar el error de odometría cada periodo *TAMBUFFER*, suponiendo que la estimación dada por el XPFCP sea fiable, y transformar las coordenadas relativas que ofrece el robot en coordenadas absolutas.

Por último, se calcula la distancia acumulada entre la trayectoria estimada con el XPFCP para cada clase y la reconstruida a partir de la odometría, en las últimas *TAMBUFFER* iteraciones, y se determina en a partir de ello cuál es la clase que tiene una probabilidad mayor de ser el robot, mediante un proceso de votación descrito en el siguiente punto.

3.2.2 Estructura del sistema

El diagrama funcional del proceso de identificación descrito se muestra en la Figura 3.7 y se explica a continuación.

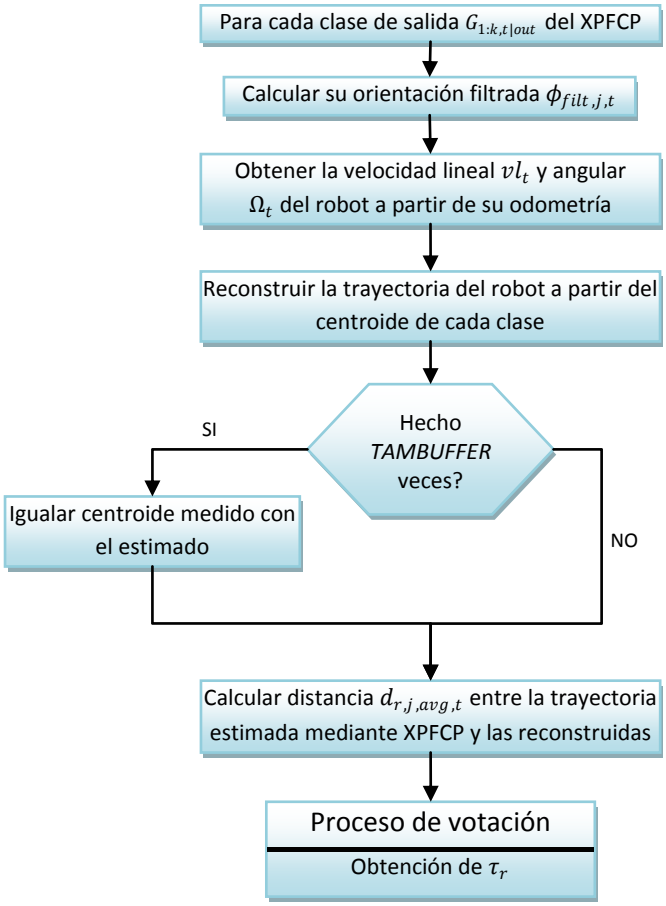


Figura 3.7. Esquema de funcionamiento del proceso de identificación.

a) Cálculo de la orientación filtrada

Como se ha visto en el punto 3.1, el vector de estado a estimar incluye las componentes bidimensionales de velocidad vx_t y vz_t. Dado que el XPFCP permite realizar el seguimiento de múltiples objetos, se tienen las componentes de velocidad para cada uno de ellos. A partir de estas componentes es posible calcular la orientación ϕ_j de cada elemento j sobre el que se realiza el seguimiento mediante la siguiente expresión:

$$\phi_j = \tan^{-1}\left(\frac{vz_t}{vx_t}\right) / j = 1{:}\, k_t \tag{3.12}$$

Esta orientación ϕ_j es filtrada promediándola a lo largo de las últimas *ORBUFFER* iteraciones, según la expresión:

$$\phi_{filt,j,t} = \frac{\sum_{i=1}^{ORBUFFER} \phi_i}{ORBUFFER} / j = 1{:}\, k_t \tag{3.13}$$

b) Obtención de la velocidad lineal y angular del robot

Para reconstruir la trayectoria del robot a partir de una posición y orientación inicial es necesario conocer la velocidad lineal y angular del robot vl_t y Ω_t, respectivamente, que son calculadas a partir de los dos últimos instantes de tiempo según las siguientes expresiones:

$$vl_t = \sqrt{\left(odo_{x,t} - odo_{x,t-1}\right)^2 + \left(odo_{z,t} - odo_{z,t-1}\right)^2} \tag{3.14}$$

$$\Omega_t = odo_{\phi,t} - odo_{\phi,t-1} \tag{3.15}$$

Donde odo_x, odo_z y odo_ϕ son las coordenadas x e y, y orientación relativas del robot proporcionadas por la odometría, respectivamente.

c) Reconstrucción de la trayectoria del robot a partir del centroide de cada clase

La trayectoria del robot se reconstruye mediante el modelo de movimiento asociado al mismo, suponiendo por razones de simplicidad un robot sencillo de ruedas diferenciales que se mueve en el plano del suelo. Este modelo de movimiento se corresponde con la siguiente expresión:

$$\vec{x}_{r,t} = \begin{bmatrix} x_{r,t} \\ z_{r,t} \\ \phi_{r,t} \end{bmatrix} = \vec{x}_{r,t-1} + \begin{bmatrix} (vl_t + w_t^1) \cdot \cos\left(\phi_{r,t-1} + \Omega_t + w_t^2\right) \\ (vl_t + w_t^1) \cdot \sin\left(\phi_{r,t-1} + \Omega_t + w_t^2\right) \\ \Omega_t + w_t^2 \end{bmatrix} \tag{3.16}$$

Donde $\vec{x}_{r,t}$ es el vector de estado del robot, y vl_t y Ω_t son la velocidad lineal y angular del robot, respectivamente, ya definidas en el apartado b). El ruido de odometría se modela mediante un proceso gausiano $(w_t', w_t^2)^t$, estocásticamente independiente respecto de $\vec{x}_{r,t-1}$.

El estado inicial $\vec{x}_{r,0}$ a partir del cual se realiza la reconstrucción, es el estimado para la clase j correspondiente mediante el XPFCP, con la orientación inicial $\phi_{filt,j,t}$ calculada por el procedimiento descrito en el apartado a). Esto puede verse gráficamente en la Figura 3.8.

La trayectoria es reconstruida durante *TAMBUFFER* iteraciones consecutivas. Cuando ha transcurrido ese número de iteraciones, se vuelve igualar el vector de estado del robot al vector de estado estimado por el XPFCP, repitiéndose el proceso.

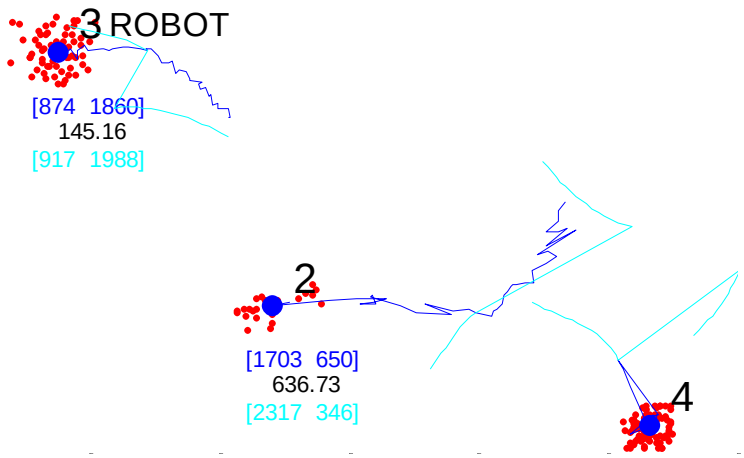


Figura 3.8. Evolución de las trayectorias estimadas mediante el XPFCP (en azul) y de las reconstruidas mediante la odometría (cyan) en tres clases distintas. Puede observarse, sobre todo en las clases con identificador 3 y 4, como una vez transcurridas *TAMBUFFER* iteraciones, el vector de estado reconstruido mediante odometría se iguala al vector de destado estimado con el XPFCP. Entre corchetes azules se representa la posición del objeto en el entorno. Entre corchetes cyan la de la reconstruida mediante odometría. En negro, la distancia entre ambas.

d) Cálculo de la distancia entre trayectorias estimadas y reconstruidas

Se proponen dos métodos para calcular la distancia $d_{r,j,t}$ entre un punto la trayectoria estimada y la reconstruida, como se exponen a continuación.

1. Dado que se supone un ruido de odometría gausiano $(w_t^{'}, w_t^2)^t$, es posible tener en cuenta las medias y desviaciones típicas del mismo para calcular la distancia de Mahalanobis acumulada a lo largo de las últimas *TAMBUFFER* iteraciones. La distancia de Mahalanobis entre dos estados $\vec{x}_{r,t}$ y $\vec{x}_t$ viene dada por ([Pizarro08]):

$$d_{r,j,t} = d_M(\vec{x}_{r,t}, \vec{x}_t) = (\vec{x}_{r,t} - \vec{x}_t)^T \Sigma_t^{-1} (\vec{x}_{r,t} - \vec{x}_t)^T \tag{3.17}$$

   Donde Σ_t es la matriz de covarianza del error, en un instante *t* determinado. Es necesario calcular esta matriz, mediante el proceso descrito a continuación.

   Para el propagar la covarianza del error se utilizan el siguiente procedimiento: dada la matriz de propagación del error, definida por la siguiente expresión ([Pizarro08]):

$$\Sigma_u = \begin{pmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_\Omega^2 \end{pmatrix} \tag{3.18}$$

   Donde σ_v^2 y σ_Ω^2 son las covarianzas del ruido de velocidad lineal y angular, respectivamente. Y dada también la matriz de covarianza inicial:

$$\Sigma_0 = \begin{pmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\Omega^2 \end{pmatrix} \tag{3.19}$$

   Y las matrices:

$$J_x = \frac{\delta x}{\delta x_{u-1}} = \begin{pmatrix} 1 & 0 & -vl_t \cdot \sin(\phi_{t-1} + \Omega_t) \\ 0 & 1 & vl_t \cdot \cos(\phi_{t-1} + \Omega_t) \\ 0 & 0 & 1 \end{pmatrix} \tag{3.20}$$

$$J = \frac{\delta x}{\delta u} = \begin{pmatrix} \frac{\delta x}{\delta v} & \frac{\delta x}{\delta \Omega} \\ \frac{\delta y}{\delta x} & \frac{\delta y}{\delta \Omega} \\ \frac{\delta \sigma}{\delta x} & \frac{\delta \sigma}{\delta \Omega} \end{pmatrix} \tag{3.21}$$

   La matriz de covarianza del error, en un instante *t* determinado, viene dada por la siguiente expresión ([Pizarro08]):

$$\Sigma_t = J_x \Sigma_{t-1} J_x^T + J_t \Sigma_u J_t^T \tag{3.22}$$

   En la Figura 3.9 se puede apreciar gráficamente el efecto del incremento del error de odometría a lo largo de una trayectoria, representado mediante elipses de error. La posición reconstruida (en verde) puede estar en cualquier punto dentro de las mismas. En rojo se muestra la posición real del robot.

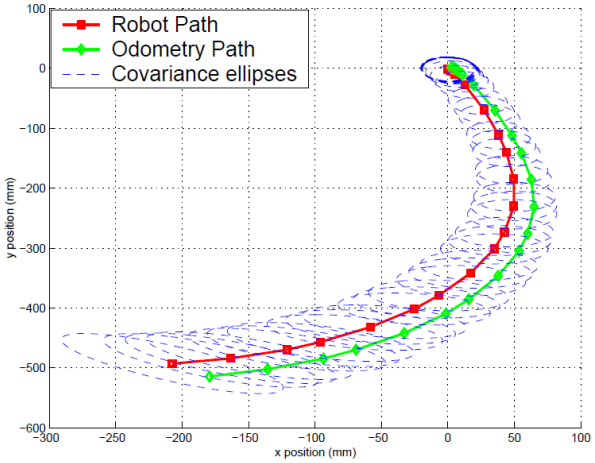


Figura 3.9. Evolución de la incertidumbre de la posición del robot a lo largo de una trayectoria.

2. El otro método propuesto es la utilización de distancia euclidea en lugar de la distancia de Mahalanobis, ya que es más simple de calcular. Basta con sustituir la matriz de covarianza Σ_t por la matriz identidad *I* en la ecuación 3.17, lo que da lugar a la siguiente expresión:

$$d_{r,j,t} = d_E(\vec{x}_{r,t}, \vec{x}_t) = (\vec{x}_{r,t} - \vec{x}_t)^T I (\vec{x}_{r,t} - \vec{x}_t)^T \qquad (3.23)$$

Al terminar este paso se han obtenido una serie de distancias $d_{r,j,t}$ entre un punto la trayectoria estimada de cada clase j y la reconstruida a partir del primer punto de la misma mediante la odometría del robot, en el instante de tiempo *t*. Para obtener la distancia acumulada promedio se utiliza la siguiente expresión:

$$d_{r,j,avg,t} = \frac{1}{TAMBUFFER} \sum_{i=1}^{TAMBUFFER} d_{r,j,i} \qquad (3.24)$$

Cuanto mayor sea esta distancia $d_{r,j,avg,t}$, mayor la diferencia entre trayectorias, y por tanto menor la probabilidad de ser robot. En la Figura 3.10 se muestra un ejemplo con datos reales:

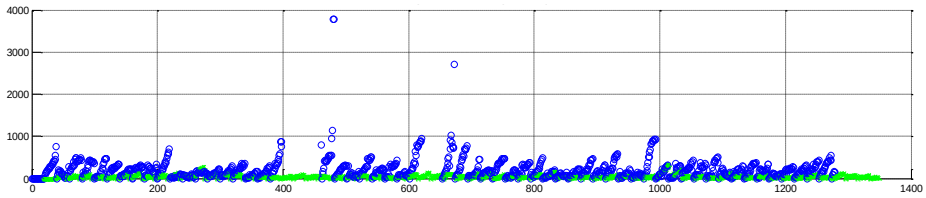


Figura 3.10. En verde: distancia acumulada entre la trayectoria reconstruida y estimada de un robot. En azul: distancia acumulada entre la trayectoria reconstruida y estimada de una persona.

e) Proceso de votación

Para determinar la identidad τ_r del robot (que constituye la salida final del proceso de identificación) se emplea, a partir de los datos relativos a las distancias entre trayectorias obtenidos anteriormente, un proceso de votación.

Este proceso de votación, como su nombre indica, está basado en votos. Existe un voto χ_j asociado a cada clase *j* seguida por el XPFCP. Inicialmente todos los votos valen cero.

Se recompensa a la clase cuya trayectoria sea más parecida a la del robot (es decir, la que tenga una menor distancia $d_{r,j,avg,t}$ asociada) incrementando su voto χ_j *VOTOPOS* veces. Por el contrario, se penaliza a la clase cuya trayectoria difiere en mayor medida de la del robot (por lo que tiene una mayor distancia $d_{r,j,avg,t}$ asociada) decrementando su voto χ_j *VOTONEG* veces. Los votos asociados a las clases que no se corresponden ni con la mayor distancia entre trayectorias ni con la menor se dejan como estaban.

El voto asociado nunca se hace negativo, por lo que el mínimo valor asociado es cero. No existe un límite positivo de votos asociados a cada clase. Esto puede verse en la Figura 3.11:

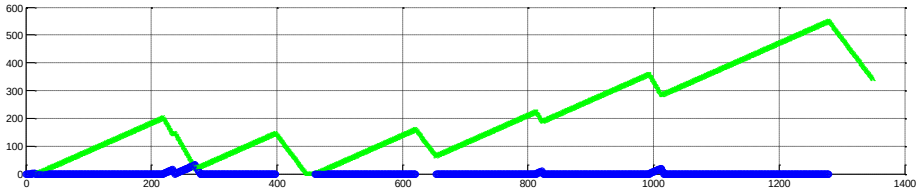


Figura 3.11. Evolución del voto asociado a dos clases distintas (la clase con voto verde es el robot, y la clase con voto azul es una persona) en un experimento real.

El identificador del robot τ_r (y salida final de esta rutina) es el de la clase que tenga un número mayor de votos asociado χ_j en ese instante de tiempo.

3.3 Conclusiones

En este capítulo se ha hecho un repaso del XPFCP propuesto por [Marrón08] para realizar el seguimiento bidimensional de múltiples objetos, así como de un proceso de identificación de robots inicialmente propuesto por [Pizarro08], ya que ambos serán puestos a prueba en el capítulo 5 en un espacio inteligente, constituyendo el sistema de seguimiento bidimensional completo.

El XPFCP ya había sido testado en tiempo real, sin embargo no en un espacio inteligente. El proceso de identificación hasta ahora sólo había sido comprobado mediante simulaciones en el entorno Matlab.

Una vez hecho un repaso de la teoría de funcionamiento que hay detrás del sistema de seguimiento bidimensional se extraen las siguientes conclusiones:

- El XPF es uno de las pocas propuestas que cumplen la especificación de multimodalidad.
- Sin embargo presenta el conocido como "problema del empobrecimiento del set", solucionado en [Marrón08] mediante la inclusión de pasos de clasificación de medidas y partículas.
- El clasificador utilizado es una modificación del k-medias, dando lugar al k-medias secuencial con proceso de validación, siendo un algoritmo iterativo y sin número de clases a clasificar prefijado.

- El proceso de identificación compara las trayectorias de cada clase de salida del XPFCP con la del robot, para así determinar cuál es la más parecida y obtener el identificador del robot.

CAPÍTULO 4

SEGUIMIENTO TRIDIMENSIONAL

En este capítulo se trata el procedimiento de diseño y distintas alternativas del proceso de seguimiento en tres dimensiones, realizado mediante un filtro de partículas tridimensional o 3DPF a partir de la idea del XPFCP visto en el capítulo anterior.

4.1 Filtro de Partículas Tridimensional (3DPF)

Como se ha visto en el capítulo 1, existen distintas tendencias para llevar a cabo el seguimiento de múltiples objetos. En esta tesis se propone realizar dicha tarea mediante un único filtro de partículas.

Esto tiene ventajas, como la obtención de un tiempo de ejecución constante frente a variaciones del número de objetos seguidos ([Marrón08]). Sin embargo, también plantea inconvenientes. El más importante es la necesidad de mantener siempre la representación de todas las hipótesis sobre las que se realiza el seguimiento en la creencia multimodal representada por el set de partículas. Si se crea una distribución desigual por las partículas entre las distintas hipótesis se hace vulnerable al sistema frente a perturbaciones, por ejemplo, en el modelo de observación.

Para evitar este problema se propone el algoritmo que se muestra en la

Figura 4.1 y se explica a lo largo de este capítulo. En naranja se muestran los pasos con modificaciones respecto al algoritmo XPFCP.

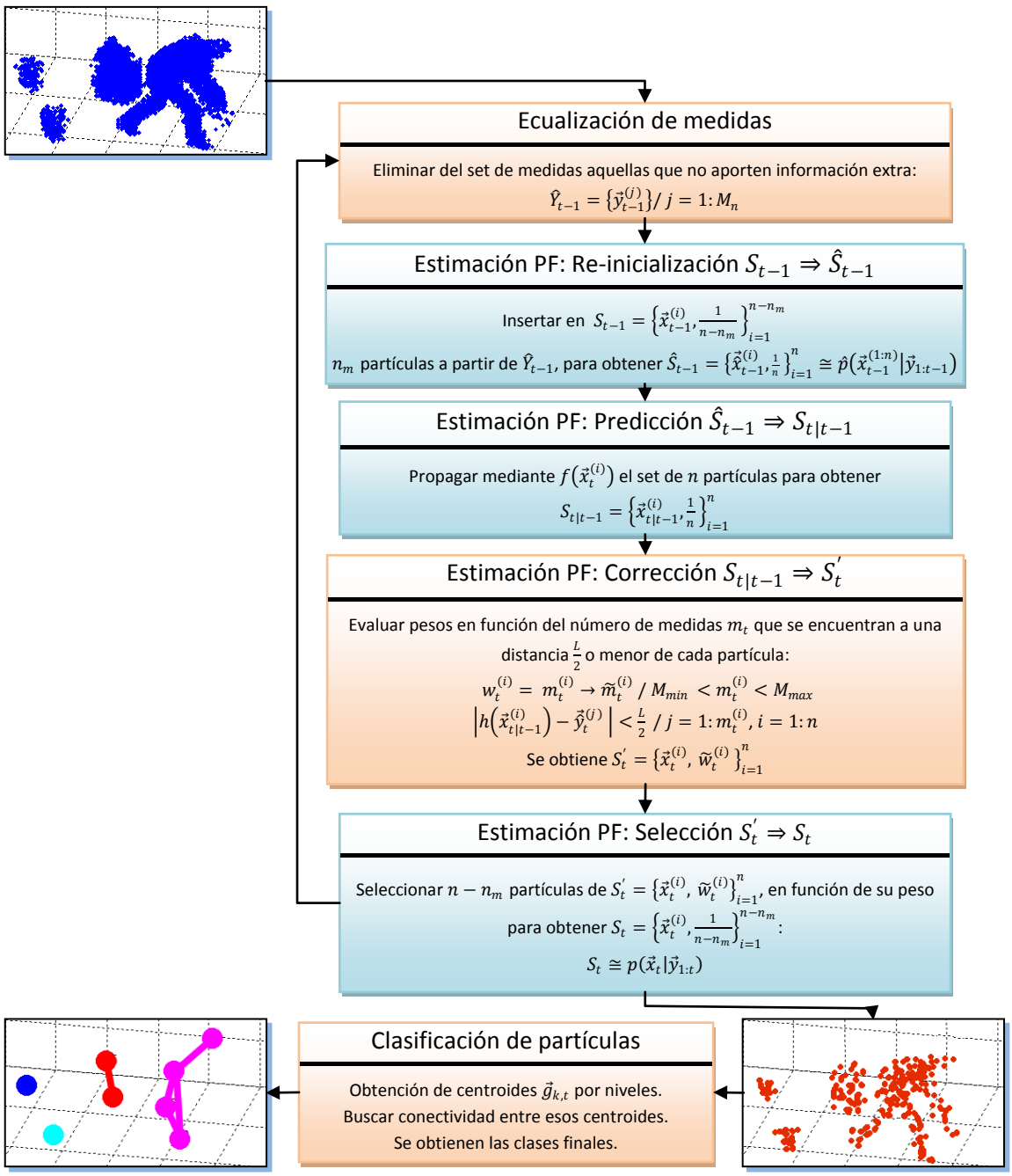


Figura 4.1. Diagrama funcional del 3DPF.

La estructura del algoritmo está basada en la del XPFCP ya visto en el capítulo anterior (ver figura 3.6). Sin embargo, en el 3DPF no se realiza clasificación de medidas, ya que en se requiere una clasificación más elaborada para que sea adaptable a cualquier modelo deformable. La ventaja del XPFCP es la gran robustez aportada por un proceso de clasificación de medidas que, utilizado en los pasos de re-inicialización y corrección, robustece la representación de todas las hipótesis de la creencia discreta. Para el 3DPF se propone prescindir de este clasificador, con objeto de evitar un proceso de seguimiento demasiado lento (ya que el número de elementos a clasificar es elevado) y poco flexible para ser utilizado en aplicaciones en tiempo real. Para mantener en todo caso el seguimiento multimodal es necesario por tanto actuar sobre el paso de corrección y el de clasificación, sustituyendo el efecto del clasificador eliminado del XPFCP.

Además, para obtener la salida final determinística del algoritmo se ha añadido a la salida del 3DPF un clasificador de partículas. Éste no funciona del mismo modo que el clasificador del XPFCP. Como se verá en detalle, más adelante, primero se realiza una clasificación por alturas de las partículas, y posteriormente se busca conectividad entre los grupos originados con dicha clasificación.

Por último y a modo de nota, es necesario mencionar que a lo largo de este capítulo se van a mostrar distintas figuras relacionadas con las distribuciones de partículas y clasificación de medidas. En todas se ha partido de una escena en la que aparece una escalera, tres papeleras y un robot, como se muestra en la Figura 4.2:

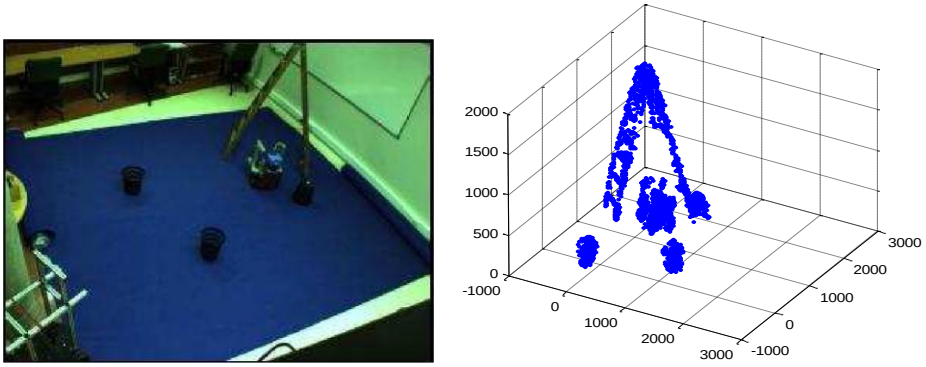


Figura 4.2. Izquierda: Imagen de la escena utilizada, obtenida a partir de una de las cámaras situada del espacio inteligente. Derecha: reconstrucción mediante Visual Hull de la misma.

4.1.1 Ecualización de medidas

En el paso de ecualización de medidas se reduce la densidad de las mismas en zonas en las que ya haya suficientes como para obtener una salida del algoritmo fiable. Esto tiene tres objetivos:

1. Por una parte, equilibrar la densidad de medidas a introducir en el paso de re-inicialización relacionadas con los distintos objetos existentes en la escena. Este objetivo es el mismo que tiene la clasificación de medidas en el XPFCP, que ha sido eliminada para el 3DPF.

2. Otro de los objetivos es el de igualar los valores de los pesos de aquellas partículas que tengan medidas cerca, para así tener representados los objetos pobremente sensados en la función de creencia. No obstante, como se verá más adelante, el algoritmo de cálculo de pesos también contribuye a esto.

3. Por último, el de reducir el tiempo de ejecución. Como también se verá, el tiempo de ejecución de algunas de las etapas del 3DPF depende del número de medidas. Si se reducen estas, se reduce el tiempo de ejecución global, siempre que, como es el caso, el tiempo invertido para ecualizar las medidas sea lo suficientemente bajo.

Se han implementado dos algoritmos distintos para llevar a cabo la ecualización de medidas. En el primero se determinan las zonas en las que hay medidas redundantes a través del análisis de los histogramas de medidas en distintas dimensiones. En el segundo, más complejo, se hace uso de un clasificador k-medias, de un análisis de los histogramas parecido al del primer

algoritmo, y de una discretización del volumen en celdas. Ambos algoritmos se explican a continuación.

4.1.1.1. Ecualización de medidas basada en histograma

El algoritmo de ecualización de medidas basada en histograma está formado por los pasos descritos a continuación:

1. Se obtienen los histogramas de medidas de las dimensiones *X* y *Z*, dividiendo éstos en n_{div} secciones.
2. Se determinan las zonas del histograma en las que se supera un límite predeterminado (*umbralHist*) de medidas.
3. A partir de estas zonas, se delimita la zona *XZ* en la que hay exceso de medidas, como puede verse en la Figura 4.3.
4. En las zonas con exceso de medidas se elimina una proporción *propRed* de las mismas.

La Figura 4.3 muestra el principio de funcionamiento.

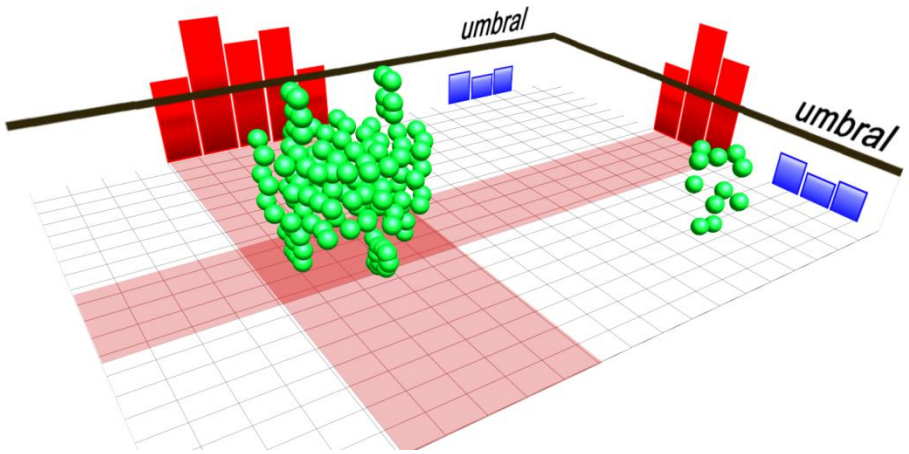


Figura 4.3. Diagrama de funcionamiento del paso de ecualización de medidas. Se han representado los histogramas de medidas en dos dimensiones. Se ha dibujado en rojo la zona en la que hay medidas redundantes (se supera el umbral establecido). Esa es la zona en la que se eliminan algunas medidas para evitar su reduncancia.

Los parámetros a determinar son n_{div}, *umbralHist* y *propRed*.

4.1.1.2. Ecualización de medidas basada en grupos

El algoritmo anterior arroja resultados satisfactorios en el experimento realizado (detallado en el capítulo 5). Incluso ha sido comprobado creando artificialmente situaciones en las que un objeto de la escena está sensado de una forma mucho más pobre que el resto. No obstante, para cubrir los casos más complejos y poco habituales se ha ideado otro algoritmo similar, pero realizando la ecualización en porciones del espacio tridimensional más específicas. Por ejemplo, es capaz de eliminar medidas redundantes asociadas al robot cuando este se encuentra debajo de la escalera, dejando intactas las medidas asociadas a ésta.

El diagrama funcional de este algoritmo se muestra en la Figura 4.4:

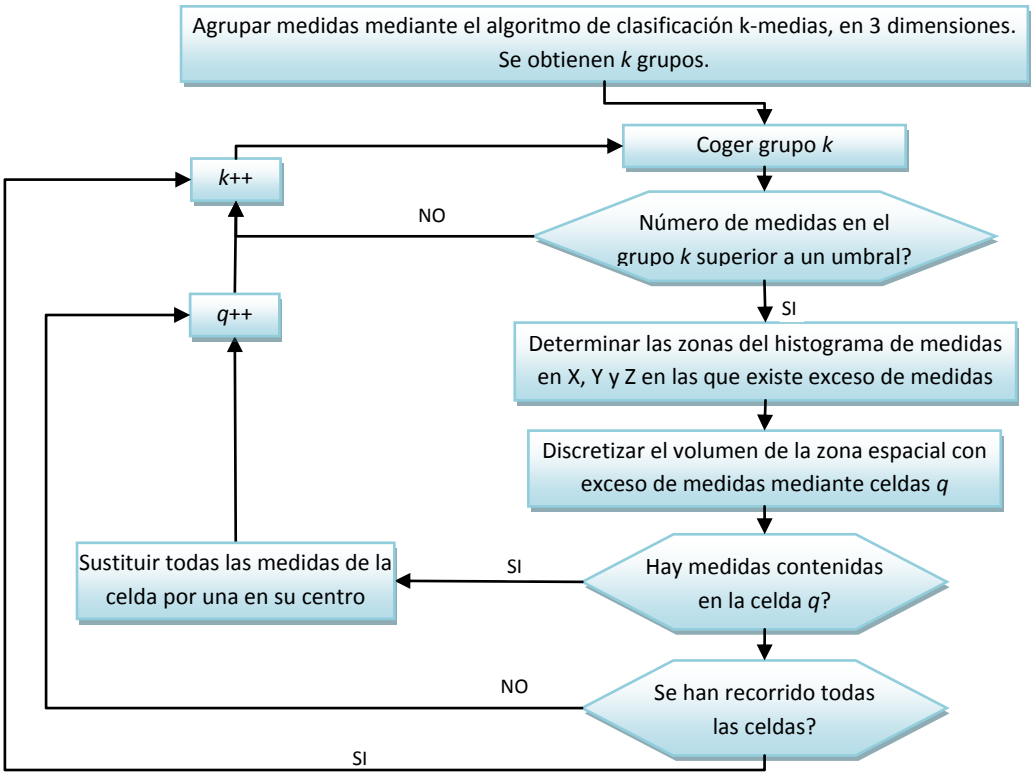


Figura 4.4. Diagrama funcional del algoritmo de ecualización de medidas por grupos.

Esta ecualización de medidas basada en grupos está formada por cuatro pasos:

1. Se agrupan las medidas utilizando el algoritmo secuencial k-medias con validación presentado en el capítulo 2. En este caso la clasificación se realiza sobre las tres dimensiones del espacio. El vector de medidas, como también se ha visto en el capítulo 2, es de la forma $[x_t\ y_t\ z_t]^T$. Al ser un clasificación por distancia euclidea *distG*, los grupos están representados por esferas en el espacio XYZ.
2. Se analiza el número de medidas asociadas a cada grupo $H_{1:k,t}$. Si superan un determinado umbral (al que se le ha llamado *umbralRed*) se va al siguiente paso.
3. Se analiza el histograma del grupo con exceso de medidas, de la misma forma que en el algoritmo de ecualización basado en histograma. En las zonas en las que se supere un determinado umbral (llamado *umbralGr)* se va al siguiente paso.
4. Se discretiza el volumen en celdas, en la zona con exceso de medidas determinada mediante los tres primeros pasos. Si en una celda existen medidas, se sustituyen todas por una única medida situada en el centro de la celda.

Así se consigue igualar la densidad de medidas entre las zonas bien sensadas y las pobremente sensadas.

Los parámetros a ajustar en este algoritmo son *distG*, *umbralRed* y *umbralGr*.

Frente a esta propuesta, se consideró eliminar los tres últimos pasos y utilizar una distancia de asociación más pequeña para el algoritmo k-medias, y así sustituir todas las medidas asociadas a un grupo por su centroide.

En la Figura 4.5 se aprecian los resultados de clasificación de medidas obtenidos al utilizar dos distancias de clasificación distintas para el algoritmo k-medias en tres dimensiones: una usual, parecida a la que se utilizaba en dos dimensiones en el XPFCP, y otra pequeña. El

problema de realizar la clasificación con una distancia pequeña es el tiempo de ejecución: se necesitaron 2,8 segundos frente a los 0,5 de una distancia normal, por lo que finalmente no se ha implementado.

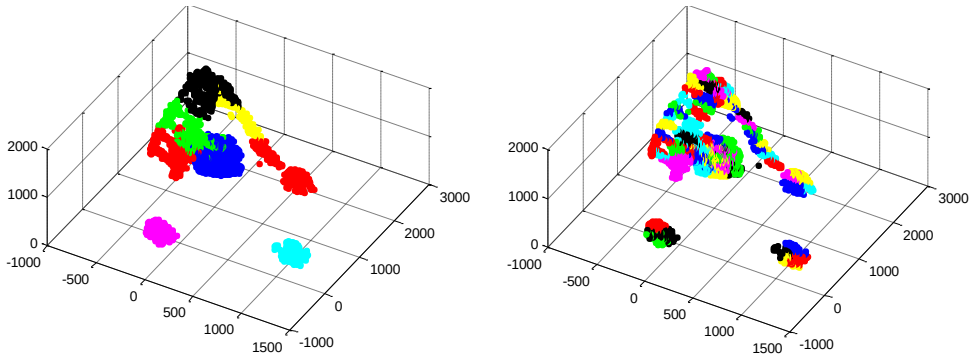


Figura 4.5. Resultados de clasificación de medidas obtenido para una distancia de clasificación normal (izquierda) y una pequeña (derecha).

Por lo tanto, se han visto dos métodos distintos de ecualización de medidas. El primero, basado en histograma, es más fácil de implementar y de poner a punto, ya que tiene menos parámetros involucrados en su funcionamiento. Como inconveniente, sólo funciona bien en situaciones en las que no hay un objeto encima del otro, ya que se analizan los histogramas en *X* y en *Z*, no en *Y* (la altura). Este inconveniente no aparece en la ecualización de medidas basada en grupos, la que por el contrario, necesita del ajuste de un mayor número de parámetros para su correcto funcionamiento.

4.1.2 Corrección

El diagrama de flujo del algoritmo de corrección puede verse en la Figura 4.6 y se explica en párrafos posteriores:

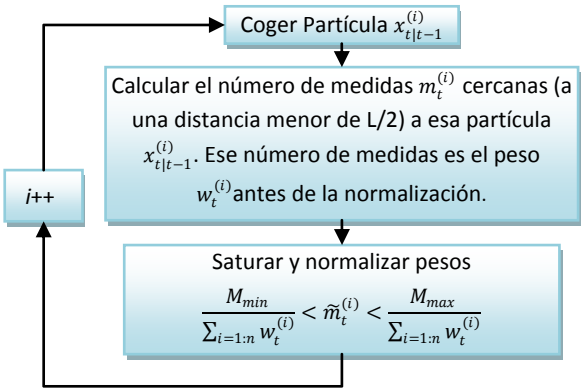


Figura 4.6. Diagrama de flujo del algoritmo empleado para el paso de corrección.

En el algoritmo propuesto el paso de corrección o ponderación de cada partícula se realiza asignando a cada partícula un peso $w_t^{(i)}$ proporcional al número $m_t^{(i)}$ de observaciones cercanas a las hipótesis de posición que ésta representa. En la propuesta implementada se entiende como observaciones cercanas a aquellas que caen dentro de un cubo de lado *L* con centro en la hipótesis de posición que representa la partícula correspondiente.

Para reducir el impacto de las distintas densidades de medidas asociadas a la naturaleza de cada objeto (los objetos más finos estarán más pobremente sensados, es decir, generarán una menor densidad de medidas), se realiza una saturación del conjunto de pesos al rango:

$$\frac{M_{min}}{\sum_{i=1:n} w_t^{(i)}} < \tilde{m}_t^{(i)} < \frac{M_{max}}{\sum_{i=1:n} w_t^{(i)}} \tag{4.1}$$

Donde $\tilde{m}_t^{(i)}$ es el número normalizado de medidas asociado a cada partícula $x_{t|t-1}^{(i)}$ y:

$$M_{max} = \alpha\, M_{\max\ cubo} \tag{4.2}$$

$$M_{min} = \beta\, M_{\max\ cubo} \tag{4.3}$$

Con $\alpha > \beta$, $\alpha, \beta \in [0,1]$, y:

$$M_{\max\ cubo} = \frac{L^3}{o^3} \tag{4.4}$$

Donde *o* es el tamaño del lado de cada celda en la rejilla de ocupación (ver Figura 2.2).

Esta saturación asegura que no se asigne a la partícula $x_{t|t-1}^{(i)}$ un número de medidas mayor que M_{max} ni menor de M_{min}. Se consigue así establecer un límite a partir del cual la densidad de medidas que rodea a cada partícula no modifica el peso de la partícula a la que se asocia. De esta forma, se disminuye la diferencia de pesos entre las partículas que representan a objetos pobremente sensados y las que representan a los mejor sensados, eliminando el problema de degeneración del set de partículas, ya comentado en el capítulo 2.

Realizar el pesado de esta forma equivale a dotar a las partículas de volumen, de modo que cuanto mayor es la distancia de búsqueda de medidas en torno a partículas (*L/2*), mayor es este volumen. En la Figura 4.7 puede verse un ejemplo:

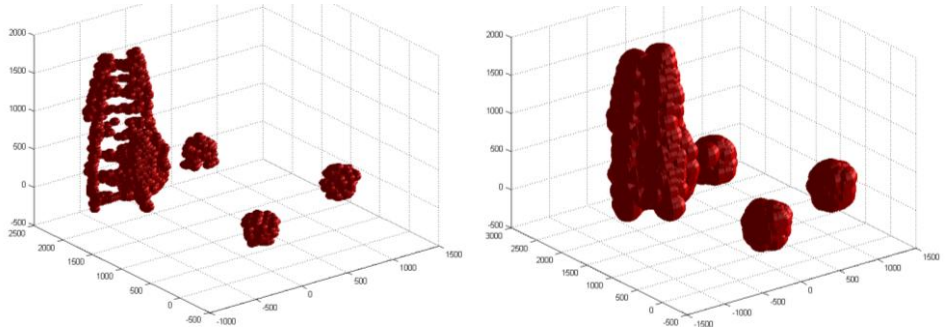


Figura 4.7. Partículas representadas mediante esferas de radio L/2. Izquierda: L=100. Derecha: L=300

Los parámetros a ajustar en este paso son α, β y *L*.

4.1.3 Selección

Para realizar el paso de selección se han explorado distintos algoritmos existentes (los dos primeros descritos en [Marrón08]): multinomial, residual y sistemático.

El muestreo multinomial es el primer algoritmo de resampling diseñado para el PF ([Marrón08]). El algoritmo consiste en muestrear uniformemente *n* veces la creencia discreta a la salida del paso de corrección. Las partículas con mayores pesos se repiten un número mayor

de veces que aquellas que cuenten con un peso menor. En [Gordon93] puede encontrarse una implementación de este algoritmo.

El muestreo residual, presentado por [Liu98] constituye una alternativa al multinomial. El tiempo de ejecución necesario es menor que en el caso del muestreo multinomial ([Marrón08]). En este algoritmo, primero se obtiene el número *n* de repeticiones (o hijos) asociadas a cada partícula, y luego se reparte el número de muestras que falte por asignar de forma aleatoria entre todo el set de partículas mediante el procedimiento de selección multinomial.

El muestreo sistemático está basado en el conocido como muestreo estratificado ([Douc05]), sobre el que se añade un proceso determinístico, también explicado en [Douc05]. Se describe al algoritmo como uno computacionalmente simple que ofrece buenos resultados empíricos.

Las distribuciones de partículas obtenidas con los distintos algoritmos de resampling mencionados se muestran en la Figura 4.8. Las partículas se representan como esferas cuyo radio es de la longitud *L/2*, que como se ha visto en la Figura 4.7, es el que se utiliza para buscar medidas en torno a cada partícula.

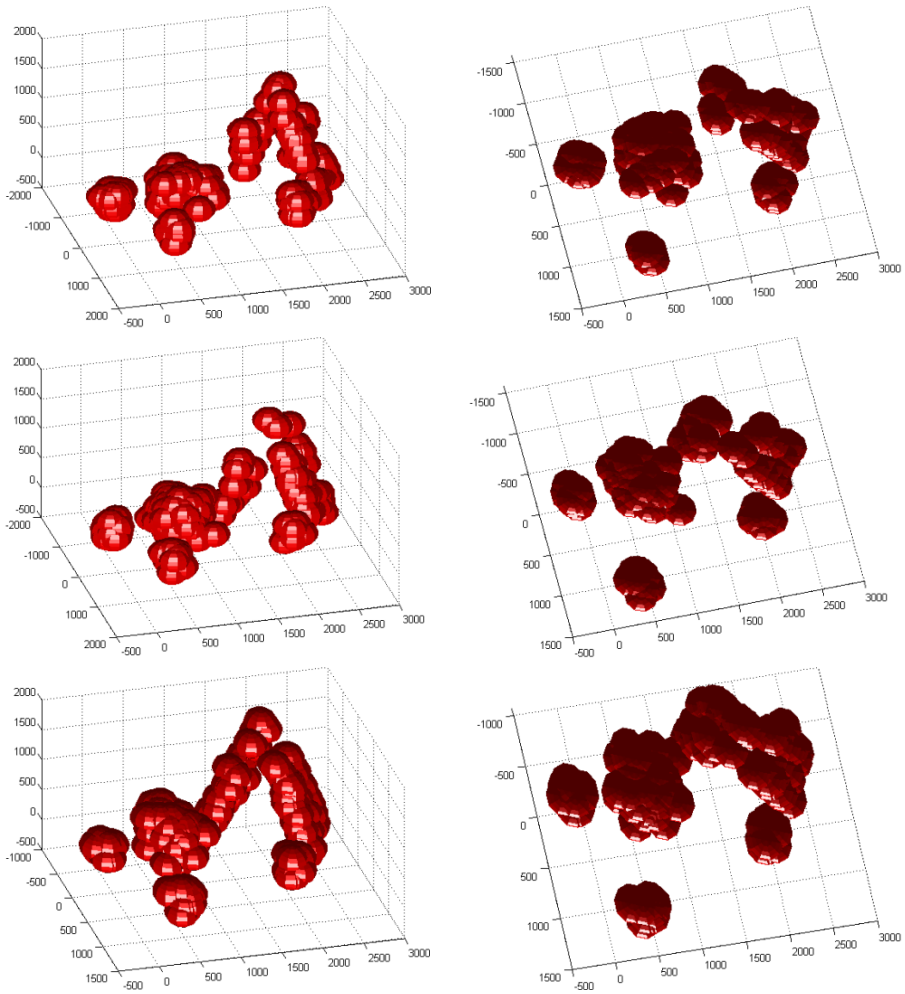


Figura 4.8. Distribución de partículas utilizando tres algoritmos de selección distintos. Arriba: residual, dos perspectivas distintas. Medio: sistemático, dos perspectivas distintas. Abajo: multinomial, dos perspectivas distintas.

Salvando las diferencias inherentes a los resultados no determinísticos propios de un sistema de seguimiento probabilístico, no se aprecian diferencias ni aparentemente visuales, ni en cuanto

a número eficaz de partículas n_{eff} (descrito en el punto 2.2.2), estando en todos los casos en torno al 70%.

Debido a ello se ha optado por utilizar el algoritmo ya probado en el XPFCP, el residual. En cualquier caso, en el proyecto se incluye la implementación de los tres aquí mostrados (ver manual de usuario) por si se quiere emplear otro en posteriores implementaciones del sistema.

4.1.4 Clasificación de partículas

La salida del PF es una función de creencia discretizada $S_t = \left\{\vec{x}_t^{(i)}, \frac{1}{n-n_m}\right\}_{i=1}^{n-n_m} \cong p\left(\vec{x}_t^{(1:n-n_m)} \middle| \vec{y}_{1:t}\right)$ mediante partículas de las hipótesis de posición de los múltiples objetos seguidos. La salida del algoritmo debe ser, sin embargo, una estimación determinística de la posición de cada objeto representado como un modo de esta creencia. Es por lo tanto necesario añadir un proceso de agrupación de partículas a la salida del 3DPF propuesto, como también se incluye en el XPFCP (ver Figura 3.6).

El proceso de clasificación propuesto en esta tesis se realiza en dos pasos.

1. *Primer paso*: Se divide el volumen del entorno observado con *P* planos equidistantes entre sí y paralelos al del suelo, con una distancia *h* entre dos planos contiguos. Por lo tanto, si el tamaño del espacio es conocido, el parámetro *P* fija la distancia *h*. Las partículas $x_t^{(i)}$ (que a la salida del paso de corrección representan una posición en la que existen observaciones), contenidas entre dos planos de altura y_P contigua son proyectadas en un plano intermedio. La altura $\mathrm{y}_{\Omega_P,proy}$ de este plano intermedio es:

$$\mathrm{y}_{\Omega_P,proy} = \frac{\mathrm{y}_{P+1}-\mathrm{y}_P}{2} \tag{4.5}$$

   De este modo se obtienen *P* planos Ω_P paralelos al del suelo sobre los que se encuentra la proyección de las partículas de un rango de altura $\mathrm{y}_{\Omega_P,proy}$ determinado.

   A continuación se aplica un algoritmo de clasificación k-medias extendido con soporte para número de clases variables con distancia de clasificación *distM* (ver 3.1.2). En este caso no se utiliza el proceso de predicción de centroides, ya que no es necesario depender de los centroides del instante de tiempo anterior. Dado que este algoritmo de clasificación se aplica sobre las proyecciones de las partículas en alturas fijas $\mathrm{y}_{\Omega_P,proy}$, sólo se tiene en cuenta la coordenada *X* y la *Z* de las mismas. Es decir, se hace una clasificación en dos dimensiones. De este modo se obtiene conjunto de *k* centroides $\vec{g}_t^{(l)} = \left[x_t^{(l)}\ y_t^{(l)}\ z_t^{(l)}\right] / l = 1{:}\,k$ en el espacio *XYZ* en alturas $\mathrm{y}_{\Omega_P,proy}$ predeterminadas. Estos puntos se corresponden con las posiciones estimadas del centroide de las secciones de los objetos en el rango de altura $[\mathrm{y}_P, \mathrm{y}_{P+1}]$.

2. *Segundo paso*: Consiste en aplicar un filtro de conectividad para unir los centroides anteriormente obtenidos. El algoritmo de conectividad busca, para cada punto $\vec{g}_t^{(l)}$, otros puntos ($\vec{g}_t^{(1)} \ldots \vec{g}_t^{(k)}$) a una distancia menor que *distC*, que de este modo se definen como

conexos a $\vec{g}_t^{(l)}$. El proceso se repite para cada punto, desechando los ya analizados, y originando n_e estructuras $M_q = \{\vec{\mu}_{s,q} / \, s = 1{:}\,\mu_q\}$. Cada elemento de una estructura $\vec{\mu}_{s,q}$ tiene un identificador (ID) asociado, y es conexo a cada elemento de la misma estructura, y no conexo a cualquier elemento de otra. El esquema de funcionamiento de este algoritmo de conectividad se muestra en la Figura 4.9.

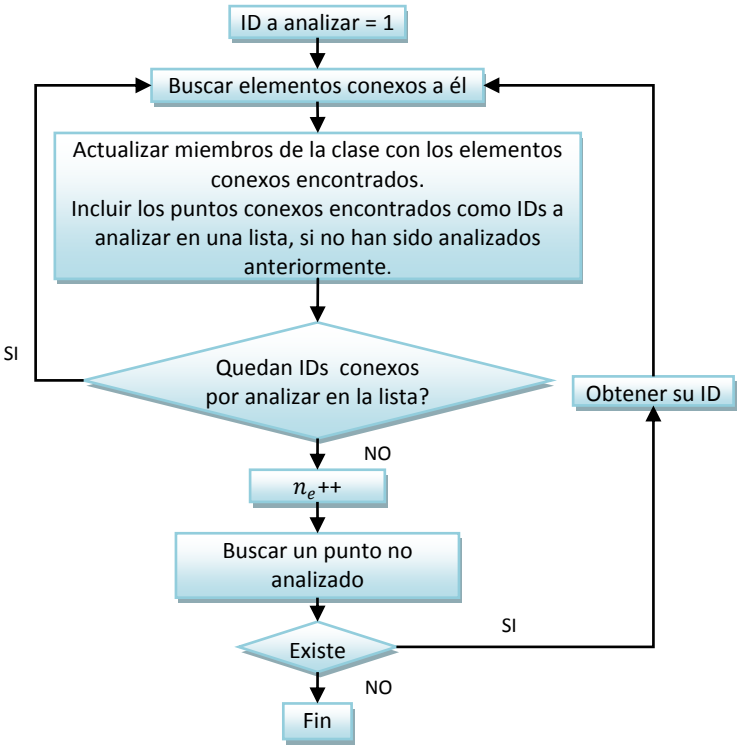


Figura 4.9. Diagrama funcional del segundo paso del algoritmo de conectividad.

Es necesario tener en cuenta que la coordenada *Y* en la que pueden estar presentes los centroides se ha discretizado en el paso primero. Sin embargo, el clasificador proporciona unas coordenadas *XZ* continuas.

Además, para flexibilizar la forma deformable de los objetos seguidos en el entorno, es necesaria la inclusión de dos parámetros, ξ y δ, que escalen las coordenadas tridimensionales para calcular la distancia de conectividad d_{con} (debe ser menor que *distC* para que los elementos sean conexos) entre dos puntos $\vec{g}_t^{(u)}$ y $\vec{g}_t^{(v)}$ según la siguiente expresión:

$$d_{con}(u,v)_t = \sqrt{\xi\left(x_t^{(u)} - x_t^{(v)}\right)^2 + \xi\left(z_t^{(u)} - z_t^{(v)}\right)^2 + \delta\left(y_t^{(u)} - y_t^{(v)}\right)^2} \qquad (4.6)$$

Durante el proceso de conectividad se propone asignar inequívocamente una clase de salida a cada elemento presente en el entorno, considerando que cada estructura generada por el algoritmo de conectividad es una clase de salida.

En la Figura 4.10 se ilustra el funcionamiento del clasificador de partículas descrito en este apartado.

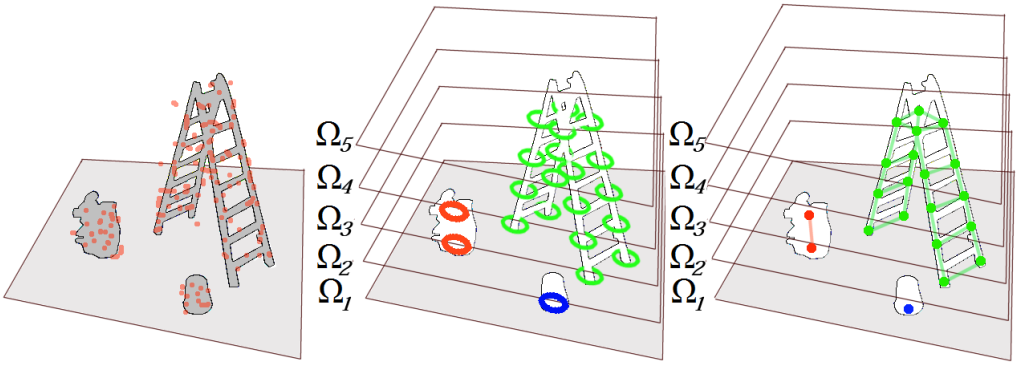


Figura 4.10. Clasificador de partículas propuesto en la tesis para el algoritmo 3DPF. Izquierda: partículas (en rojo) distribuidas sobre los objetos de la escena a la salida del paso de corrección del 3DPF propuesto (ver apartado 4.1.2). Centro: Conjunto de clases obtenidas al ejecutar el k-medias extendido bidimensional sobre la proyección de las partículas. Derecha: Resultado final al aplicar el algoritmo de conectividad propuesto sobre los centroides. Cada estructura de centroides conectada representa a un objeto individual y se muestra en un color diferente.

Los parámetros a ajustar en la clasificación de partículas son *distM*, *distC*, ξ y δ.

Antes de llegar a la solución final del paso de clasificación de salida expuesto en los párrafos anteriores, se exploraron distintas alternativas. Una de ellas consistía en realizar el mismo primer paso y sustituir el segundo por otra clasificación, ahora tridimensional, de los centroides obtenidos, en lugar de utilizar el algoritmo de conectividad. Algunos de los resultados arrojados con esta otra propuesta se muestran en Figura 4.11:

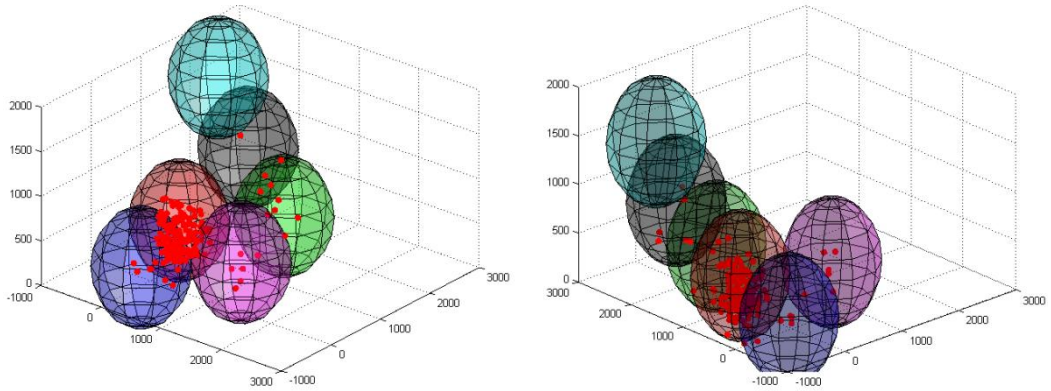


Figura 4.11. Resultados obtenidos aplicando un k-medias tridimensional a los centroides, utilizando el mismo ejemplo de la Figura 4.2. Se representan como esferas los distintos grupos obtenidos (objetos identificados como distintos por este clasificador), y en rojo las proyección de las partículas en el espacio *XYZ*.

Como puede verse en la Figura 4.11, utilizando este otro método no se tiene en cuenta la continuidad de las partículas para construir las clases finales, ni el carácter deformable y muy diferente de los objetos que aparecen en la escena. Por ejemplo, se forman dos clases para la escalera, una por cada pata (la esfera cyan y la negra). Además, no es posible reconstruir la estructura morfológica de los elementos de la escena. Por todo ello, este otro método se ha descartado como paso de clasificación.

4.2 Conclusiones

En este capítulo se ha explicado el sistema de seguimiento tridimensional propuesto, el cual está basado en el XPFCP ya visto en el capítulo 3. De todo ello se extraen las siguientes conclusiones:

- Debido a ciertas características propias del seguimiento tridimensional que no aparecen en el seguimiento bidimensional, como el hecho de que las entidades a seguir suelen ser deformables y flexibles, es necesario modificar varios pasos del XPFCP.
- Ya no se utiliza la clasificación de medidas (también conocida como clasificación de entrada), que como se ha visto en el capítulo 3, es la propuesta realizada por [Marrón08] para hacer robusto al XPF.
- Por lo tanto, al quitar este importante paso, es necesario actuar sobre los pasos de corrección y clasificación de partículas para obtener un efecto equivalente, así como añadir una etapa de ecualización de medidas, todo ello para mantener una densidad de partículas constante en torno a objetos con diferentes calidades de sensado.

CAPÍTULO 5

RESULTADOS

En este capítulo se muestran los resultados de distintos experimentos, cuya finalidad es tanto analizar como optimizar los algoritmos presentados. En el apartado 5.1 se muestra el análisis del sistema de seguimiento bidimensional (como ya se ha visto anteriormente, compuesto por un XPFCP y un proceso de identificación de un robot), mientras que en el 5.2 se hace lo propio para el seguimiento tridimensional (3DPF).

5.1 Resultados XPFCP y proceso de identificación

En esta sección se analiza el funcionamiento del sistema implementado para realizar el seguimiento bidimensional. Ya que se trata de procesos separados, se ha dividido el análisis en XPFCP y proceso de identificación.

A lo largo de esta parte se hace referencia a tres pruebas o experimentos distintos: uno sencillo, otro de dificultad media y uno complejo:

1. En el sencillo aparecen en todo momento dos entidades en la escena: una persona y un robot. Ambos se mueven únicamente en línea recta, cambiando de sentido, durante aproximadamente 45 segundos.
2. En el de dificultad media no aumenta el grado de complejidad en cuanto al número de clases, ya que también aparecen dos en todo momento: una persona y un robot. Sin embargo, ahora la persona trata en todo momento de imitar los movimientos realizados por el robot. Dura unos 85 segundos.
3. Por último, en el experimento complejo, aparecen entre cero y cinco entidades en la escena, siendo una de ellas un robot, otras dos personas, y el resto objetos inanimados como

papeleras. Aparecen distintas situaciones que ponen a prueba tanto el funcionamiento del XPFCP como el del proceso de identificación, durante aproximadamente 2 minutos.

5.1.1. Resultados XPFCP

A continuación se exponen los resultados relativos al XPFCP. Estos resultados tienen como fin analizar la fiabilidad del sistema frente a variaciones de parámetros, comprobar el correcto funcionamiento del XPFCP, obtener la distribución de los tiempos de ejecución.

En la Figura 5.1 se muestran resultados obtenidos desde la aplicación en tiempo real:

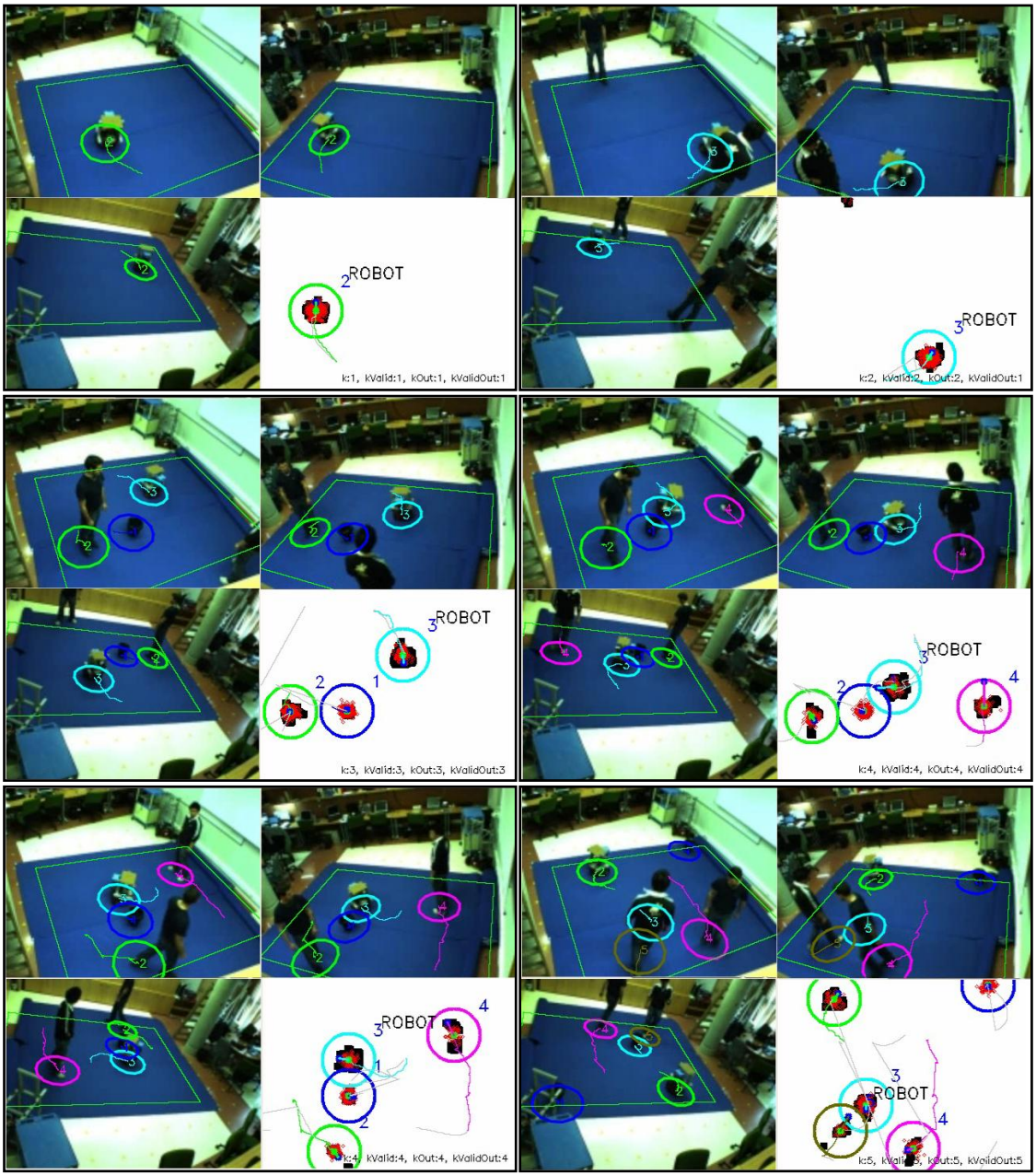


Figura 5.1. Resultados obtenidos con la aplicación en tiempo real, en 6 iteraciones distintas. Se muestran las tres perspectivas que proporcionan las cámaras, y una representación del funcionamiento del XPFCP y proceso de identificación. En esta última (llamada GRID) se representa cada clase mediante una circunferencia con un color asociado. La trayectoria de la misma estimada mediante el XPFCP aparece en el mismo color. La reconstruida a partir de la odometría se representa en gris. En la clase que se considera robot se añade la etiqueta "ROBOT".

5.1.1.1 Variación de parámetros

En esta sección se analizan los efectos de modificar dos parámetros importantes en el XPFCP: el número de partículas total n y el porcentaje de partículas insertadas en el paso de re-inicialización γ. Se repite el proceso para una prueba sencilla y una compleja. También para errores superiores a medio segundo y errores superiores a un segundo.

a) Variación de γ – Errores superiores a medio segundo

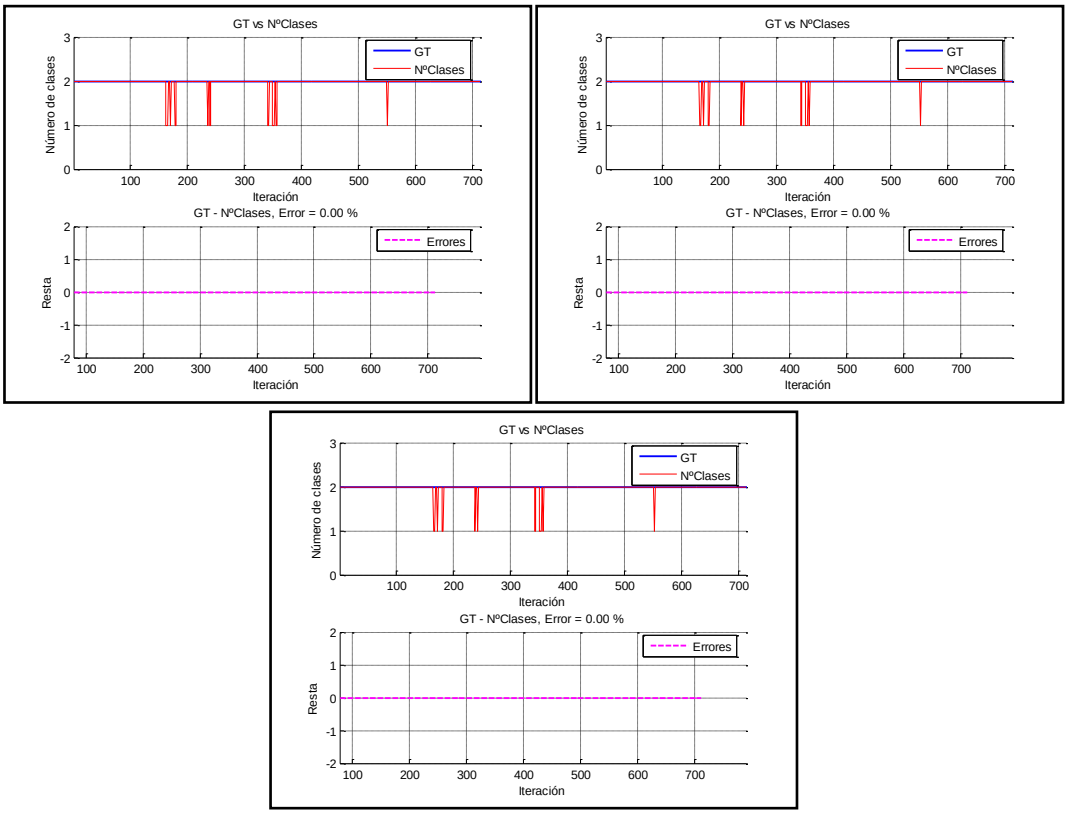


Figura 5.2. Prueba sencilla. Arriba izquierda: resultados para γ=20. Arriba derecha: resultados para γ=40. Abajo: resultados para γ=60.

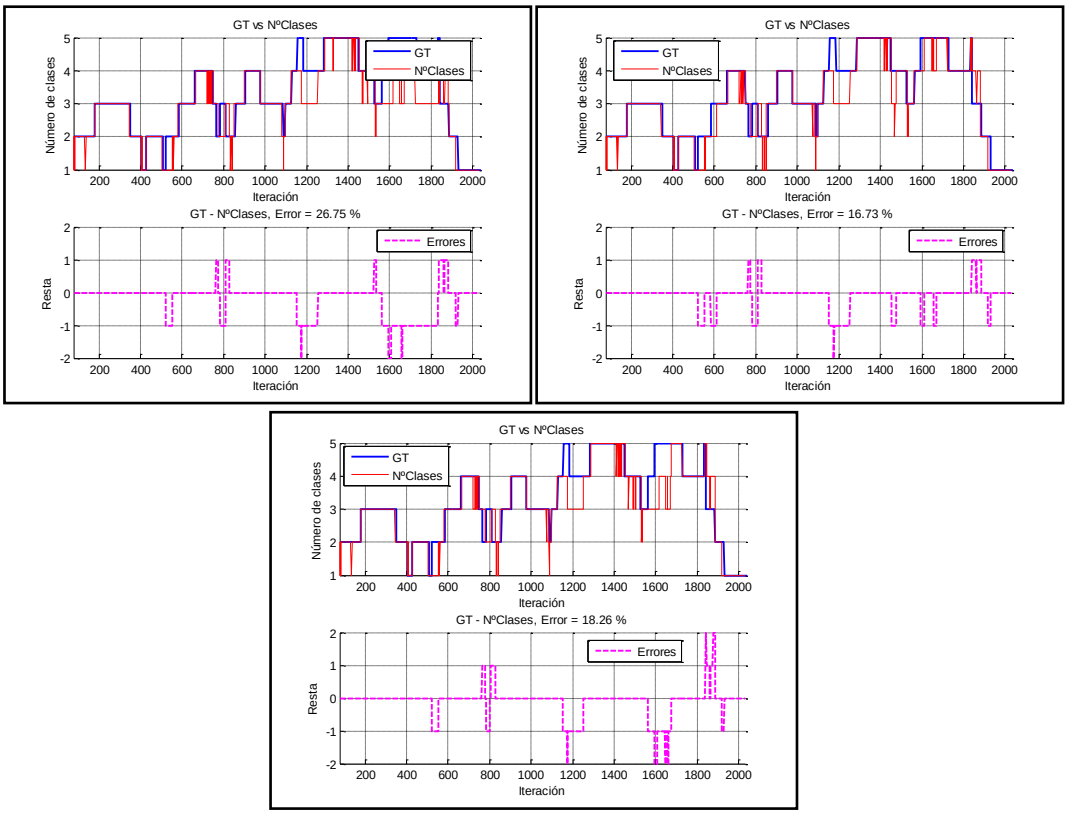


Figura 5.3. Prueba compleja. Arriba izquierda: resultados para γ=20. Arriba derecha: resultados para γ=40. Abajo: resultados para γ=60.

b) Variación de γ – Errores superiores a un segundo

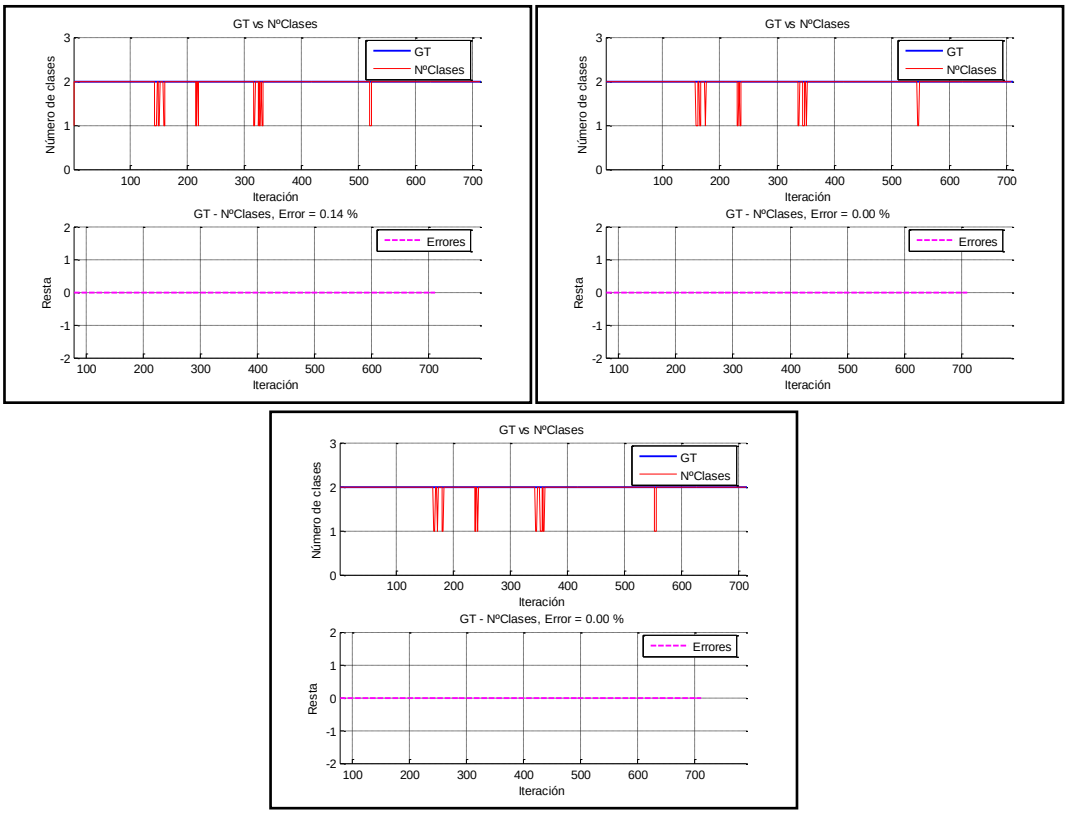


Figura 5.4. Prueba sencilla. Arriba izquierda: resultados para γ=20. Arriba derecha: resultados para γ=40. Abajo: resultados para γ=60.

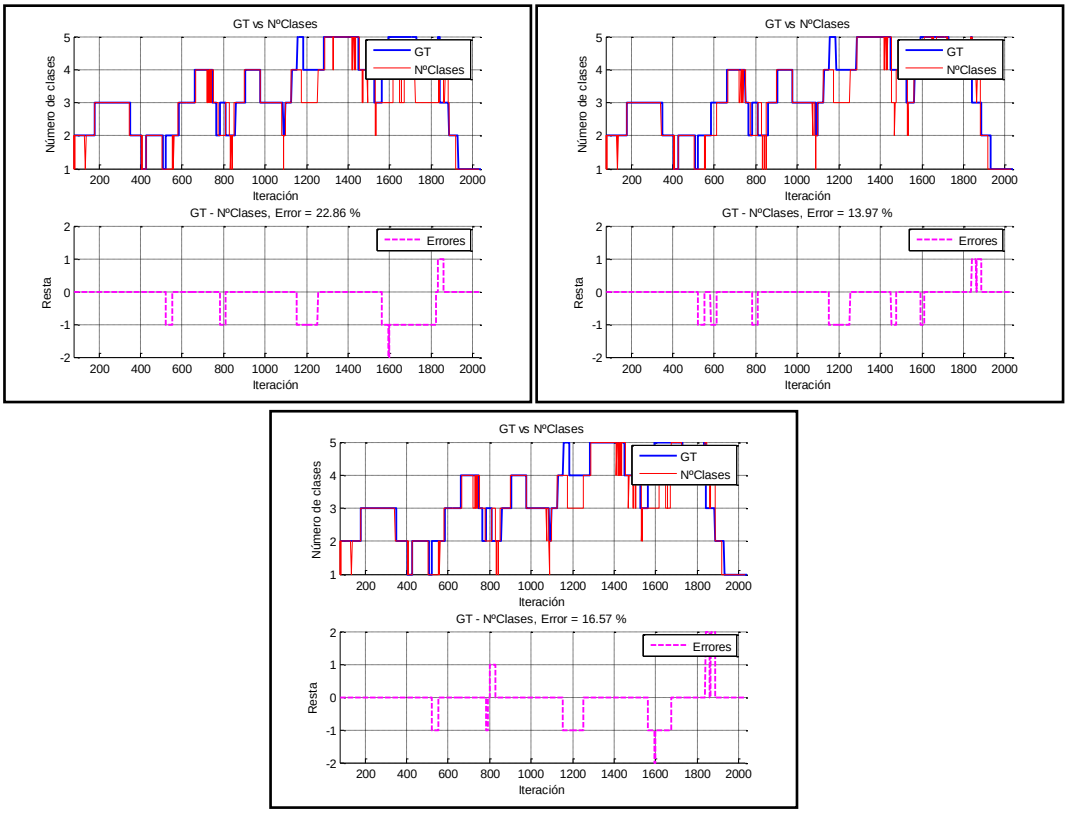


Figura 5.5. Prueba compleja. Arriba izquierda: resultados para γ=20. Arriba derecha: resultados para γ=40. Abajo: resultados para γ=60.

c) Variación de n – Errores superiores a medio segundo

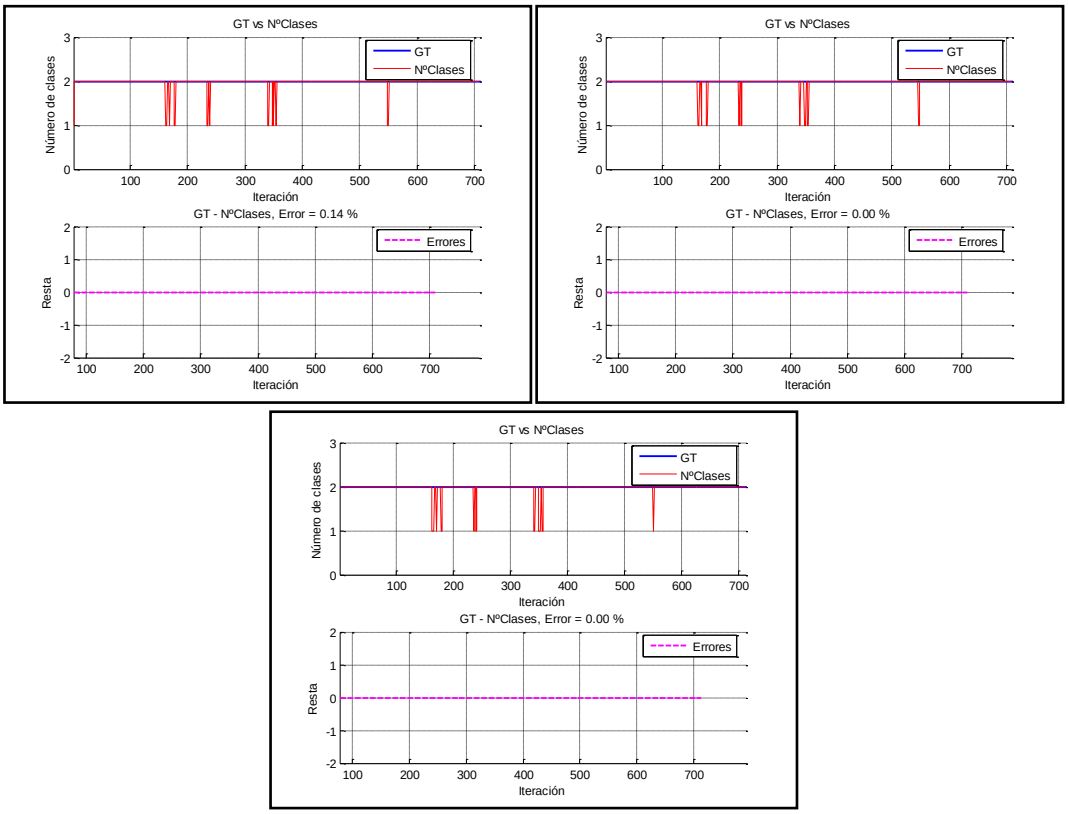


Figura 5.6. Prueba simple. Arriba izquierda: resultados para n=300. Arriba derecha: resultados para n=600. Abajo: resultados para n=900.

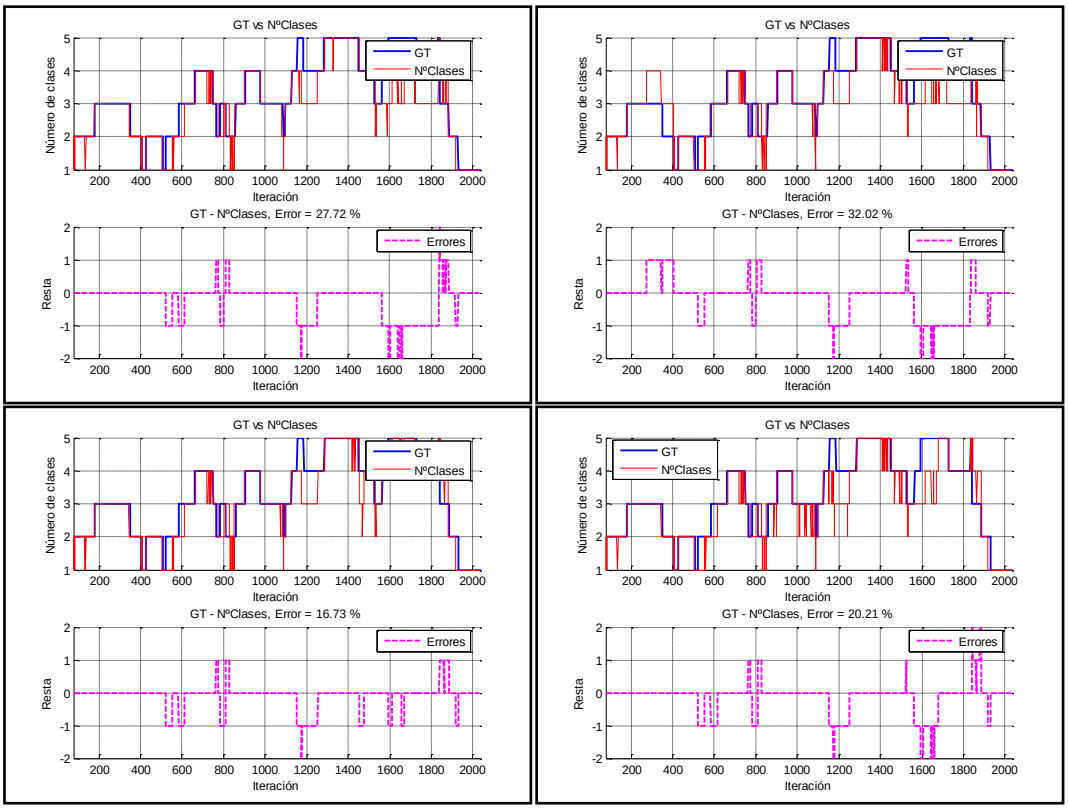


Figura 5.7. Prueba compleja. Arriba izquierda: resultados para n=300. Arriba derecha: resultados para n=600. Abajo izquierda: resultados para n=900. Abajo derecha: resultados para n=1200.

d) Variación de n – Errores superiores a un segundo

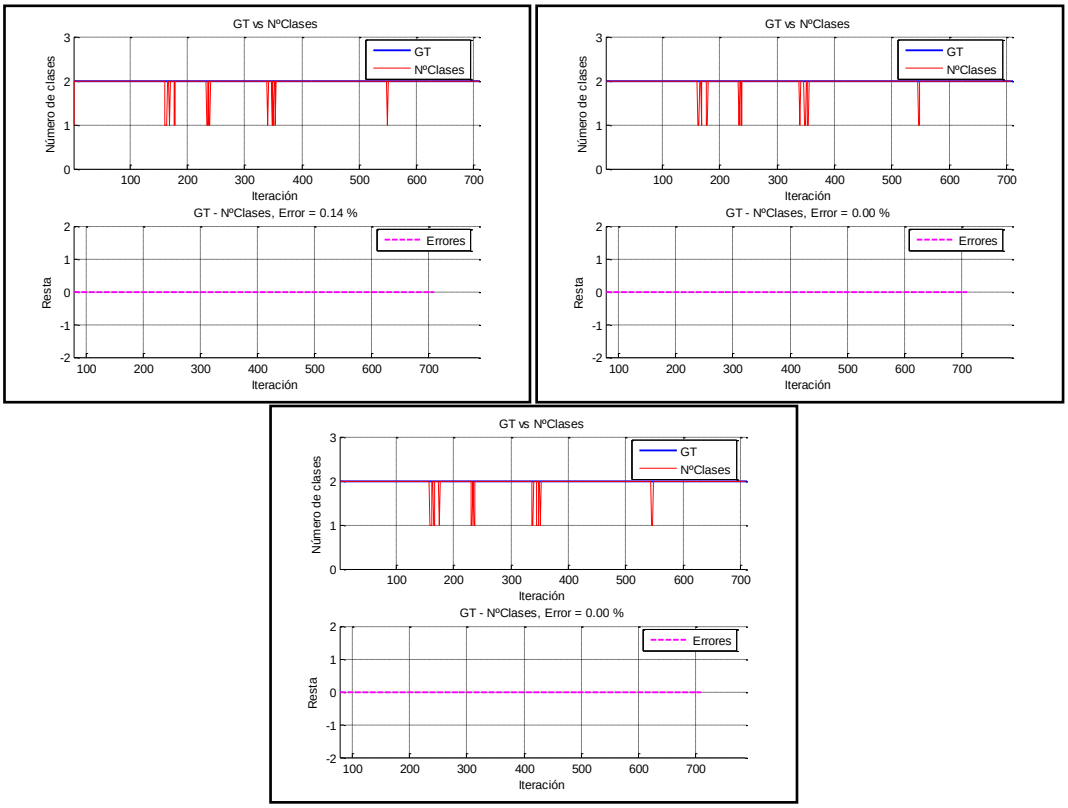


Figura 5.8. Prueba simple. Arriba izquierda: resultados para n=300. Arriba derecha: resultados para n=600. Abajo: resultados para n=900.

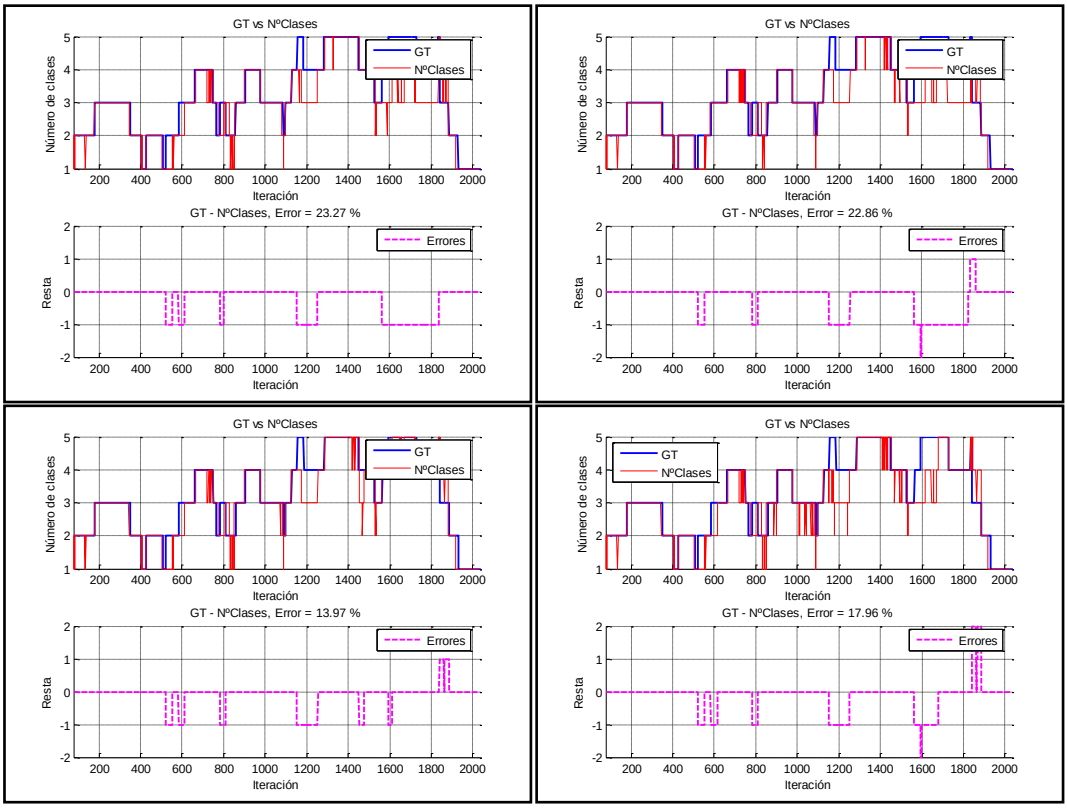


Figura 5.9. Prueba compleja. Arriba izquierda: resultados para n=300. Arriba derecha: resultados para n=600. Abajo izquierda: resultados para n=900. Abajo derecha: resultados para n=1200.

Analizando los resultados mostrados en las Figura 5.2, Figura 5.3, Figura 5.4, Figura 5.5, Figura 5.6, Figura 5.7, Figura 5.8 y Figura 5.9, se puede concluir que:

a) Resultados relativos al test sencillo

- En el test sencillo no aparecen errores sea cual sea el valor de γ utilizado. Las clases suelen estar bien separadas y sensadas a lo largo de todo el experimento por lo que como es de esperar, no se producen uniones ni grupos no generados.
- También aparece un error nulo variando n para este experimento, ya que al haber sólo dos hipótesis a seguir, el número mínimo de partículas parametrizado (n = 300) es suficiente.

b) Resultados relativos al test complejo

- En el experimento complejo, se observan los mejores resultados cuando se utiliza $\gamma = 40\%$. Números más bajos hacen que las nuevas hipótesis no se introduzcan de forma satisfactoria y porcentajes más altos quitan rigor al seguidor.
- Respecto al número de partículas, la tasa más baja de error se obtiene al utilizar n = 900.

c) Otras conclusiones

- Por otra parte, recordando que las partículas son una representación discreta de una función densidad de probabilidad, tiene sentido el hecho que al sobremuestrearla (es decir, utilizar más partículas de lo necesario) no mejore la calidad del filtro, pero se incremente el tiempo de procesamiento. De hecho puede verse que utilizando 1200 partículas las tasas de error no bajan.

- Todo esto es aplicable tanto considerando los errores superiores a medio segundo como los superiores a un segundo.

5.1.1.2 Análisis de la fiabilidad

A partir de los resultados obtenidos en el set de pruebas realizadas anteriormente, se ha hecho una recopilación para dar lugar a las siguientes tablas, a modo de resumen:

Test sencillo (v1)	
Características del test	2 objetos durante toda la duración.
Parámetros elegidos	*n* 900, γ 40%
Fiabilidad	0 % de errores
Naturaleza de los errores	0% no generado, 0% duplicado, 0% desplazado
Sensibilidad a parámetros	n - 50% variación -> std: 0,080829038
	γ- 50% variacion -> std: 0,080829038
n_{eff} (%)	63,54
t_{exe}	18 FPS
Test complejo (v6)	
Características del test	Entre 1 y 5 objetos.
Parámetros elegidos	*n* 900, γ 40%
Fiabilidad	13,97 % de errores
Naturaleza de los errores	84,62% no generado, 15,38% duplicado, 0% desplazado
Sensibilidad a parámetros	n - 50% variación -> std: 4,452755701
	γ- 50% variacion -> std: 4,570853312
n_{eff} (%)	61,32
t_{exe}	18 FPS

Tabla 5.1. Fiabilidad del XPFCP considerando errores superiores a un segundo.

Test sencillo (v1)	
Características del test	2 objetos durante toda la duración.
Parámetros elegidos	*n* 900, γ 40%
Fiabilidad	0 % de errores
Naturaleza de los errores	0% no generado, 0% duplicado, 0% desplazado
Sensibilidad a parámetros	n - 50% variación -> std: 0,080829038
	γ - 50% variacion -> std: 0
n_{eff} (%)	63,54
t_{exe}	18 FPS
Test complejo (v6)	
Características del test	Entre 1 y 5 objetos.
Parámetros elegidos	*n* 900, γ 40%
Fiabilidad	13,97 % de errores
Naturaleza de los errores	78.29% no generado, 21.71% duplicado, 0% desplazado
Sensibilidad a parámetros	n - 50% variación -> std: 8,014264366
	γ - 50% variacion -> std: 5,397860687
n_{eff} (%)	61,32
t_{exe}	18 FPS

Tabla 5.2. Fiabilidad del XPFCP considerando errores superiores a medio segundo.

	Análisis parámetros			
n (con γ=40)	300	600	***900***	std
% error	0,14	0	0	0,080829
γ (con *n*=900)	20	***40***	60	std
% error	0,14	0	0	0,080829

Tabla 5.3. Recopilación de tasas de errores en el test sencillo contando errores superiores a un segundo.

	Análisis parámetros				
n (con γ=40)	300	600	***900***	1200	std
% error	23,27	22,86	13,97	17,96	4,452756
γ (con *n*=900)	20	***40***	60	std	
% error	22,86	13,97	16,57	4,570853	

Tabla 5.4. Recopilación de tasas de errores en el test complejo contando errores superiores a un segundo

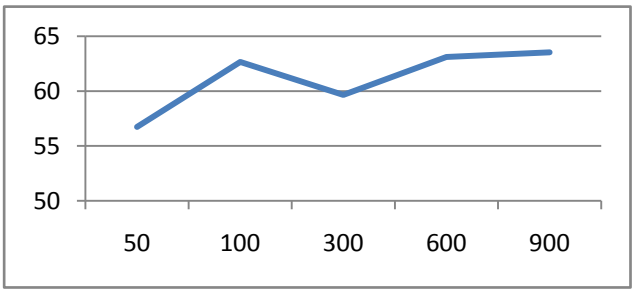
	Análisis parámetros			
n (con γ=40)	300	600	***900***	std
% error	0,14	0	0	0,080829
γ (con *n*=900)	20	***40***	60	std
% error	0	0	0	0

Tabla 5.5. Recopilación de tasas de errores en el test sencillo contando errores superiores a medio segundo.

	Análisis parámetros				
n (con γ=40)	300	600	***900***	1200	std
% error	27,72	32,02	16,73	20,21	8,014264
γ (con *n*=900)	20	***40***	60	std	
% error	26,75	16,73	18,26	5,397861	

Tabla 5.6. Recopilación de tasas de errores en el test complejo contando errores superiores a medio segundo

Experimento 1 (sencillo)	
γ constante (40%)	
n	n_{eff} (%)
50	56,73
100	62,66
300	59,66
600	63,11
900	63,54
n constante (900)	
γ	n_{eff} (%)
5	64,48
10	63,86
20	64,31
40	63,54
60	62,54
80	60,46



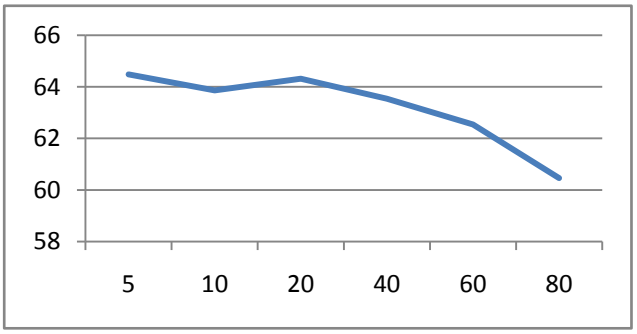
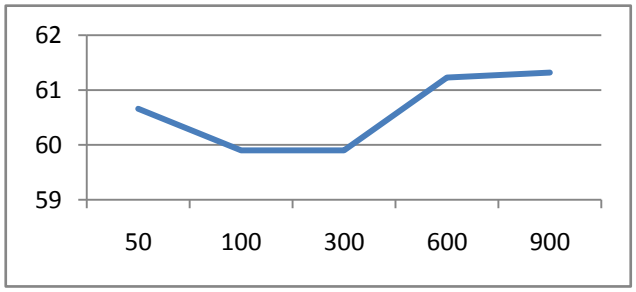


Tabla 5.7. Recopilación del número eficaz de partículas en el experimento sencillo.

Experimento 6 (complejo)	
γ constante (40%)	
n	n_{eff} (%)
50	60,66
100	59,9
300	59,9
600	61,23
900	61,32
n constante (900)	
γ	n_{eff} (%)
5	64,1
10	64,07
20	63,6
40	61,32
60	59,65
80	60,46



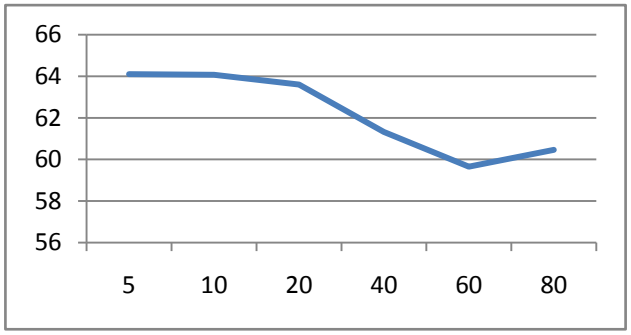


Tabla 5.8. Recopilación del número eficaz de partículas en el experimento complejo.

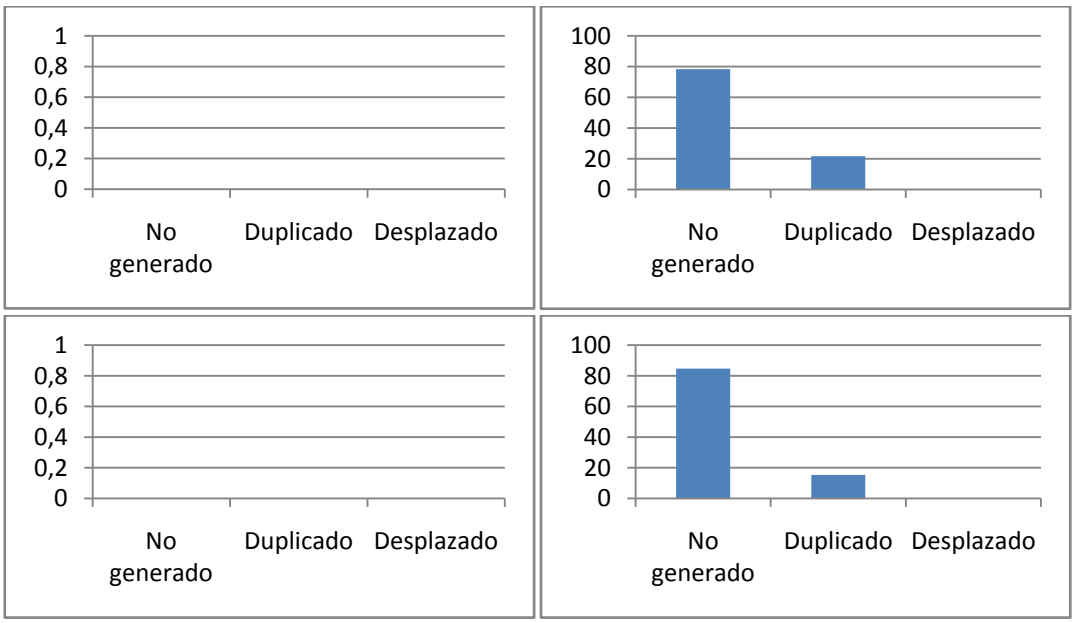


Figura 5.10. Tipos de errores. Arriba: Superiores a medio segundo. Abajo: superiores a un segundo. Izquierda: test sencillo. Derecha: test complejo.

- Como puede verse en la Tabla 5.3 y Tabla 5.5, en el experimento sencillo la tasa de error es prácticamente 0% (el pequeño error que aparece en dos de los test puede considerarse como retardo del estimador).
- En el complejo (Tabla 5.4 y Tabla 5.6), la menor tasa de error aparece para n = 900 y γ = 40, por lo que esos parámetros son los elegidos.
- La mayoría de los errores aparecen debido a que el estimador no ha podido diferenciar dos clases distintas, o no ha incorporado los nuevos objetos de la escena (es decir, errores de tipo "clase no generada"), como puede verse en la Figura 5.10.

- El número eficaz de partículas n_{eff} se mantiene prácticamente constante ante variaciones de n, y decrece ligeramente al utilizar valores elevados de γ.

Por otra parte se comprueba la distribución de partículas en las distintas clases, a lo largo de 5 iteraciones consecutivas, como se muestra en la Figura 5.1:

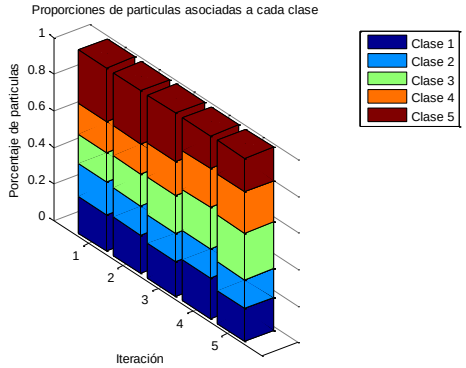


Figura 5.11. Proporción de partículas asociadas a cada clase, en cinco iteraciones consecutivas.

Los histogramas bidimensionales de los pesos de las partículas, asociados a las mismas iteraciones anteriores, se muestran en la Figura 5.12:

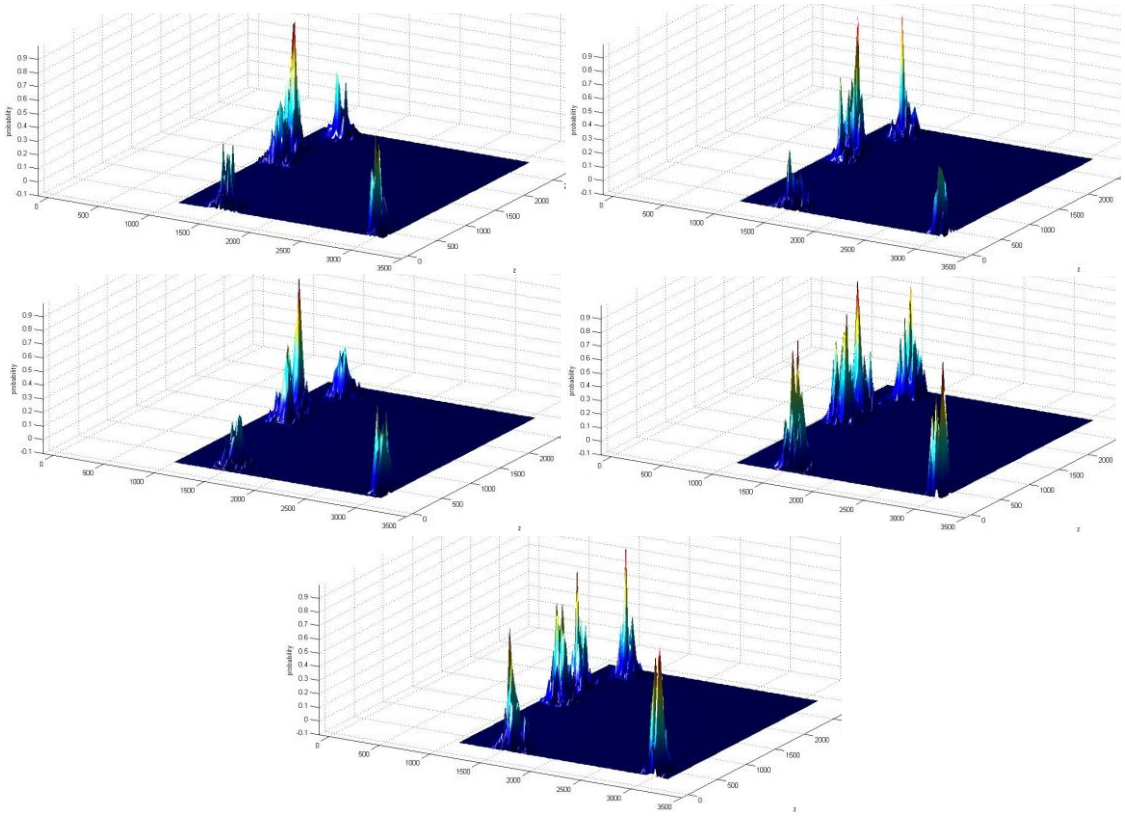


Figura 5.12. Histogramas bidimensionales de los pesos en cinco iteraciones consecutivas.

En capítulos anteriores se ha visto que el propósito de introducir un proceso de clasificación en el XPF es el de obtener una distribución de partículas lo más uniforme posible en torno a los distintos objetos de la escena, independientemente de la calidad de sensado de los mismos. Este efecto puede verse claramente en la Figura 5.12, y sobre todo en la Figura 5.11. El número de partículas asociadas a cada objeto es aproximadamente el mismo, manteniéndose en gran parte

constante a lo largo de las cinco iteraciones consecutivas analizadas, en las cuales aparece el número máximo de clases del experimento complejo.

5.1.1.3 Tiempos de ejecución

En la Figura 5.13 se muestra cómo se distribuyen los tiempos de ejecución empleados para procesar la clasificación de entrada, el XPF, y la clasificación de salida, para n = 300, n = 600 y n = 900.

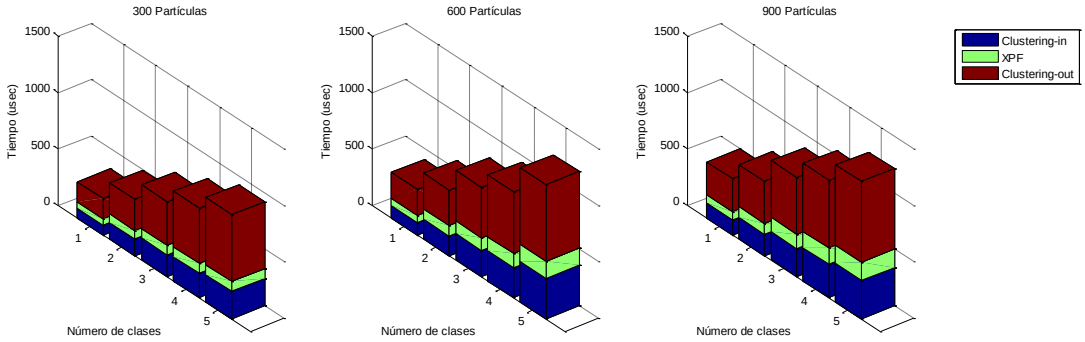


Figura 5.13. Distribución de los tiempos de ejecución en función del número de clases, en la clasificación de entrada y salida, y en el XPF, para distinto número de partículas.

- Como puede verse, el tiempo empleado en ejecutar el XPF apenas varía con el número de clases, como ya se había predicho al revisar la teoría en capítulos anteriores.
- Sin embargo, el necesario para ejecutar los procesos de clasificación sí varían. El tiempo requerido se incrementa según aumenta el número de clases, ya que al haber más grupos, aumenta el número de iteraciones necesarias hasta completar la clasificación. Además, si aumenta el número de grupos por lo general aumentará el número de medidas, que son los elementos a clasificar en el clustering de entrada. Por lo general hay menos medidas que número de partículas (además el número de medidas está limitado a 500), por lo que es necesario emplear más tiempo en la clasificación de partículas que en la de medidas.

5.1.2. Resultados del proceso de identificación

El procedimiento para la obtención de resultados es el mismo que en el caso del XPFCP: primero se varían los parámetros para obtener los que mejor funcionan, luego se analiza la fiabilidad frente a los mismos, y por último se muestran los tiempos de ejecución.

5.1.2.1 Variación de parámetros

En el proceso de identificación se varían dos parámetros: el tamaño del buffer en el cual se guarda tanto la trayectoria estimada como la real (*TAMBUFFER*), y la relación entre el voto positivo (*VOTOPOS* con el cual se premia al objeto cuya trayectoria se parezca más a la del robot) y el voto negativo (*VOTONEG* con el cual se penaliza al objeto cuya trayectoria se parezca menos a la del robot). En este caso se va a repetir el proceso con tres experimentos distintos: la prueba sencilla, la de dificultad media y la compleja.

a) Variación de TAMBUFFER

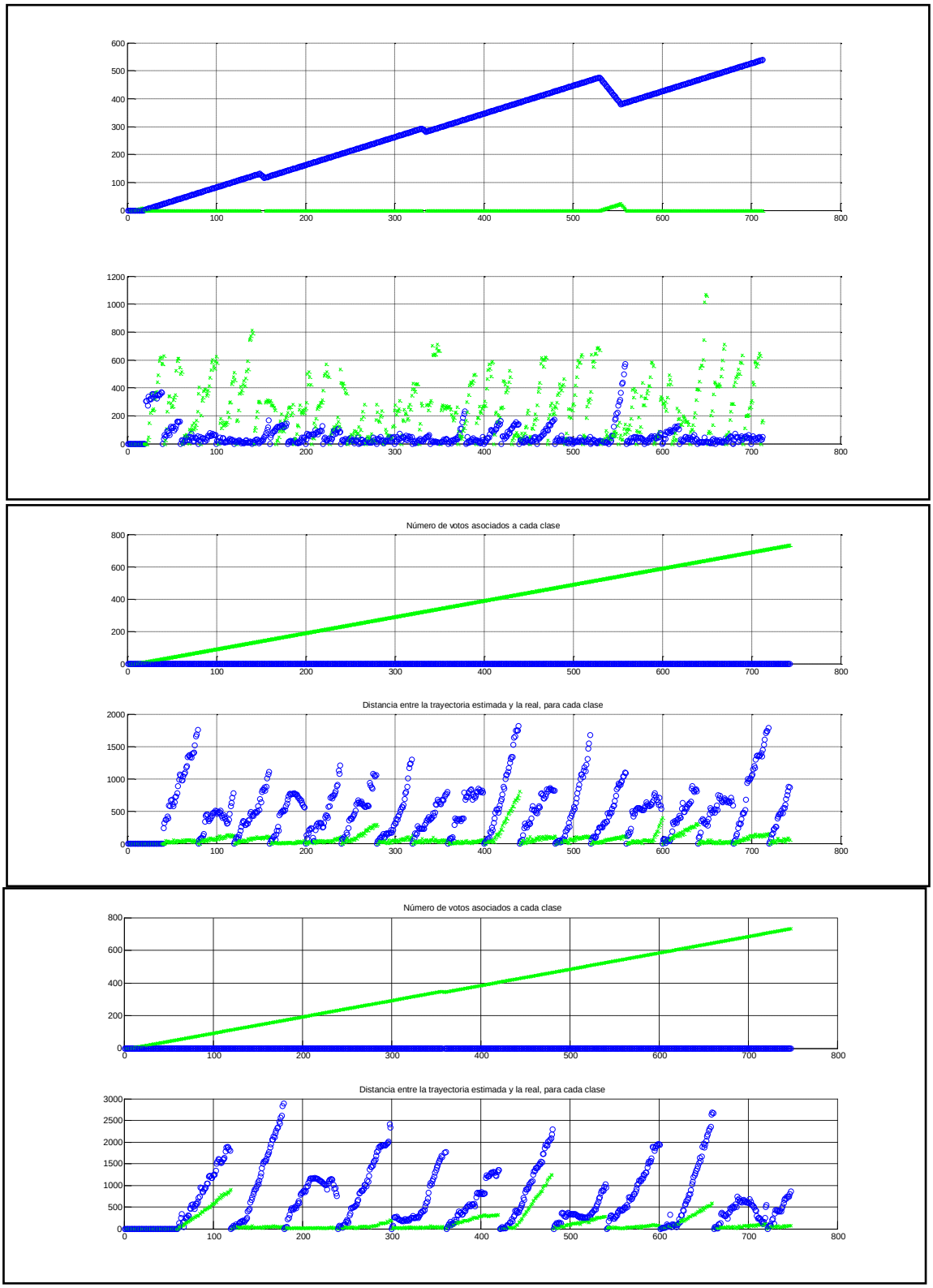


Figura 5.14. Resultados del proceso de identificación, en el experimento sencillo. Arriba: para *TAMBUFFER*=20. Centro: para *TAMBUFFER*=40. Abajo: para *TAMBUFFER*=60

En todo momento la diferencia de la trayectoria de la persona con respecto a la de la odometría es mucho mayor que la del robot, debido a que son muy distintas.

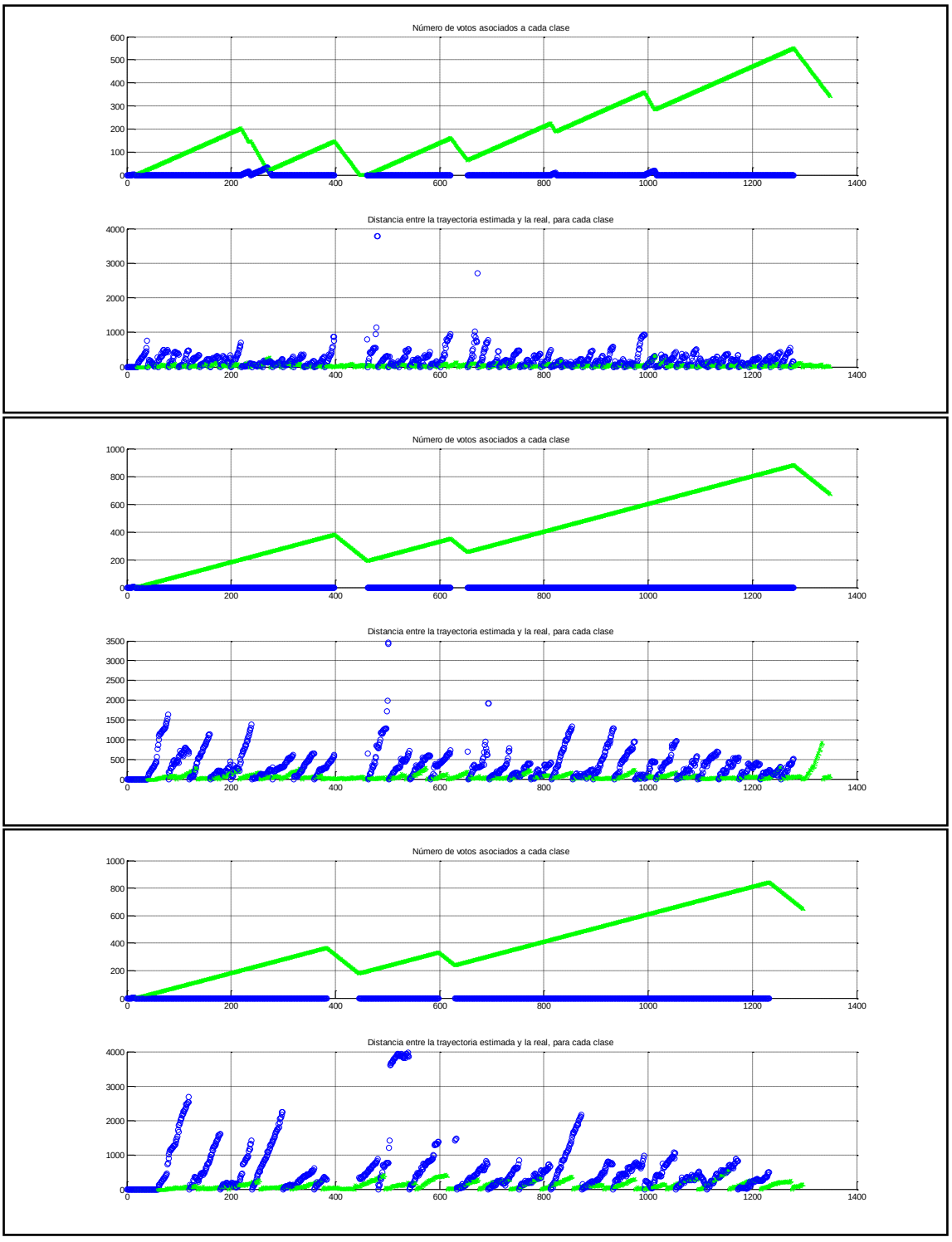


Figura 5.15. Resultados del proceso de identificación, en el experimento medio. Arriba: para *TAMBUFFER*=20. Centro: para *TAMBUFFER*=40. Abajo: para *TAMBUFFER*=60

En este experimento una persona trata de imitar el movimiento del robot. Sin embargo puede comprobarse que si bien la diferencia de trayectorias es menor que en el experimento anterior, no es suficiente como para generar errores de identificación.

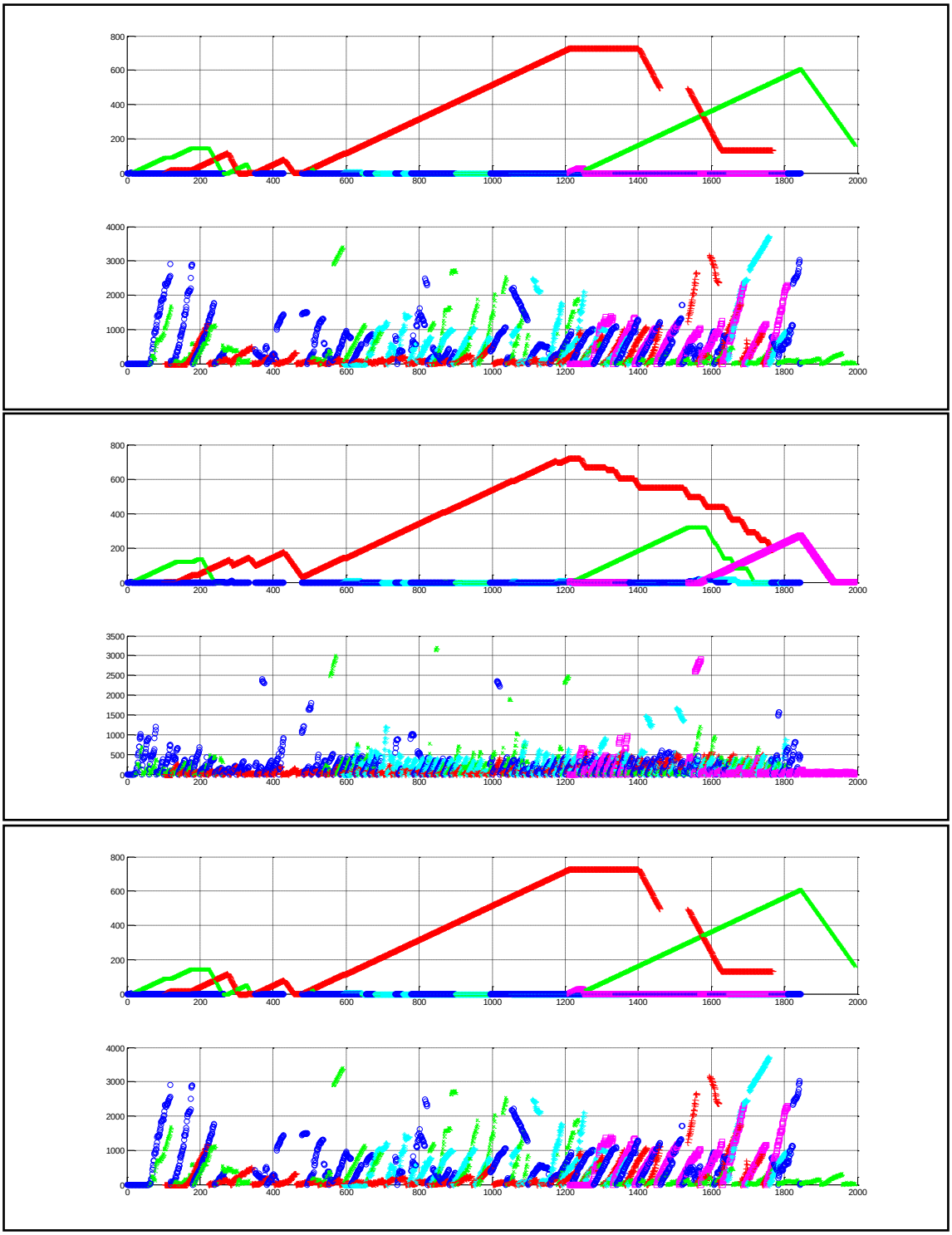


Figura 5.16. Resultados del proceso de identificación, en el experimento complejo. Arriba: para TAMBUFFER=20. Centro: para TAMBUFFER=40. Abajo: para TAMBUFFER 60.

El tamaño del buffer hace que el error de estimación de trayectorias se re-inicialice con más (valores de *TAMBUFFER* bajos) o menos (valores de *TAMBUFFER* altos) frecuencia. Si se re-inicializa con demasiada frecuencia, la diferencia de trayectorias entre las distintas clases va a ser parecida, por lo que pueden aparecer errores de identificación. Si se re-inicializa con poca frecuencia, la estimación se hace muy dependiente del ángulo estimado en el instante de la re-inicialización, disminuyendo la capacidad de corregir los errores.

b) Variación de VotoNegativo

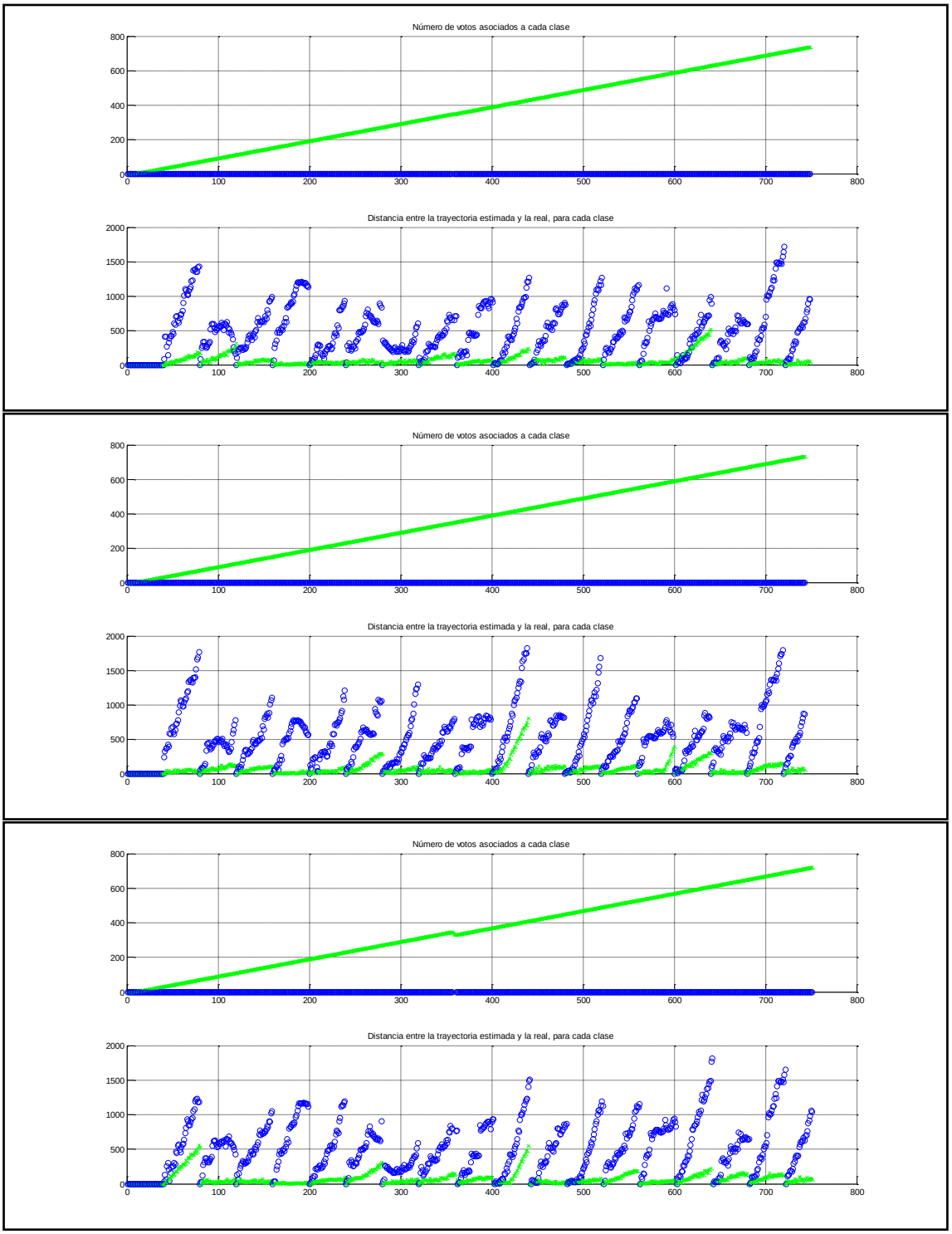


Figura 5.17. Resultados del proceso de identificación, en el experimento sencillo. Arriba: para *VOTONEG*=-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7.

No existe influencia en este experimento ya que no se producen errores de estimación, y el voto χ_j asociado a la clase del robot está incrementándose prácticamente durante todo el experimento.

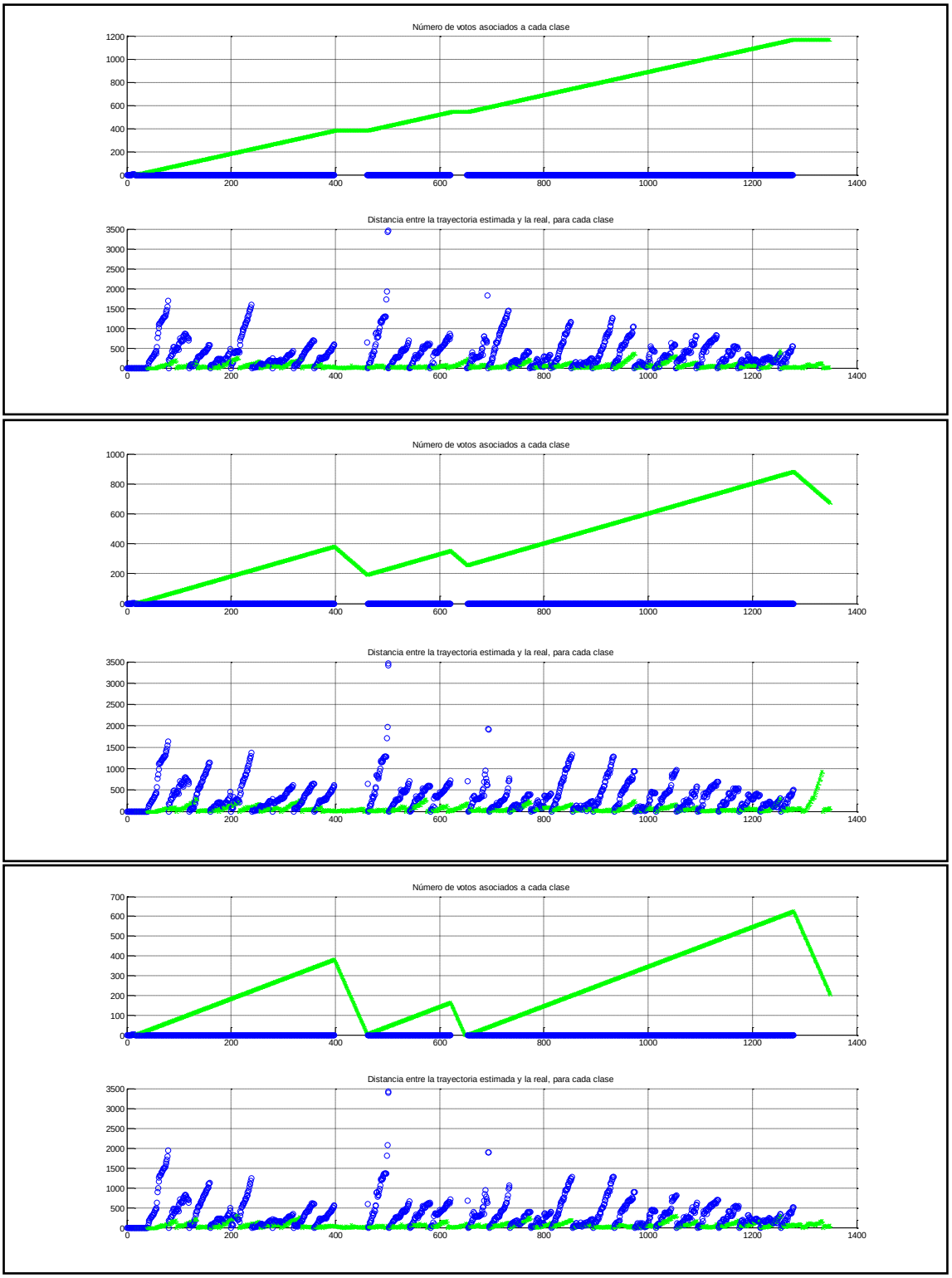


Figura 5.18. Resultados del proceso de identificación, en el experimento medio. Arriba: para *VOTONEG* =-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7.

En este experimento aparecen dos errores cortos de identificación. Se observa cómo el voto asociado al robot cae con más o menos rapidez. Utilizando un valor de -7 para el *VOTONEG* se ve que el voto llega a cero durante estos errores, lo que se traduce en una posible conmutación de identidad del robot. Es necesario buscar un equilibrio entre inercia del valor de votos y rapidez para corregir errores de identificación.

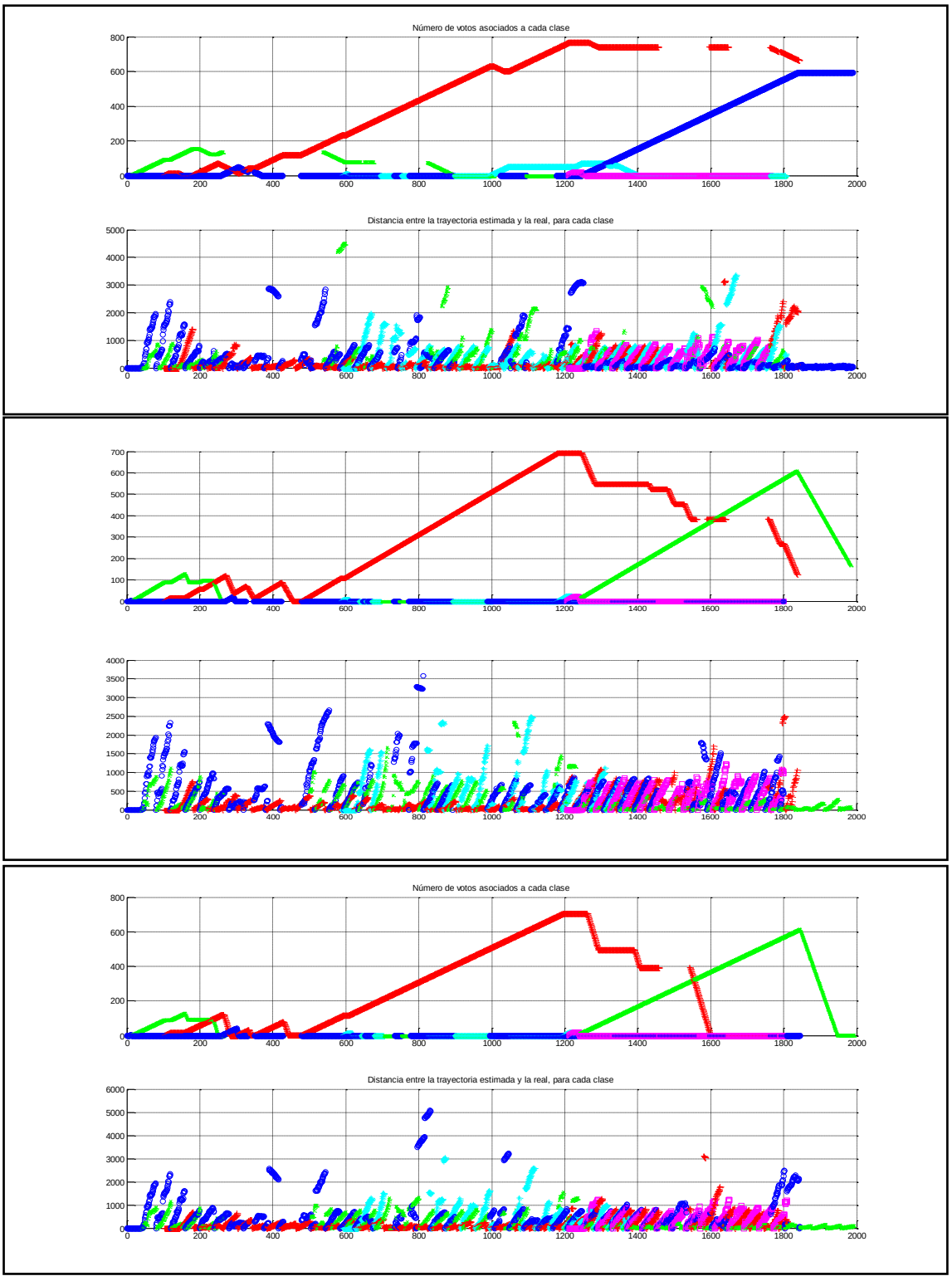


Figura 5.19. Resultados del proceso de identificación, en el experimento complejo. Arriba: para *VOTONEG* =-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7.

En este experimento existe una conmutación de la clase asociada al robot, en torno a la iteración 1200, lo que hace que una clase que no es el robot se vea repentinamente con una gran cantidad de votos, mientras que el robot no tiene votos asociados. Las distintas relaciones entre *VOTOPOS* y *VOTONEG* corrigen la situación con mayor o menor rapidez, pero a costa de mantener con más o menos efectividad la identidad del robot en situaciones en las que se pierden votos durante un corto periodo de tiempo.

5.1.2.2 Análisis de la fiabilidad

A continuación se muestran las comparaciones entre la identidad real del robot, obtenida a mano y la que el sistema determina (τ_r).

a) *Ante variaciones de TAMBUFFER*

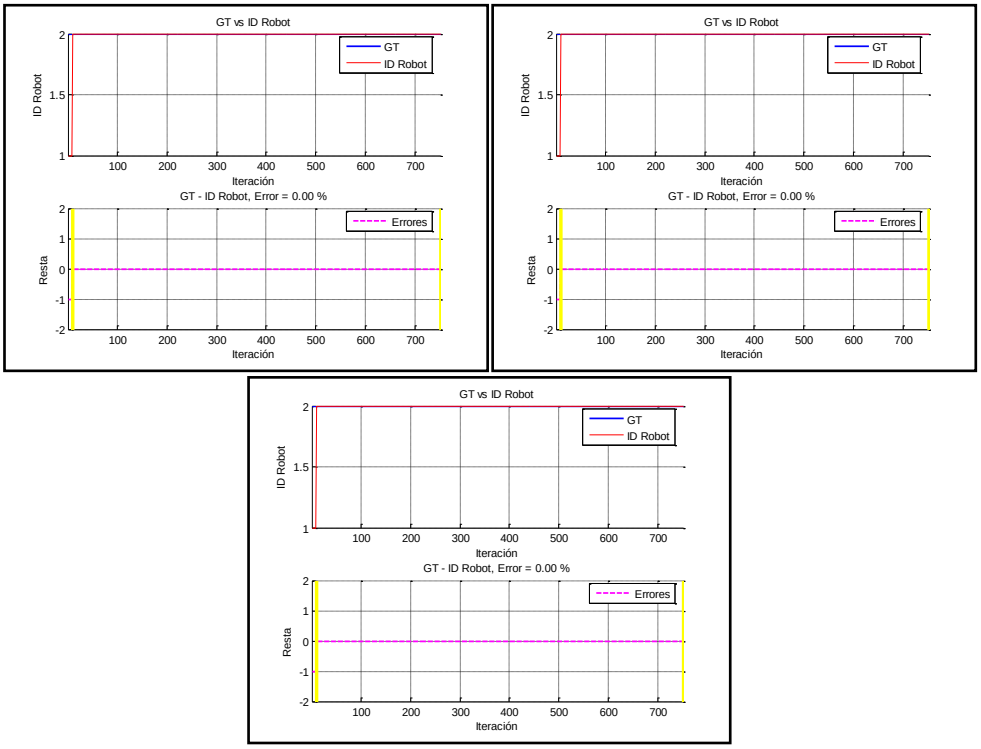


Figura 5.20. Resultados del proceso de identificación, en el experimento sencillo. Arriba: para *TAMBUFFER*=20. Centro: para *TAMBUFFER*=40. Abajo: para *TAMBUFFER*=60.

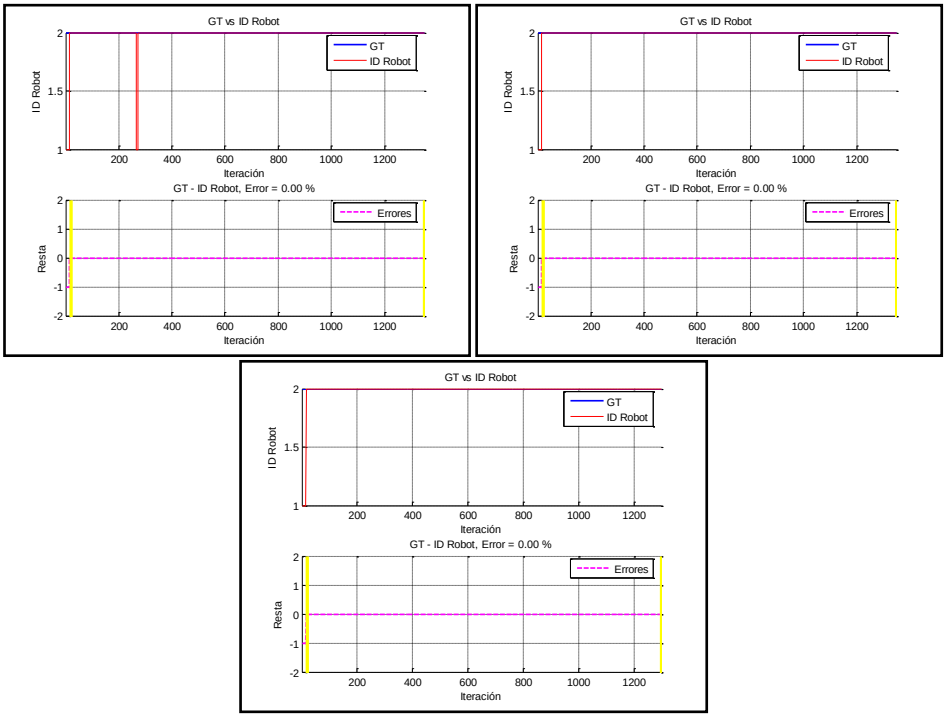


Figura 5.21. Resultados del proceso de identificación, en el experimento medio. Arriba: para *TAMBUFFER*=20. Centro: para *TAMBUFFER*=40. Abajo: para *TAMBUFFER*=60.

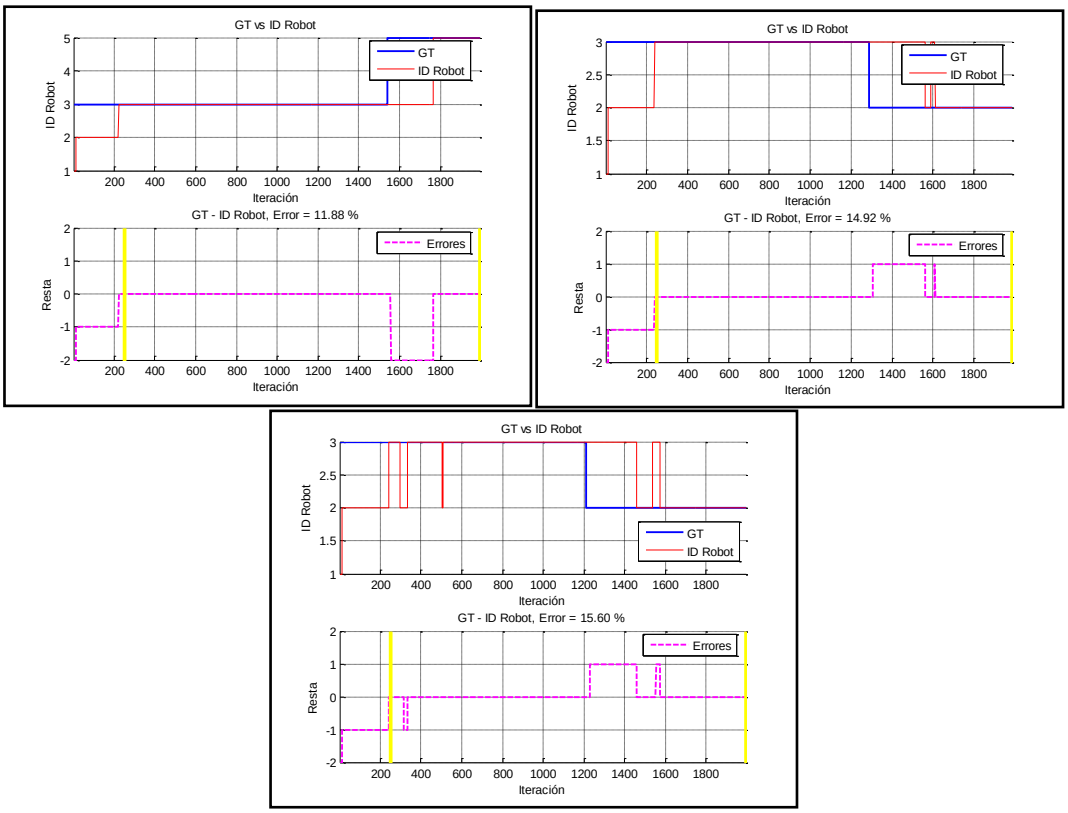


Figura 5.22. Resultados del proceso de identificación, en el experimento complejo. Arriba: para *TAMBUFFER*=20. Centro: para *TAMBUFFER*=40. Abajo: para *TAMBUFFER*=60.

b) *Ante variaciones de VOTONEG*

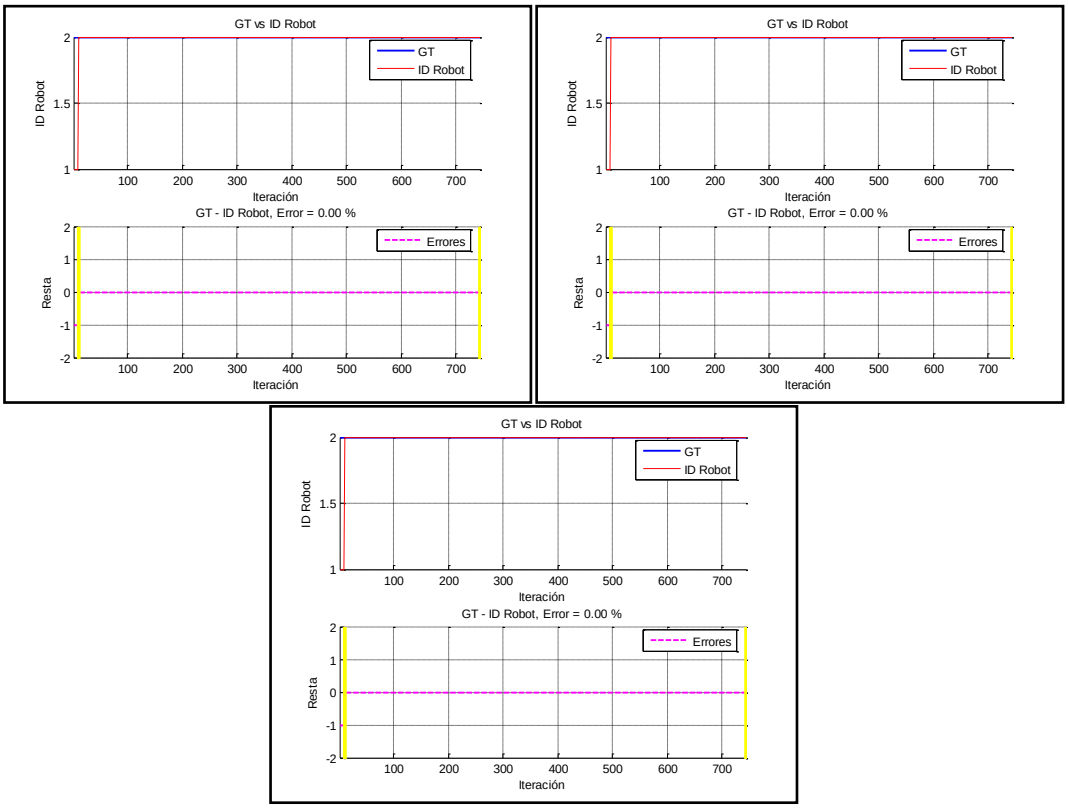


Figura 5.23. Resultados del proceso de identificación, en el experimento sencillo. Arriba: para *VOTONEG*=-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7.

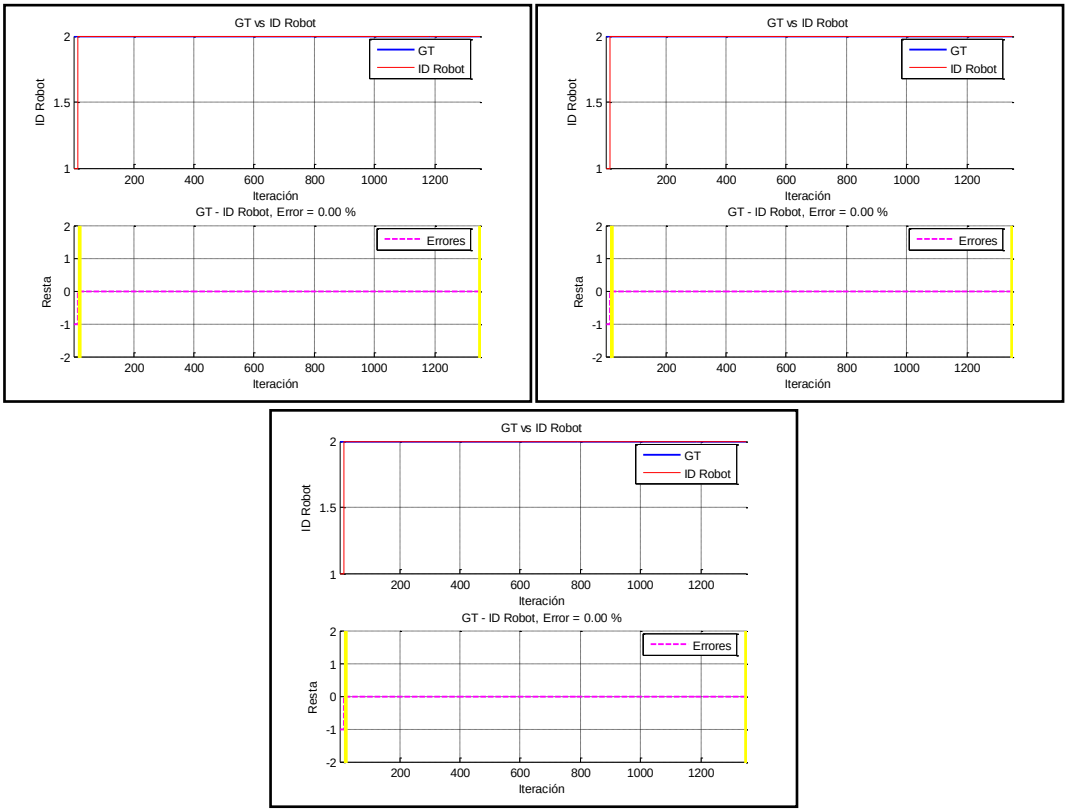


Figura 5.24. Resultados del proceso de identificación, en el experimento medio. Arriba: para *VOTONEG* =-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7

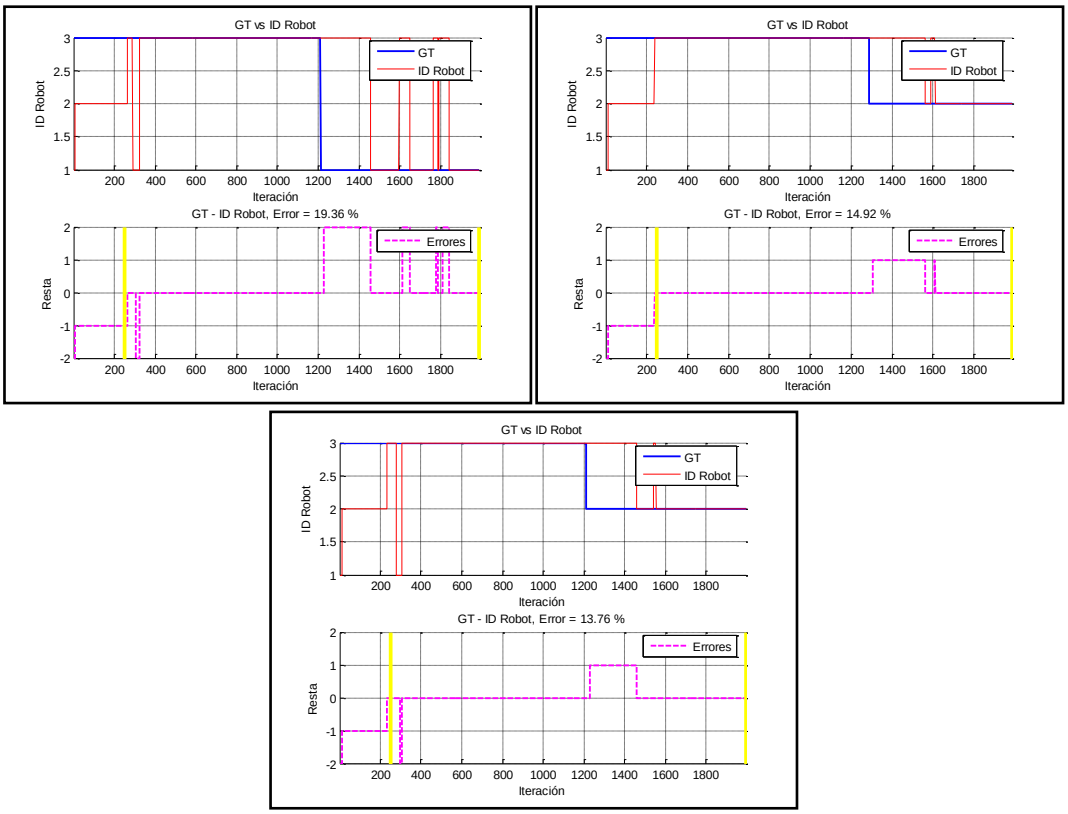


Figura 5.25. Resultados del proceso de identificación, en el experimento complejo. Arriba: para *VOTONEG* =-1. Centro: para *VOTONEG* =-4. Abajo: para *VOTONEG* =-7.

Los datos de la Figura 5.20, Figura 5.21, Figura 5.22, Figura 5.23, Figura 5.24 y Figura 5.25 se recopilan y presentan a continuación.

Test sencillo (v1)		
Características del test	2 objetos durante toda la duración.	
Parámetros	n 900, γ 40%, *TAMBUFFER* 40, *VOTONEG* -4	
Fiabilidad	0 % de errores	
Sensibilidad a parámetros	*TAMBUFFER* - 50% variación -> std:	0
	VOTONEG - 50% variacion -> std:	0
n_{eff} (%)		30.58
t_{exe}		18 FPS

Tabla 5.9. Fiabilidad del proceso de identificación considerando errores superiores a un segundo.

Test medio (v4)		
Características del test	2 objetos durante toda la duración.	
Parámetros	n 900, γ 40%, *TAMBUFFER* 40, *VOTONEG* -4	
Fiabilidad	0 % de errores	
Sensibilidad a parámetros	*TAMBUFFER* - 50% variación -> std:	0
	VOTONEG - 50% variacion -> std:	0
n_{eff} (%)		25.98
t_{exe}		18 FPS

Tabla 5.10. Fiabilidad del proceso de identificación considerando errores superiores a un segundo.

Test complejo (v6)	
Características del test	Entre 1 y 5 objetos.
Parámetros	n 900, γ 40%, *TAMBUFFER* 40, *VOTONEG* -4
Fiabilidad	14,92 % de errores
Sensibilidad a parámetros	*TAMBUFFER* - 50% variación -> std: 1,980841572
	VOTONEG - 50% variacion -> std: 2,955762733
n_{eff} (%)	25.98
t_{exe}	18 FPS

Tabla 5.11. Fiabilidad del proceso de identificación considerando errores superiores a un segundo.

	Análisis parámetros			
TAMBUFFER (*VOTONEG*=-4)	20	***40***	60	std
% error	0	0	0	0
VOTONEG (*TAMBUFFER*=40)	-1	***-4***	-7	std
% error	0	0	0	0

Tabla 5.12. Recopilación de tasas de errores en el test sencillo contando errores superiores a un segundo.

	Análisis parámetros			
TAMBUFFER (*VOTONEG* =-4)	20	***40***	60	std
% error	0	0	0	0
VOTONEG (*TAMBUFFER*=40)	-1	***-4***	-7	std
% error	0	0	0	0

Tabla 5.13. Recopilación de tasas de errores en el test sencillo contando errores superiores a un segundo.

	Análisis parámetros			
TAMBUFFER (*VOTONEG* =-4)	20	***40***	60	std
% error	11,88	14,92	15,6	1,980842
VOTONEG (*TAMBUFFER*=40)	-1	***-4***	-7	std
% error	19,36	14,92	13,76	2,955763

Tabla 5.14. Recopilación de tasas de errores en el test sencillo contando errores superiores a un segundo.

- Puede comprobarse que existe una baja sensibilidad a variaciones de parámetros.
- Teniendo en cuenta la complejidad de los test, sobre todo del último, la tasa de errores es baja.

5.1.2.3 Tiempos de ejecución

En la Figura 5.26 se hace una comparación entre el tiempo de ejecución del XPFCP (ya examinado anteriormente), y el del proceso de identificación de robots.

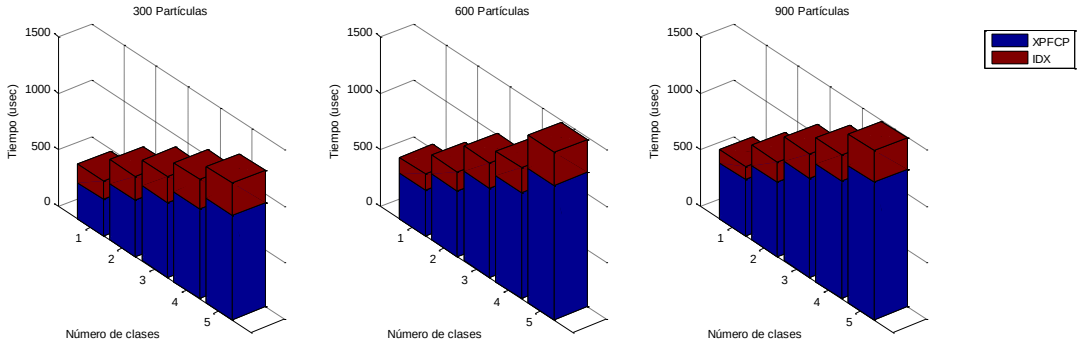


Figura 5.26. Tiempos de ejecución del sistema. Comparación entre el XPFCP y el proceso de identificación.

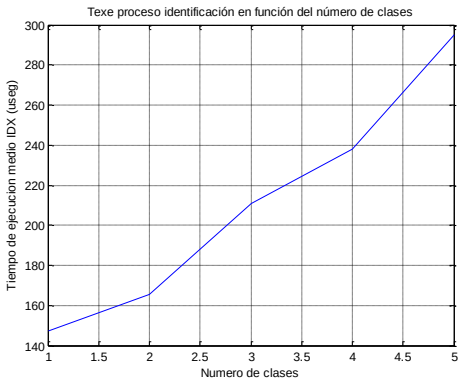


Figura 5.27. Tiempo de ejecución del proceso de identificación en función del número de clases

- El tiempo de ejecución del proceso de identificación aumenta según lo hace el número de clases, como puede verse de forma más clara en la Figura 5.27 ya que necesita comparar las trayectorias estimadas de cada clase mediante la odometría, con las trayectorias obtenidas mediante el XPFCP.
- En comparación con el XPFCP, el tiempo requerido para ejecutar el proceso de identificación es bajo, manteniendo las especificaciones de tiempo real requeridas.

En la Figura 5.28 se han representado los tiempos de ejecución tanto del XPFCP como del proceso de identificación en función del tiempo, para la prueba compleja. Los tiempos de ejecución permiten que las restricciones de tiempo real se cumplan, permitiendo una velocidad de la aplicación en cuadros por segundo suficiente y sobre todo, constante.

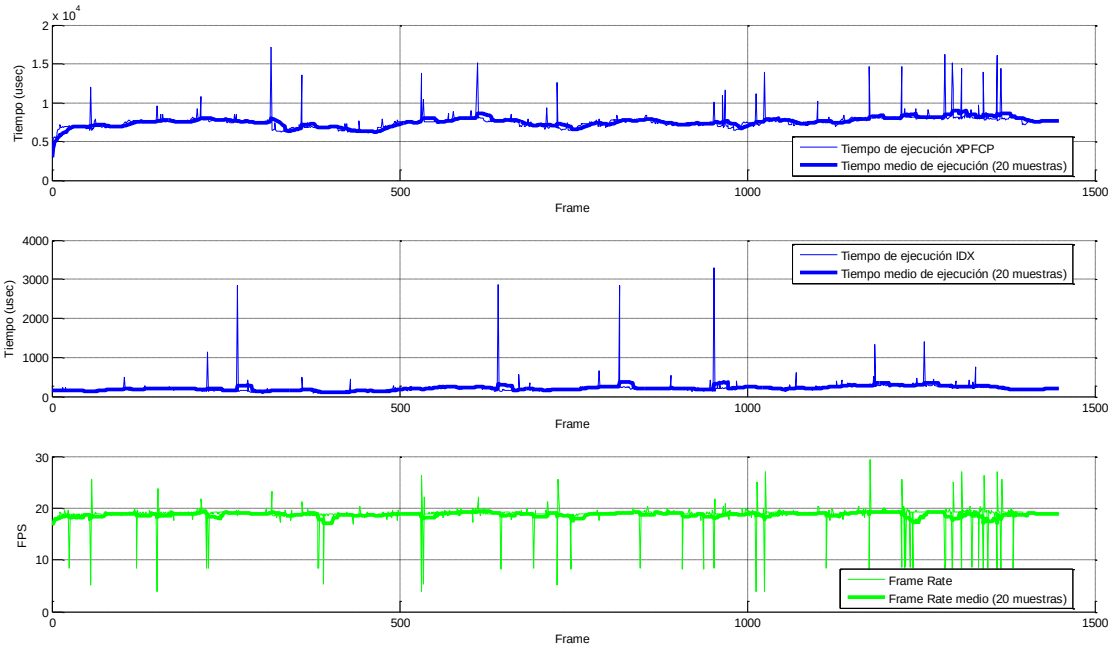


Figura 5.28. Arriba: tiempo de ejecución del XPFCP a lo largo de la prueba compleja. Centro: tiempo de ejecución del proceso de identificación a lo largo de la prueba compleja. Abajo: velocidad de la aplicación en cuadros por segundo, en la prueba compleja.

5.2 Resultados 3DPF

En esta sección se analiza el funcionamiento del sistema implementado para realizar el seguimiento tridimensional.

Únicamente se dispone de un test, relativamente complejo. En él, en una primera parte aparece un robot en la escena (escena A), moviéndose. Posteriormente se detiene, y una persona entra para colocar una serie de papeleras. A continuación aparece una escalera, y el robot se mueve hasta situarse debajo de ella, para luego salir (escena B). La duración aproximada del experimento es de 25 segundos.

En la Figura 5.1 se muestra una secuencia con los resultados finales obtenidos:

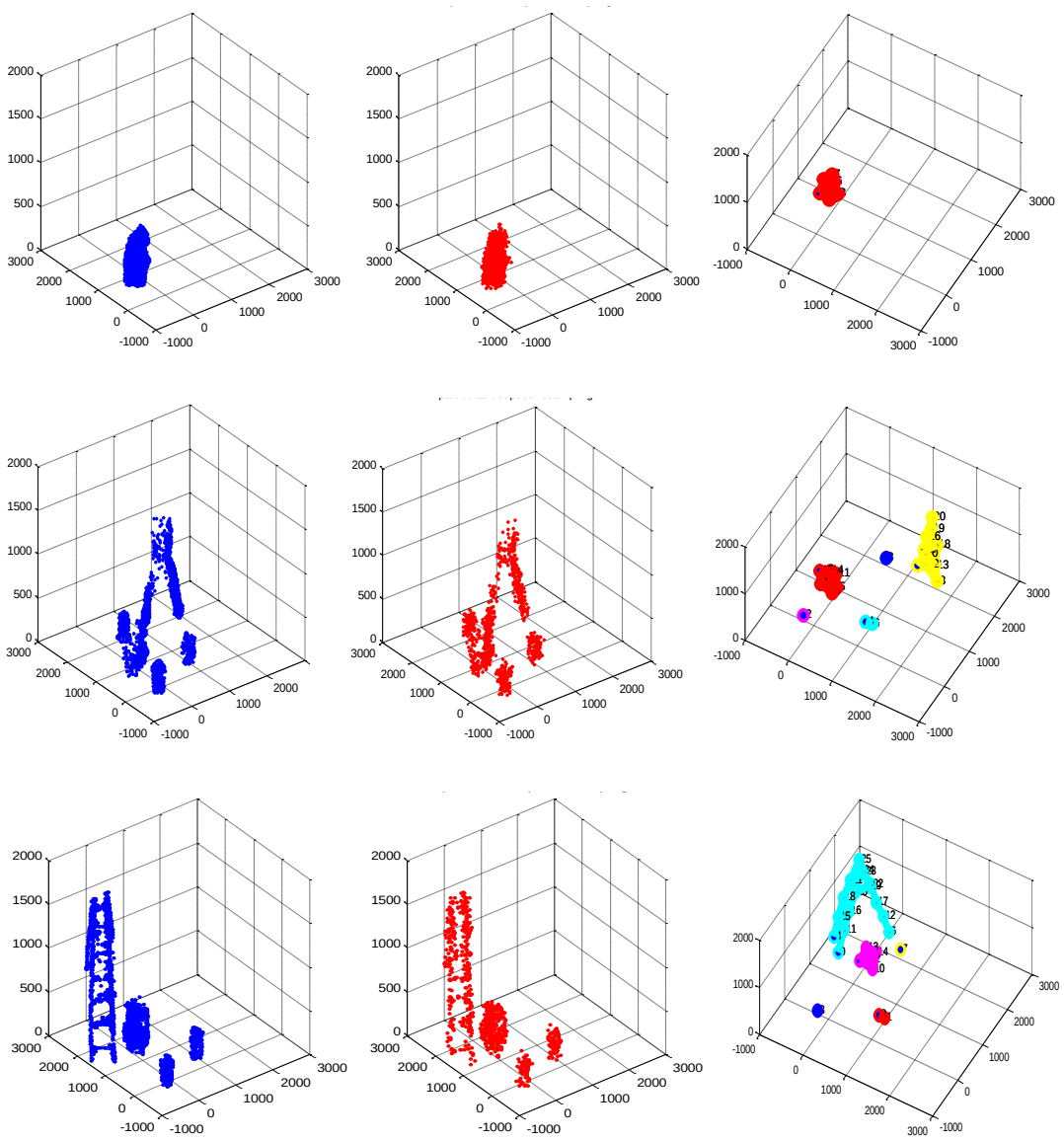


Figura 5.29. Secuencia con los resultados obtenidos al ejecutar el 3DPF. En cada fila se muestra una iteración distinta. En la columna izquierda se muestran las medidas en azul. En la columna central se muestra la distribución de partículas después del paso de selección. En la columna derecha se muestra la salida final del algoritmo. Se representa cada clase con un color distinto.

5.2.1. Ajuste de parámetros

El procedimiento seguido se divide en tres etapas:

- *Primera etapa*: Se ajustan los parámetros pertinentes de manera que la distribución de partículas sea la mejor posible. No existe un criterio objetivo para hacer esto, ya que el número eficaz de partículas pierde su relevancia al añadir el paso de saturación de pesos (ver ecuación 2.12). Por lo tanto, se valorará que haya una densidad de partículas suficiente en torno a los objetos pobremente sensados.

- *Segunda etapa*: Ahora el objetivo es la correcta formación de las clases de salida, por lo que es posible utilizar una comparación con el número real de clases en la escena, obtenido por observación directa, para establecer una medida objetiva de la calidad del proceso.

- *Tercera etapa*: En la primera etapa se busca tener la mejor distribución de partículas posible, para así disponer de una base fiable sobre la que modificar los parámetros de la segunda parte. Pero esto normalmente se traduce en un incremento de los tiempos de ejecución, ya que para conseguirlo es necesario incrementar el número total de partículas. El objetivo de este tercer paso es por tanto obtener un equilibrio razonable entre las tasas de error en la salida y el tiempo empleado para ejecutar el algoritmo.

5.2.1.1. Primera etapa

Los parámetros a ajustar son: número de partículas total *n*, porcentaje de partículas que provienen de medidas en el paso de re-inicialización γ, porcentajes de saturación de pesos M_{min} y M_{max}, umbral para considerar las medidas como redundantes *umbralHist*, y por último, la distancia dentro de la cual se buscan medidas en torno a una partícula, para la etapa de pesado, *L/2*.

Los resultados obtenidos sobre el mismo instante de la secuencia. Las medidas en este mismo instante aparecen en la Figura 5.30,

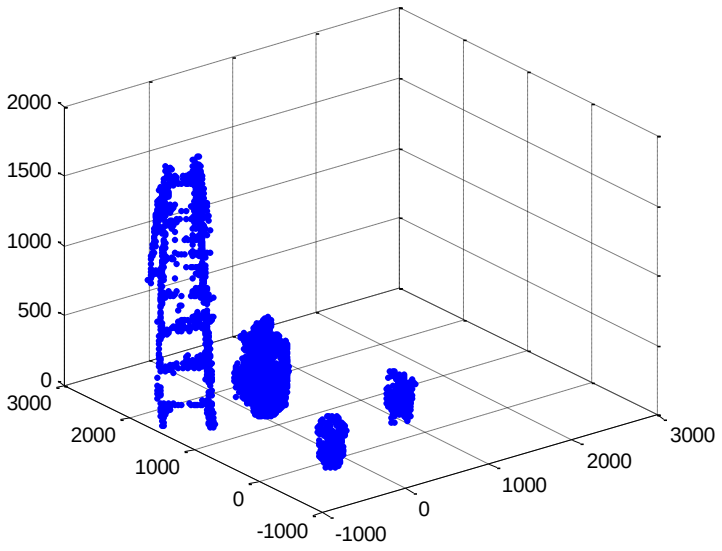


Figura 5.30. Medidas obtenidas en el instante analizado.

a) Número total de partículas (n)

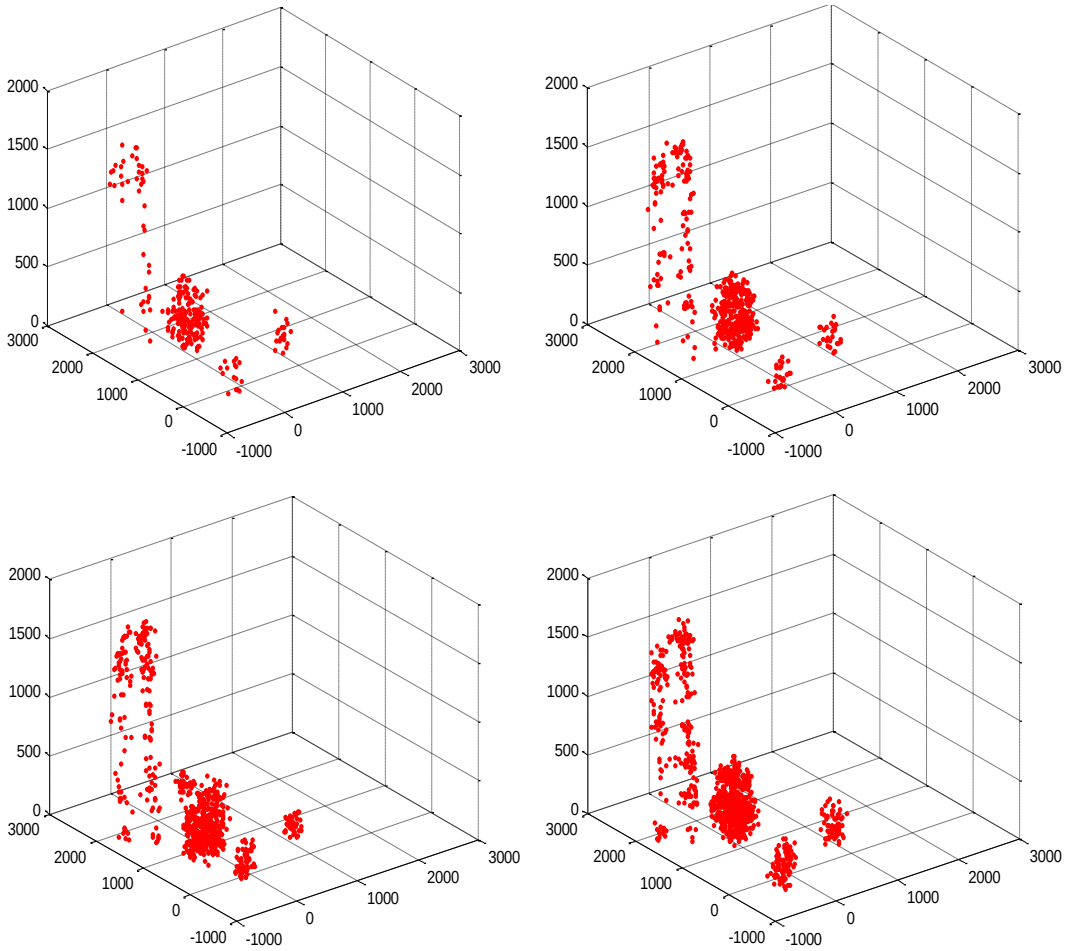


Figura 5.31. Distribución obtenida con (de izquierda a derecha y arriba a abajo): n=500, n=1000, n=1500 y n=2000.

Como puede comprobarse, utilizando el mayor número de partículas se consigue que haya más asociadas a los elementos pobremente sensados: la escalera y papeleras, por lo que se elige utilizar 2000 partículas.

b) Porcentaje de partículas provenientes de medidas en el paso de re-inicialización (γ)

Al igual que en el apartado a) y en apartados posteriores, la distribución se analiza sobre las medidas obtenidas que se muestran en la Figura 5.30. En este caso los resultados se muestran en la Figura 5.32, a continuación:

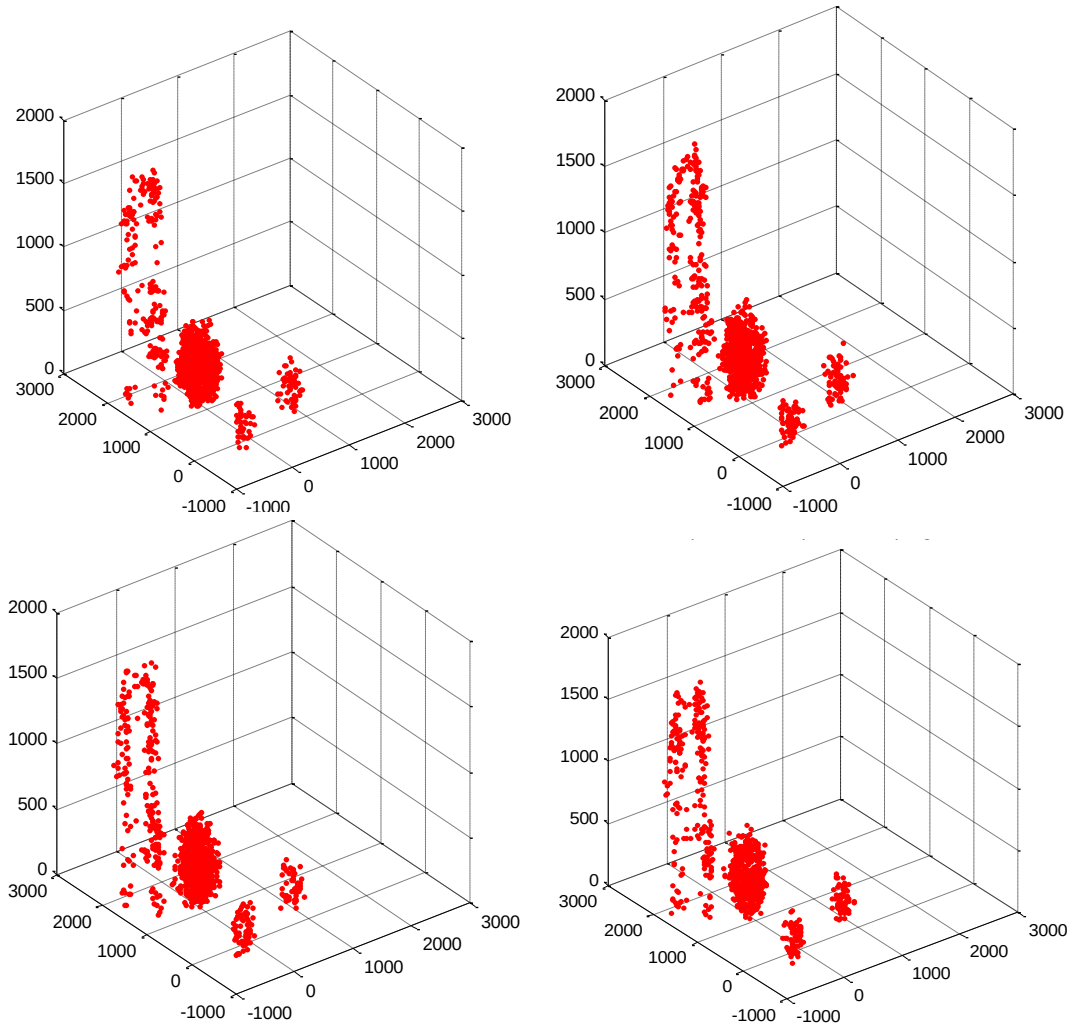


Figura 5.32. Porcentajes de partículas provenientes de medidas en el paso de re-inicialización, de arriba a abajo e izquierda a derecha: γ=10%, γ=20%, γ=30% y γ=40%.

Lógicamente, cuantas más partículas provengan de medidas, más se parece la distribución a los objetos medidos. Se ha establecido el límite en un 40% para no quitar rigor al sistema de seguimiento, y éste es el valor que mejores resultados da.

c) Porcentajes de saturación de pesos (α-β)

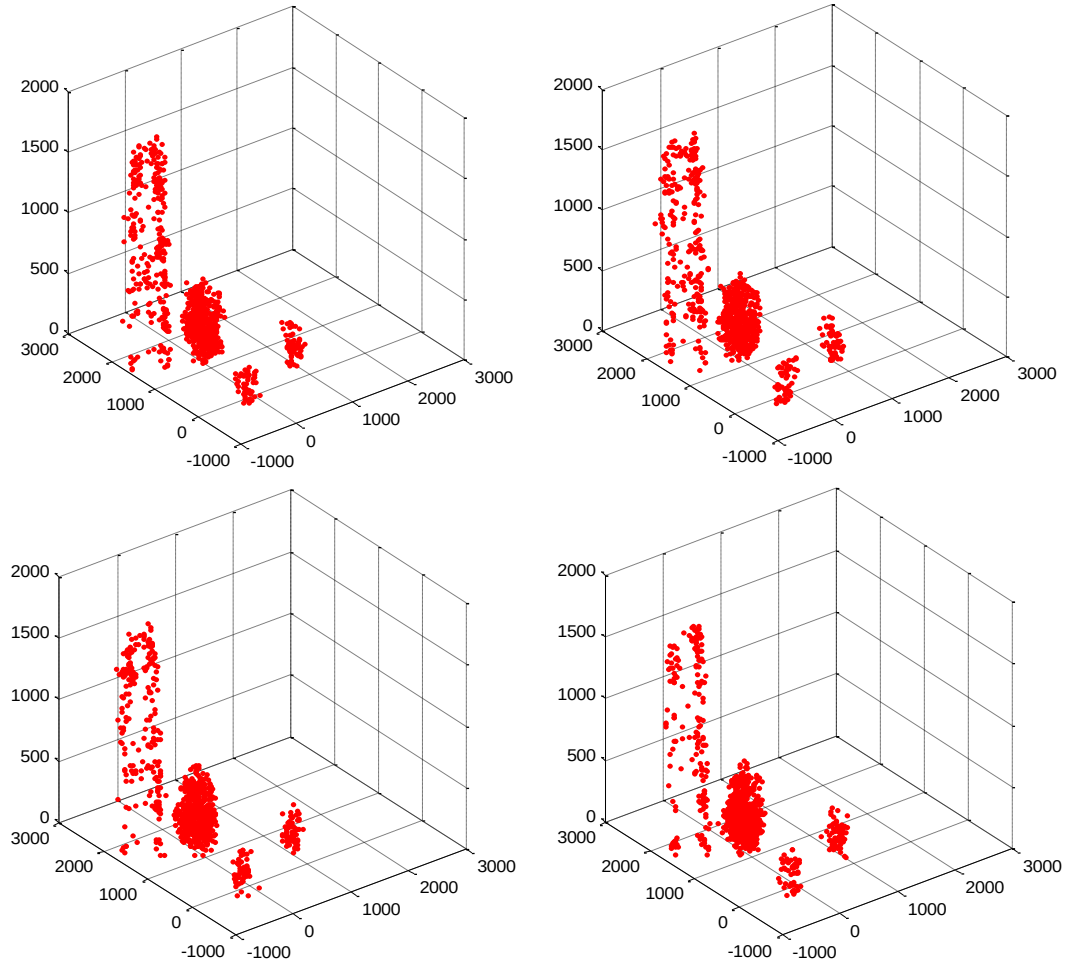


Figura 5.33. Porcentaje de saturación de pesos α-β (de arriba a abajo e izquierda a derecha): 5%-0%, 10%-0%, 20%-0%, 30%-0%.

Se consiguen los mejores resultados utilizando un 5% como límite superior, ya que es el valor que evita en mayor medida la concentración de partículas en torno al objeto mejor sensado (el robot), haciendo que se redistribuyan en torno al resto de objetos.

En la se muestra gráficamente el efecto de saturación de los pesos. Se han representado las partículas como esferas cuyo radio es proporcional a su peso:

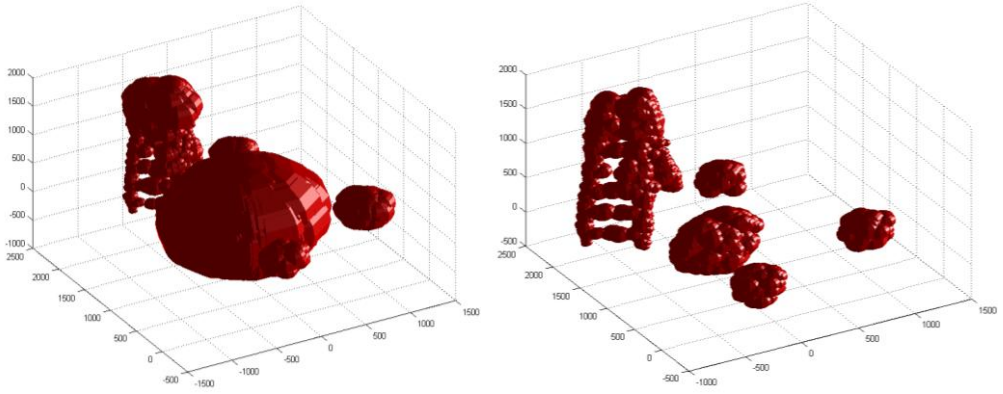


Figura 5.34. Arriba: Tamaño de partículas en función de su peso. Izquierda: saturación 30-0. Derecha: saturación 5-0. Abajo: histogramas de pesos. Izquierda: antes de la saturación. Derecha: después de la saturación

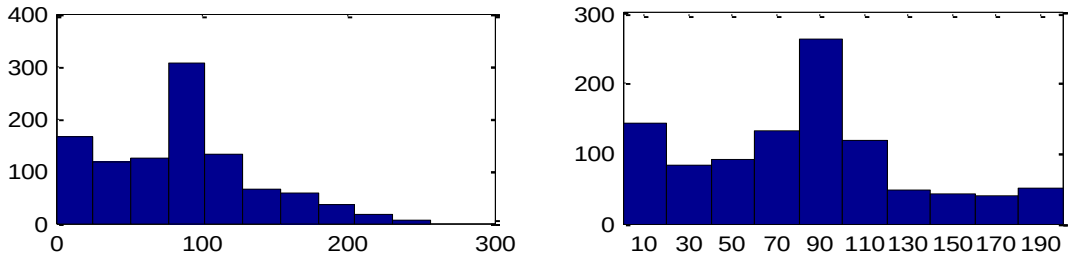


Figura 5.35. Histogramas de pesos. Izquierda: antes de la saturación. Derecha: después de la saturación.

Si no se utiliza el algoritmo de saturación de pesos, las partículas con mayor importancia están en su mayor parte en torno al robot y el final de la escalera, por lo que en el algoritmo de resampling se eliminará una gran cantidad de partículas ubicadas en el resto de zonas, dando lugar a una mala distribución, poco representativa de los objetos de la escena.

d) Umbral de medidas redundantes (umbralHist)

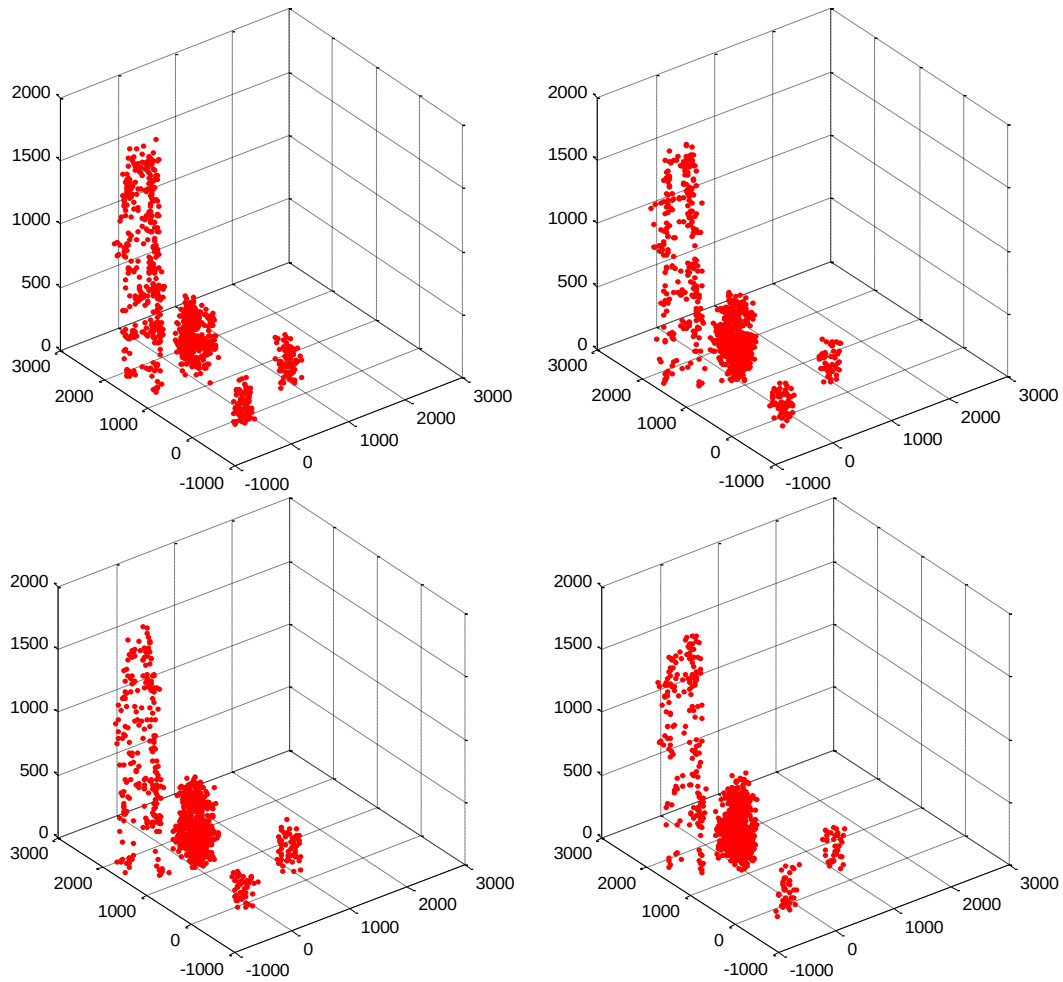


Figura 5.36. Umbral de medidas redundantes, de arriba a abajo e izquierda a derecha: *umbralHist*=600, *umbralHist*=900, *umbralHist*=1200, *umbralHist*=1500.

Estableciendo un menor límite se consigue que haya más partículas en los objetos pobremente sensados, por lo que se elige el mínimo valor, 600.

El efecto de la ecualización de medidas en los pesos puede verse de la misma forma a la mostrada en la , a continuación:

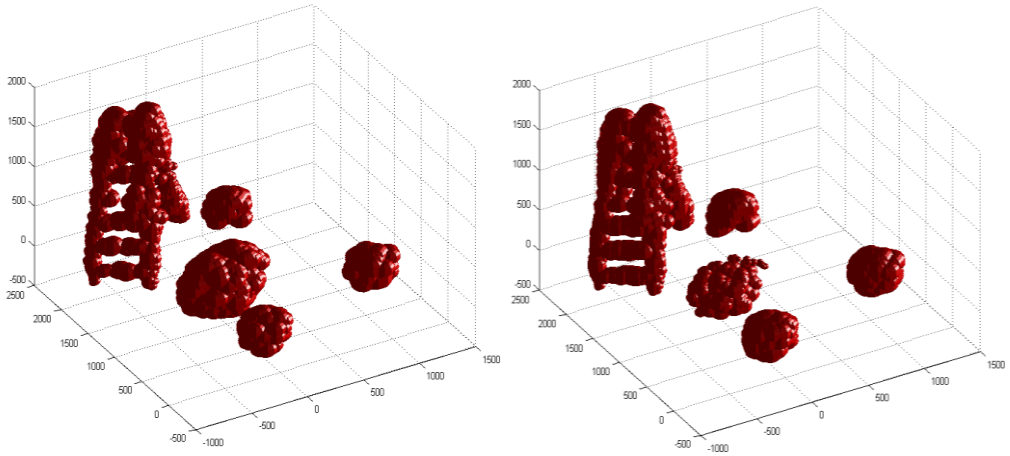


Figura 5.37. Partículas con tamaño en función de su peso. Izquierda: sin ecualización de medidas. Derecha: con ecualización de medidas.

e) Distancia en torno a la cual se buscan medidas en torno a una partícula (L/2).

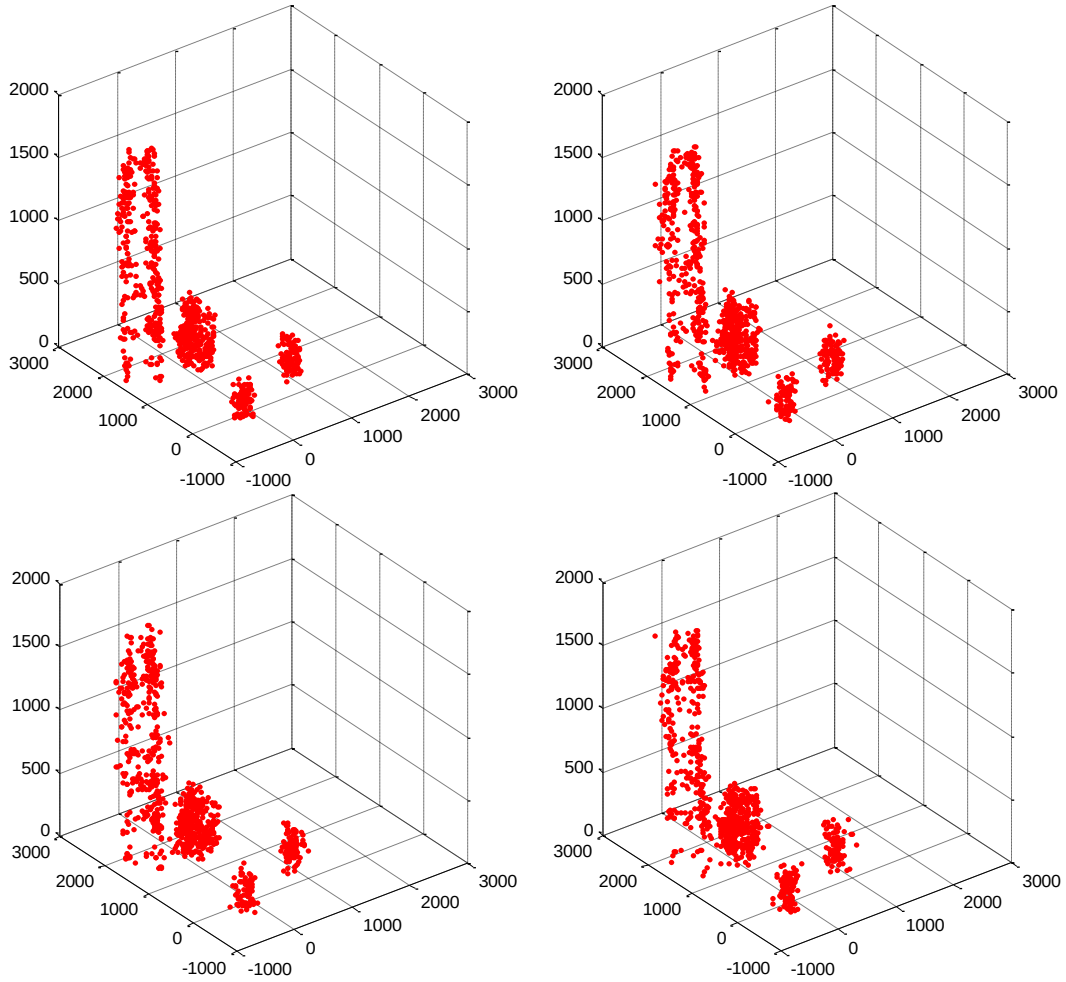


Figura 5.38. Distancia en torno a la cual se buscan medidas alrededor de una partícula. De arriba a abajo e izquierda a derecha: *L*/2=50, *L*/2=100, *L*/2=200, *L*/2=300.

Si es demasiado grande, las partículas se concentran en torno a los objetos mejor sensados. Por lo tanto, se elige el valor en el que esto influye menos, es decir, el menor: 50. Puede verse

como la reconstrucción es la mejor de todas, evitando que las partículas se dispersen demasiado. Aquí concluye la primera parte del proceso de ajuste de parámetros. Se ha llegado a la conclusión de que para obtener la mejor distribución posible de partículas, es necesario utilizar:

n	2000
γ	40%
α-β	5%-0%
umbralHist	600
L/2	50

5.15. Parámetros para obtener la mejor distribución de partículas posible.

5.2.1.2. Segunda etapa

Una vez obtenida una buena distribución de partículas, el siguiente paso es ajustar los parámetros relacionados con el algoritmo de conectividad, para obtener la menor tasa de errores posible.

Como se ha comentado anterormente, el experimento consiste en dos escenas (A y B). Se analizarán los errores de cada una por separado, ya que tienen características diferentes, las cuales favorecen ciertos parámetros de distinta forma.

En esta segunda etapa los parámetros a ajustar son: distancia de agrupación para el algoritmo de clasificación k-medias por altura *distM*, distancia de conectividad *distC*, y ponderación de las coordenadas XZ e Y a la hora de calcular esta distancia, ξ y δ.

a) Distancia de agrupación en k-medias (distM)

El instante analizado es el mismo que muestra la Figura 5.30. Se muestran los resultados en las distintas alturas de agrupación (indicadas en el título de cada gráfica en milímetros), en la Figura 5.40.

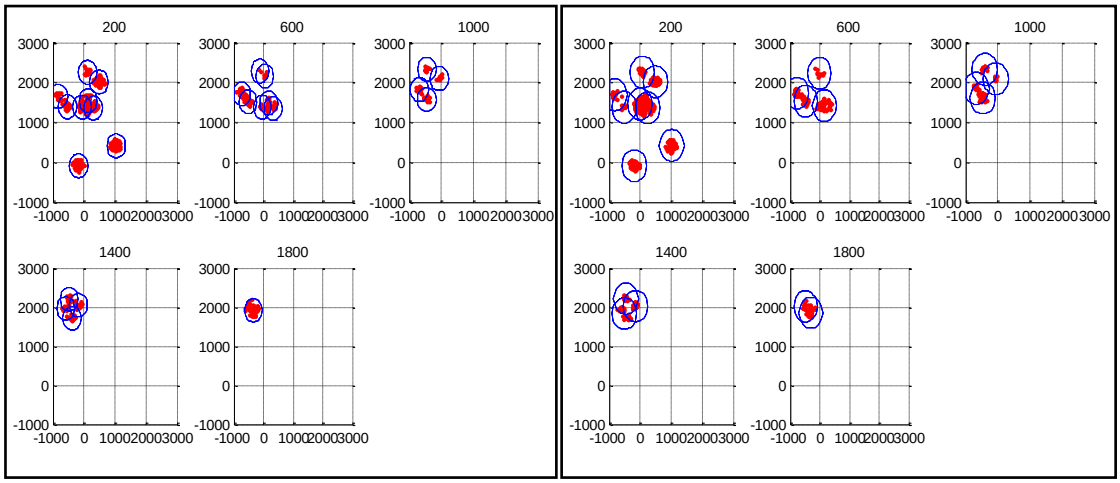


Figura 5.39. Distancia de agrupación en k-medias por niveles. Izquierda: *distM*=300. Derecha: *distM*=400.

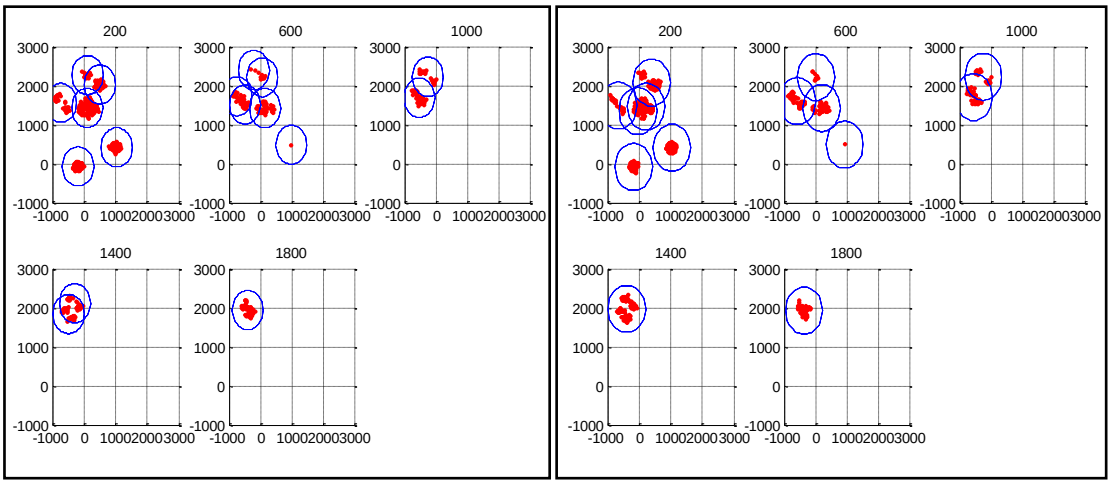


Figura 5.40. de agrupación en k-medias por niveles. Izquierda: *distM*=500. Derecha: *distM*=600.

Se inspecciona visualmente qué valor da el menor número de errores (no generados o duplicados). Los duplicados no son necesariamente algo perjudicial, porque el algoritmo de conectividad los unirá posteriormente. Se comprueba que para obtener centroides más estables en el tiempo y una mejor reconstrucción, la distancia a utilizar es la mínima analizada, en este caso 300.

b) Distancia de conectividad (distC) y ponderación XZ-Y (ξ-δ)

Dado que estos parámetros están relacionados entre sí, se estudian a la vez. En esta ocasión se analiza el error de forma objetiva según el número de clases real. Se establece que un error es tal si se mantiene durante 5 iteraciones consecutivas (se filtra el error). Los resultados obtenidos se muestran a continuación.

distC	*ξ*	*δ*	Error escena A S/F (%)	Error escena B S/F (%)	Error escena A C/F (%)	Error escena B C/F (%)
400	1	1	40	44	2,91	2,79
400	1,1	1	37	62	6,8	5,59
400	1,2	1	59	98	35,9	92,1
400	1,3	1	70	100	44,6	100
500	1	1	28	99	12,6	96,6
500	1,1	1	28	73	12,6	48
500	1,2	1	29	50	6,8	26,2
500	1,3	1	32	29	6,8	5
500	1,4	1	34	25	0,1	3,3
500	1,5	1	42	39	8,7	4,4

Tabla 5.16. Resultados de fiabilidad con los distintos parámetros. Se muestran los errores de las escenas A y B tanto teniendo en cuenta sólo los errores superiores a 5 iteraciones (C/F o con filtro), como no teniéndolos en cuenta (S/F o sin filtro).

Se observa por tanto que los errores son de naturaleza esporádica, dada la diferencia entre tener en cuenta o no errores que se mantienen en más de 5 iteraciones. El algoritmo propuesto presenta la menor tasa de errores para:

distM	300
distC	500
ξ	1,4
δ	1

Tabla 5.17. Parámetros elegidos en la segunda etapa.

A continuación, en la Figura 5.41 se muestran reconstrucciones en distintos momentos, con los parámetros elegidos:

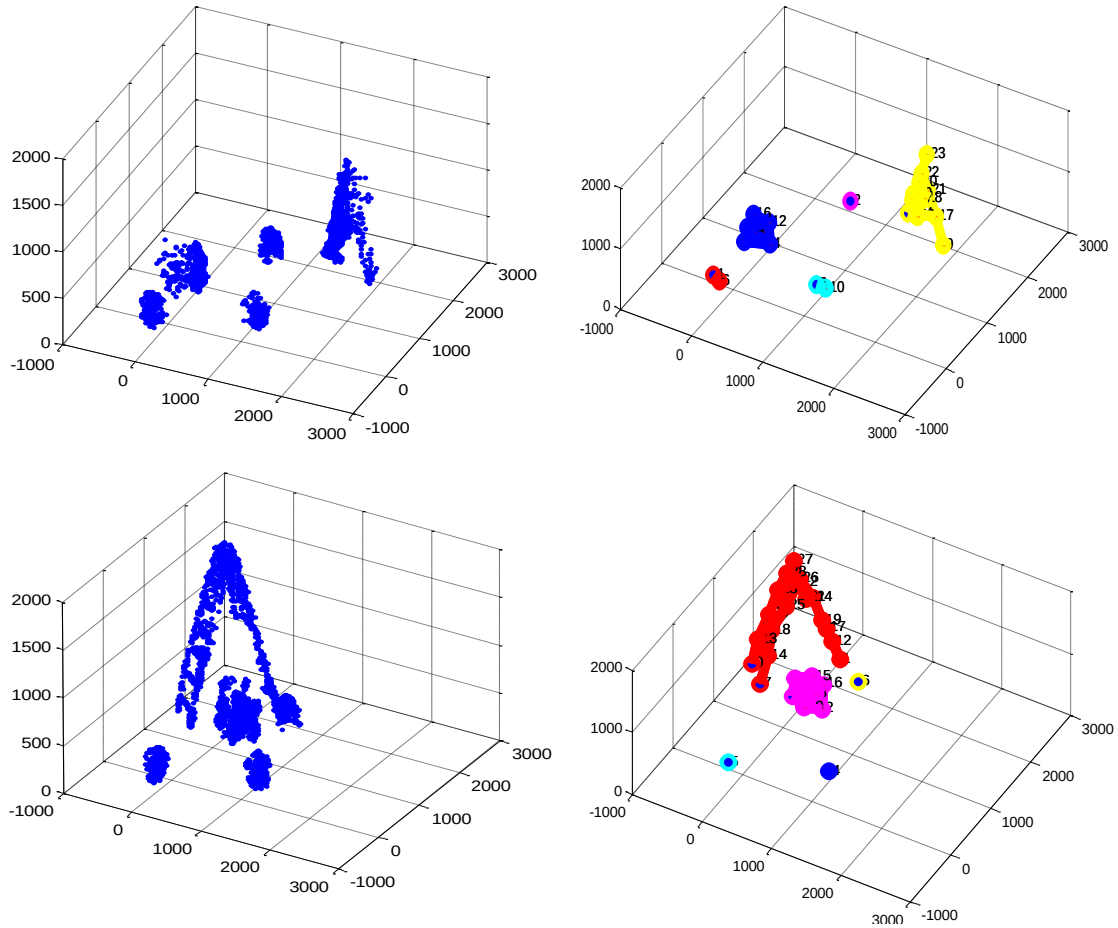


Figura 5.41. Izquierda: medidas. Derecha: clases de salida. Arriba: iteración de la escena A. Abajo: iteración de la secuencia B.

5.2.1.3. Tercera etapa

Por último, como se ha comentado anteriormente, en esta tercera etapa del ajuste de parámetros se trata de llegar a un compromiso entre tiempo de ejecución y tasa de errores. Para ello, el principal parámetro a ajustar es el número total de partículas. Se obtienen los siguientes resultados:

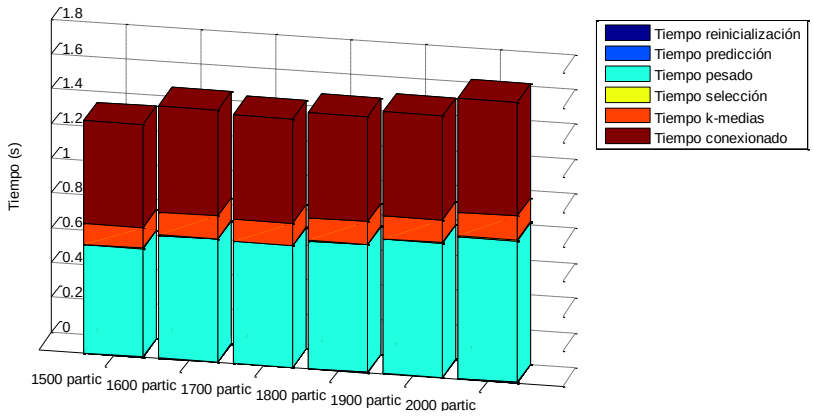


Figura 5.42. Tiempo de ejecución en función del número de partículas.

- Puede verse cómo los pasos de cálculo de pesos, agrupación por niveles mediante k-medias y algoritmo de conexionado son los que más tiempo necesitan con mucha diferencia. De hecho el resto de pasos ni siquiera pueden apreciarse en la figura.
- El tiempo necesario para calcular los pesos depende del número de partículas *n* y del de medidas, mientras que el tiempo empleado en la agrupación por alturas y conexionado depende de la distancia de agrupación *distM*, ya que a menor distancia, existen más grupos que deben converger, y más grupos sobre los que realizar el conexionado.

En la Tabla 5.18 se muestran los errores obtenidos al variar el número de partículas.

n	Error escena A S/F (%)	Error escena B S/F (%)	Error escena A C/F (%)	Error escena B C/F (%)
1500	34,95	62,36	4,85	5,59
1600	39,81	58,99	2,91	3,91
1700	31,07	55,62	2,91	5,59
1800	32,04	56,18	1,94	13,41
1900	33,98	32,58	2,91	1,68
2000	34	25	0,1	3,3

Tabla 5.18. Errores obtenidos al variar el número de partículas total

En vista a ella puede apreciarse que con 1800 partículas y menos, la tasa de error crece en gran medida. Por lo tanto el valor mínimo son 1900 partículas. De todas formas, como puede comprobarse en la Figura 5.42, no hay grandes variaciones en tiempo de ejecución.

5.3 Conclusiones

En este capítulo se han puesto a prueba los algoritmos expuestos en esta tesis. El XPFCP y el proceso de identificación han sido probados en un espacio inteligente y en tiempo real. El 3DPF, aunque actualmente está implementado en Matlab, ha sido testado con datos reales, también provenientes de un espacio inteligente. De todo ello se desprenden las siguientes conclusiones:

- El XPFCP funciona de forma robusta en todo tipo de situaciones, siendo baja su sensibilidad frente a variaciones de parámetros.

- Se cumplen las especificaciones de tiempo real, al ejecutarse satisfactoriamente a unos constantes 18 FPS.
- Los resultados obtenidos con el proceso de identificación son satisfactorios. Se obtienen porcentajes muy bajos de error en todos los experimentos realizados.
- Es posible realizar el seguimiento tridimensional de múltiples objetos de forma satisfactoria con el 3DPF propuesto.
- Éste presenta sin embargo una alta sensibilidad a las variaciones de parámetros.

CAPÍTULO 6

CONCLUSIONES Y TRABAJOS FUTUROS

Como se ha expuesto en el primer capítulo de la memoria, el objetivo de este trabajo es doble: por un lado, validar un algoritmo de seguimiento en dos dimensiones de personas, objetos, etc. en tiempo real, así como identificar un robot frente a las demás entidades presentes en la escena en este momento. Por otro lado, realizar un algoritmo que sea capaz de llevar a cabo el seguimiento tridimensional de esas mismas entidades.

En este capítulo se evalúa la consecución de los objetivos propuestos en el primer capítulo de la memoria, y se proponen posibles mejoras y caminos a partir de esta tesis.

6.1 Conclusiones

A partir del capítulo 5 puede decirse que ambos objetivos han sido alcanzados. El primero, mediante la utilización del XPFCP junto a un proceso de identificación. El segundo, mediante el 3DPF y algoritmos de conectividad. De los desarrollos aquí presentados se desprenden distintas conclusiones, que se exponen a continuación.

- En lo que se respecta al seguimiento bidimensional, la necesidad de incorporar etapas adicionales (re-inicialización y clasificación de medidas) para poder realizar el seguimiento de varias entidades de forma satisfactoria, al conseguir incluir los objetos recientemente aparecidos como hipótesis. Se comprueba que el hecho de transformar medidas en partículas no repercute negativamente en el rigor científico del seguidor,

puesto que sigue funcionando correctamente aunque sea un porcentaje muy bajo de partículas las que provengan de medidas.
- Las variaciones de parámetros no afectan de forma significativa el comportamiento del algoritmo. En los experimentos sencillos incluso se obtiene una ausencia de errores.
- Además el tiempo de ejecución es muy estable, cumpliendo las especificaciones de tiempo real.
- En lo que respecta al proceso de identificación, se comprueba que existe una diferencia relativamente grande comparando la distancia estimada y medida del robot con respecto a la del resto de clases, incluso si una persona trata de imitar el movimiento del robot.

- En cuanto al seguimiento tridimensional, se comprueba la gran sensibilidad del comportamiento del filtro frente a variaciones de parámetros.
- Estos permiten (sobre todo *distM*), por otro lado, realizar una reconstrucción morfológica más o menos exacta, o por el contrario simplemente detectar el número de objetos presente en la escena, sin dar demasiada información sobre su forma. Decantarse por uno u otro depende de la aplicación.
- Se observa además cómo la calidad del sistema de observación juega un papel importante en el buen funcionamiento del filtro. Por ese motivo, y así hacer más robusto al filtro, ha sido necesario incluir etapas de tratamiento de las observaciones, que mejoran los resultados obtenidos.

6.2 Trabajos futuros

a) Mejorar el algoritmo de segmentación de imágenes

Un problema inherente al algoritmo de segmentación utilizado como parte del algoritmo de observación (ver capítulo 2) son los cambios de iluminación y del entorno. Esto se puede solucionar utilizando algoritmos adaptativos de segmentación (como [Finlayson02]).

b) Realizar la identificación de más de un robot

En el caso de seguimiento bidimensional, la inclusión de más de un robot en el proceso de identificación es una mejora interesante, ya que abre las posibilidades de aplicación del sistema, pudiéndose realizar, por ejemplo, seguimiento en convoy de varios robots a una persona, o comportamientos organizados de los robots, ya que la salida del filtro de partículas actúa como sistema de posicionamiento global.

c) Fusionar la información visual con alguna otra existente en el espacio inteligente

Para la mejora del seguidor se puede considerar fusionarlo con otro que realice su tarea, por ejemplo, mediante sonido, a partir de arrays de micrófonos situados en el entorno. Así se podría identificar por ejemplo qué clase detectada es una persona, si ésta habla.

d) Añadir procesos de análisis de la información obtenida

Como líneas de desarrollo a añadir al seguidor tridimensional fundamentalmente aplicado a personas, pueden incluirse la identificación de personas mediante algoritmos de detección de caras como Viola-Jones ([Viola-Jones02]), el análisis de comportamientos o la interactuación con el entorno mediante lenguaje corporal, entre otros.

e) Mejorar el algoritmo de conectividad

Por otra parte, debido a la naturaleza del algoritmo de conectividad implementado, algo rudimentario, se pierde la información relativa a la continuidad de los objetos en la escena. Se pueden ensayar algunas alternativas como la caracterización mediante esqueletos a extraer de las imágenes, o algoritmos más desarrollados como MRFs (Markov Random Fields, [Khan05]) que encajan fácilmente en el entorno probabilístico de estimación dado por el filtro de partículas.

CAPÍTULO 7

MANUAL DE USUARIO

7.1. Aplicación en tiempo real de seguimiento bidimensional

En la aplicación en tiempo real están implementados tanto el XPFCP como el proceso de identificación. Esta aplicación consta de 60 archivos fuente en la parte del cliente y de 15 en la del servidor, por lo que en el siguiente apartado se incluye una breve descripción de dichos archivos, antes de explicar las partes más importantes del programa y del interfaz de usuario.

7.1.1. Estructura del sistema

El sistema utilizado para las pruebas está estructurado con el concepto servidor-cliente. Existe concretamente un cliente y cinco servidores: uno por cámara (hay cuatro), y un robot. El número de servidores asociados a cámaras es fácilmente ampliable, puesto que la diferencia entre dos servidores radica únicamente en su dirección de red y en las matrices de parámetros de la cámara asociada. Como ya se ha mencionado en capítulos anteriores, los servidores se encargan de recoger, segmentar las imágenes, y referenciar estas imágenes segmentadas en un plano común. El cliente recoge esta información para componer la rejilla de Visual Hull final, que sirve como observación al sistema de seguimiento correspondiente. El cliente además se encarga, entre otras tareas, de gestionar la comunicación y sincronización entre las cámaras y el robot y joystick, que se utiliza para controlar al robot de una forma cómoda.

Esto puede verse esquemáticamente en la Figura 7.1.

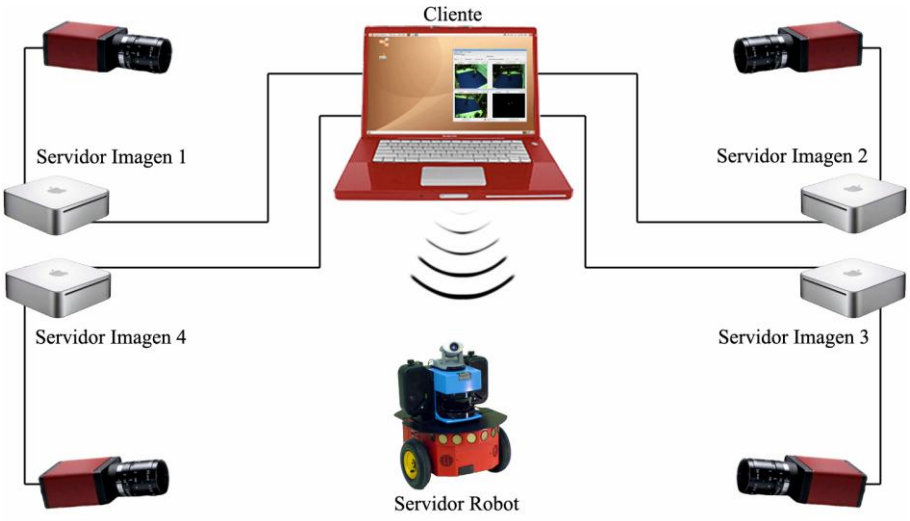


Figura 7.1. Estructura del sistema completo.

Para llevar a cabo la comunicación entre cliente y servidores se utilizan sockets distintos, ya se envíen o reciban imágenes, datos u odometría, como muestra la Figura 7.2.

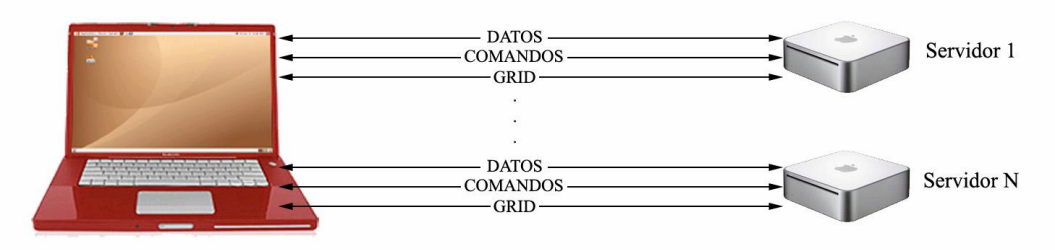


Figura 7.2. Esquema del protocolo de comunicación entre cliente y servidores.

Además, todo el intercambio de información se lleva a cabo en paralelo, gracias a la utilización de hilos. Todo este proceso se describe en más detalle en [Espejo08].

7.1.2. Archivos fuente

En esta sección se repasa la funcionalidad de cada uno de los archivos fuente que constituyen tanto la aplicación cliente como la del servidor, de cara a facilitar su comprensión y posible modificación.

7.1.2.1. Aplicación cliente

Está formada por 60 archivos fuente distintos, que se pasan a describir a continuación. Los archivos .c son los siguientes, organizados por funcionalidad:

a) General

- **Main.c** – Contiene la función main() del programa. Se encarga de inicializar el motor GTK (interfaz de usuario) y mantenerse a la espera de interrupciones software (callbacks), además de llamar a la función de inicialización de las variables y estado del sistema.
- **Inicializacion.c** – Realiza las tareas de inicialización necesarias relativas al interfaz de usuario, hilos, temporizadores, odometría, joystick, robot, servidores, filtro de partículas y semáforos.

- **Timeouts.c** – En él aparecen funciones que se ejecutan periódicamente, sobre todo de re-dibujo.
- **Utils.c** – Conjunto de pequeñas funciones de distinta naturaleza.
- **Registro_eventos.c –** Función para generar un log de eventos.
- **Ficheros.c** - Funciones relativas a la apertura y el guardado de ficheros.

b) Interfaz de usuario

- **Interface.c** – Este archivo es generado mediante el traductor de Glade a código C. Glade es el lenguaje utilizado para realizar la interfaz gráfica (GUI o Graphic User Interface).
- **Modifica_interfaz.c** – Contiene una serie de funciones que se encargan de completar la interfaz gráfica y obtener información de la misma.
- **Edicion_servidores.c** – Similar al anterior, pero relativo a la ventana de edición de servidores que aparece al seleccionar Edición > Servidores en el menú desplegable (esto se verá posteriormente).
- **Odometria.c** – Contiene funciones que modifican las partes del interfaz gráfico relativas a la odometría.
- **Support.c** – Funciones GTK adicionales para la interfaz gráfica.
- **Color.c** – Cambia el color a través de uno de los elementos que constituye la interfaz gráfica.
- **Callbacks.c** – También generado por el traductor de Glade a C, en él están contenidas las definiciones de las distintas callbacks, que son llamadas al interactuar con el interfaz gráfico.

c) Red

- **Hilos.c** – Contiene las funciones asociadas a los hilos que se ejecutan de forma paralela en la aplicación: uno asociado al robot, otro a la sincronización entre robot y cámara, otro al sondeo del joystick y un último que se encarga de recibir las imágenes desde los servidores. En [Espejo08] puede verse con más detalle la estructura de los mismos.
- **Captura.c** – Funciones relativas a capturar las imágenes de los servidores.
- **Contorno.c** – Funciones para recibir y componer el grid final.
- **Estados.c** – Funciones que se encargan de controlar el estado de los servidores.
- **Red.c** – Contiene las funciones que se encargan de realizar tareas de red y de sincronización entre el robot y servidores.
- **Grabacion.c** – Contiene las funciones para envíar las órdenes de comenzar y parar la grabación de imágenes en los servidores.
- **Localiza.c** – Función que se encarga de recibir la información relativa a las cámaras disponibles en ese instante.

d) Servidor

- **F_particulas.c** – Se encarga periódicamente de obtener las medidas a partir de la imagen del grid, y de llamar a la función del XPFCP y proceso de identificación.
- **Joystick.c** – Funciones para la utilización y configuración del joystick para controlar el robot en el espacio inteligente.
- **MiXpfcp.c** – Este archivo fuente contiene tanto la definición de la función que implementa el seguidor bidimensional utilizado, el XPFCP, como la del proceso de identificación.

- **MisMath.c** – En él aparecen funciones matemáticas relacionadas con el manejo de matrices, algoritmos de selección de partículas, y algunas funciones relativas a la modificación de las distintas imágenes para representar trayectorias, clases, textos, etc.
- **MiCluster.c** – Funciones que implementan procesos de clasificación.
- **Robot.c –** Contiene funciones de bajo nivel relacionadas con el robot: conexión y desconexión del mismo, modificación de su velocidad y recepción de la odometría.
- **Robot_alto_nivel.c** – Contiene funciones de alto nivel relacionadas con el robot. Como en robot.c, hay funciones para conectar, desconectar y modificar la velocidad, pero programadas a más alto nivel, a partir de las contenidas en robot.c. Además incluye una función para dibujar la odometría en la pestaña "Robot" del interfaz de usuario.

En cuanto a los ficheros cabecera, el principal es **ispace.h** ya que se encarga de insertar el resto, además de las librerías necesarias para el funcionamiento del programa.

Los siguientes archivos de cabecera contienen las definiciones de constantes asociadas a su .c equivalente:

- **MisMath.h**
- **MiXpfcp.h**
- **Robocom.h**

Estos otros contienen las declaraciones de las funciones que aparecen en sus .c equivalentes:

- **Localiza.h**
- **Modifica_interfaz.h**
- **Odometria.h**
- **Red.h**
- **Registro_eventos.h**
- **Robot.h**
- **Robot_alto_nivel.h**
- **Support.h**
- **Timeouts.h**
- **Callbacks.h**
- **Captura.h**
- **Color.h**
- **Joystick.h**
- **Interface.h**
- **Inicializacion.h**
- **Hilos.h**
- **Grabacion.h**
- **Ficheros.h**
- **F_particulas.h**
- **Estados.h**
- **Edicion_servidores.h**
- **Contorno.h**

Por otra parte, en:

- **Constantes.h** – Se definen constantes generales.
- **Variables.h** – Se definen las variables globales.

Por último, en **estructuras.h** se definen todos los tipos estructurados de datos que se utilizan en el programa. A continuación se describen los más relevantes para los algoritmos presentados en este trabajo.

La estructura de tipo *XpfMeas* define las medidas. La constante *NVISION* es el número máximo de medidas que se pueden obtener, y *NMEAS2D* es la dimensión de esas medidas, en este caso 2: X y Z. En el campo *ny* se guarda el número de medidas obtenido.

```
typedef struct
{
         float y[NVISION*NMEAS2D];
         int ny;
}XpfMeas;
```

Las estructuras de tipo XpfOrient se utilizan para guardar los últimos *TAMBUFFER* valores distintos parámetros relacionados con las clases y su orientación. En *xmeas* se guarda la posición del centroide de la clase, obtenida mediante el algoritmo de seguimiento bidimensional. En *xest* se guarda la trayectoria estimada mediante la velocidad angular (*vang*) y lineal (*vlineal*) calculadas mediante la odometría del robot. En *J* se guarda el Jacobiano asociado al movimiento del robot.

```
typedef struct
{
       float xmeas[NPROCESSLOC*TAMBUFFER];
       float xest[NPROCESSLOC*TAMBUFFER];
       float vang[TAMBUFFER];
       float vlineal[TAMBUFFER];
       float J[NPROCESSLOC*NPROCESSOD*TAMBUFFER];
}XpfOrient;
```

La estructura asociada a cada clase de medidas y de partículas del algoritmo de seguimiento bidimensional es de tipo XpfCluster. La descripción de los distintos campos es la siguiente:

- **Identify:** Identificador de la clase
- **Candidate:** Número de iteraciones necesarias para validar o invalidar la clase.
- **Validated:** Bandera de clase válida (1 = válido, 0 = inválido).
- **nMembers:** Número de elementos asociados a la clase. Serán medidas o partículas en función de la llamada al campo de la estructura.
- **Probability:** Grado de certeza de que la clase existe.
- **Centroid:** Coordenadas del centroide de la clase.
- **Members:** Coordenadas de cada uno de los elementos asociados a la clase.

```
typedef struct
{      int identify;
       int candidate;
       int validated;
       int nMembers;
       float probability;
       float centroid[NPROCESS3D];
       float members[NPARTICMAX*NPROCESS3D];
}XpfCluster;
```

Estructura de odometría. La velocidad lineal y angular se guardan en *linspeed* y *rotspeed*, respectivamente. La posición relativa en x, y y orientación se guardan en *x*, *y* y *phi*, respectivamente. Como se ha explicado en el capítulo 3, esta orientación no es la proporcionada por la odometría desde el primer instante de tiempo, sino la actualizada mediante el XPFCP. El campo *num_orden* se corresponde con el número del frame en el que se graba la odometría en ese instante de tiempo.

```
typedef struct
{
       double linspeed;
       double rotspeed;
       double x;
       double y;
       double phi;
       double num_orden;
}odometry;
```

7.1.2.2. Aplicación servidor

La aplicación del servidor está formada por 15 archivos distintos: 9 de código fuente y 6 archivos de cabecera.

En cuanto a código fuente:

a) General

- **Main.c –** Abre los sockets y se queda esperando conexiones. Es el fichero que contiene la función *main.*
- **Menu.c** – Función para crear ciertos archivos y actuar frente a distintos comandos recibidos por los sockets.
- **Support.c –** Funciones varias generadas con Glade.
- **Utils.c** – Funciones de diversa naturaleza.

b) Red

- **Localiza.c** – Función para localizar las cámaras disponibles y enviar al cliente esa información.
- **Red.c –** Funciones relacionadas con la comunicación con el cliente.

c) Tratamiento de imágenes

- **Captura.c –** Contiene la función de captura de imágenes.
- **Contorno.c –** Incluye diversas funciones relacionadas con la segmentación y proyección de la imagen al plano común.

En cuanto a archivos de cabecera, estos contienen las declaraciones de las funciones que aparecen en sus .c equivalentes:

- **Callbacks.h**
- **Contorno.h**

- **Interface.h**
- **Support.h**

El fichero **ispace.h** contiene tanto declaración de funciones como de constantes generales para todos los ficheros. Por último, en **tablas.h** se almacenan valores pre-calculados de la función logaritmo (utilizada para en la función de actualización de la media y desviación típica en imágenes), mejorando así el tiempo de ejecución del proceso de los servidores.

7.1.3. Utilización de la aplicación.

Una vez que el cliente y los distintos servidores, robot incluido, están encendidos, el procedimiento a seguir para arrancar la aplicación completa se detalla a continuación.

7.1.3.1. Conexión de las cámaras

Las direcciones IP de los servidores asociados a las cámaras en el espacio inteligente utilizado en las pruebas son:

- ispace0 (172.29.24.216)
- ispace1 (172.29.24.217)
- ispace2 (172.29.24.217)
- ispace3 (172.29.25.217)

El proceso se facilita si se han almacenado previamente en /etc/hosts los nombres asociados a cada IP, ya que así, por ejemplo, basta con teclear *ispace0* en lugar de *172.29.24.216*.

1. Para poner a cada servidor a la espera de conexiones, tecleamos lo siguiente en consola, para acceder a los servidores por ssh iniciando una instancia del interfaz gráfico de los mismos:

```
ssh –X servidor@ispaceN
```

Con N=0, 1, 2 ó 3.

2. A continuación hay que introducir la contraseña correspondiente, que es *servidor* en todos los casos.
3. Para ejecutar el servidor hay que acceder al directorio correspondiente y ejecutar la aplicación (*gladeservi0)*:

```
cd /home/servidor/TesisMasterAlvaroMarcos/

./src/gladeservi0
```

El siguiente mensaje aparecerá por pantalla si todo va bien:

```
Socket creado
Socket enlazado
Socket creado
Socket enlazado
Socket creado
Socket enlazado
```

Es necesario repetir los tres pasos anteriores para cada servidor.

El número mínimo de cámaras para obtener una rejilla de ocupación con calidad suficiente como para hacer el seguimiento bidimensional es 3, aunque se recomienda utilizar las 4 para reducir en mayor medida las zonas de la escena no vistas por ninguna cámara [Pizarro08].

7.1.3.2. Conexión del robot

El siguiente paso para arrancar la aplicación principal es conectar el robot.

Está conectado mediante red inalámbrica al router del espacio inteligente, por lo que la forma de acceder es la misma que en el caso de los servidores asociados a las cámaras.

Se supone que su IP (172.29.26.217, en el caso de utilizar el del espacio inteligente), ya está asociada a *robot1*.

```
ssh –X robot@robot1
```

La contraseña es *robot*. Ahora existen dos posibilidades para lanzar la aplicación correspondiente: la primera, posible si se utiliza un único robot, es acceder a la aplicación *robot_server* del siguiente directorio:

```
/home/robot1/seigyo/
```

La segunda posibilidad, en la que la aplicación servidor está programada en Player (en el caso anterior lo estaba en Aria), es acceder a *robot_server* del siguiente directorio:

```
/home/robot1/seigyo_player/
```

En el caso de utilizar un único robot, el uso de una u otro es indistinto. Sin embargo, la segunda opción posibilita la utilización de más de uno.

7.1.3.3. Conexión del joystick

Para conectar el joystick simplemente hay que enchufarlo a un puerto USB del ordenador cliente. Si no funciona correctamente así, es necesario comprobar que la dirección es la correcta. Para ello, se busca la dirección asociada al joystick que aparece en el direcorio */dev/* una vez enchufado éste, y se introduce esta dirección en la ventana de configuración del joystick, que aparece al hacer click en Edición > Joystick como se muestra en la Figura 7.3

Figura 7.3. Ventana de edición de los parámetros del joystick.

7.1.3.4. Aplicación principal

La aplicación cliente se arranca escribiendo la siguiente línea en el directorio correspondiente del ordenador cliente:

```
./src/proyecto2
```

Aparecerá la siguiente ventana:

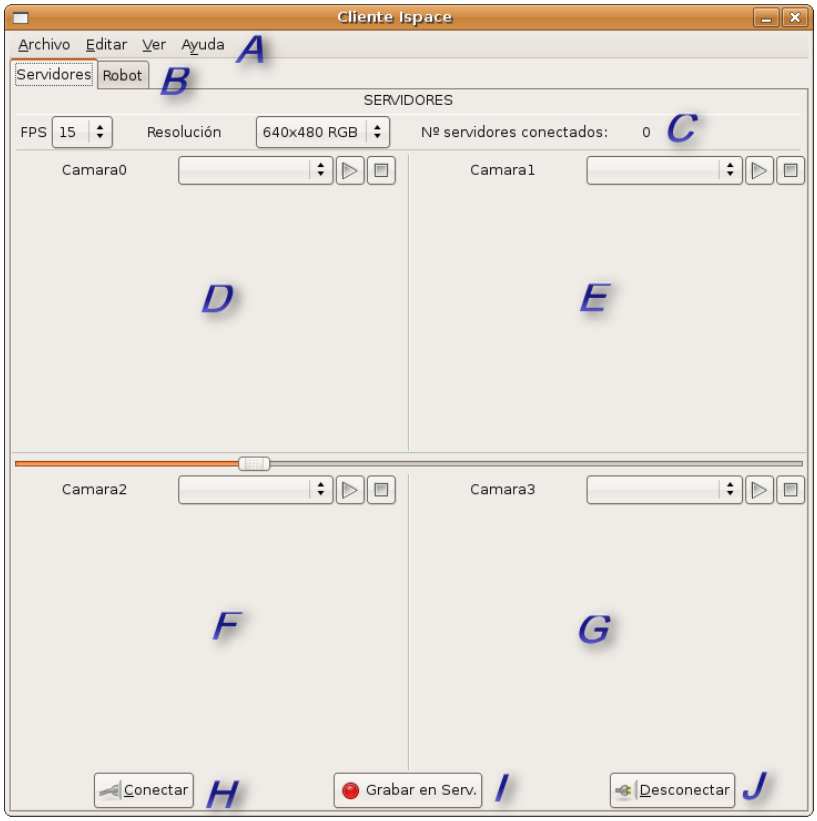


Figura 7.4. Ventana principal de la aplicación.

Los elementos que la componen con sus opciones, se enumeran a continuación:

- **A:** Barra de menús.
  - Archivo
    - Guardar odometría
    - Salir
  - Editar
    - Servidores
    - Joystick
    - Robot
    - Odometría
      - Colores
      - Propiedades odometría
      - Cargar fondo odometría
  - Ver (Versión)

- Ayuda

- **B:** Pestañas principales, para pasar de la de análisis de imágenes de servidores a la configuración del robot.
- **C:** Configuración del modo de captura de los servidores.
- **D, E, F, G:** Ventanas que muestran las imágenes recibidas por las distintas cámaras o la rejilla de ocupación (llamada GRID), a elegir mediante los menús desplegables situados en la parte superior de cada una, y pulsando el botón de reproducir.
- **H:** Botón para conectarse con los servidores.
- **I:** Botón para grabar imágenes capturadas y segmentadas en los servidores.
- **J:** Botón para desconectarse de los servidores.

Al pulsar la pestaña "Robot", se muestra lo que aparece en la Figura 7.5:

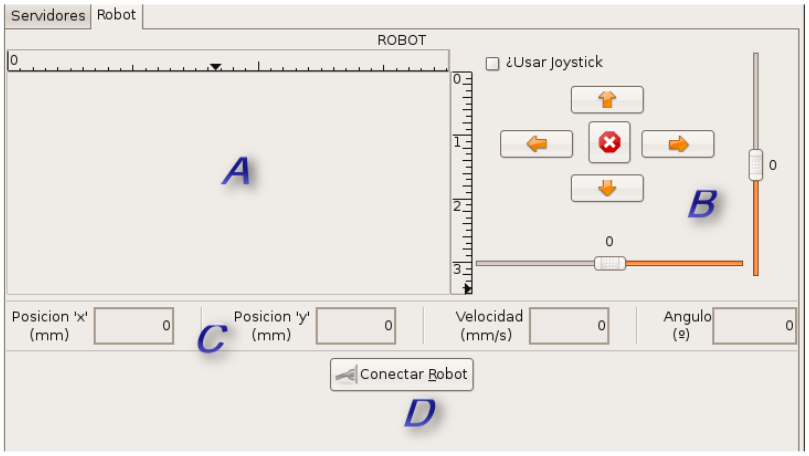


Figura 7.5. Pestaña relativa al robot.

Los elementos que la componen son los siguientes:

- **A:** Representación de la odometría acumulada (también conocida como "dead reckoning" sobre la imagen que se cargue (es posible elegir una imagen en formato bitmap que un mapa del entorno, para representar la odometría sobre él).
- **B:** Flechas para mover el robot desde la interfaz de usuario. Es posible elegir si se desea utilizar además el joystick o no.
- **C:** Información obtenida con la odometría.
- **D:** Botón de conexión al robot.

Los pasos necesarios para conectar el cliente y los servidores se muestran a continuación. Primero es necesario configurarlos. En Edición > Servidores se especifica las direcciones de los servidores, como se muestra en la Figura 7.6:

Figura 7.6. Ventana de edición de servidores.

Es posible añadir direcciones personalizadas y guardar esa configuración como predeterminada, así como guardarla en un archivo para utilizarla posteriormente. Para completar la configuración de los servidores, se elegirá la velocidad de captura en FPS (Frames Per Second o Cuadros Por Segundo) y el modo (tamaño y espacio de color, **C** de la Figura 7.4). Por último, se pulsa el botón "Conectar".

Si todo ha ido sin contratiempos aparecerá un mensaje de confirmación en una ventana emergente, así como una contestación por parte de los servidores en la consola de estos, como se muestra en la Figura 7.7, en este caso una vez que se han conectado 3 servidores:

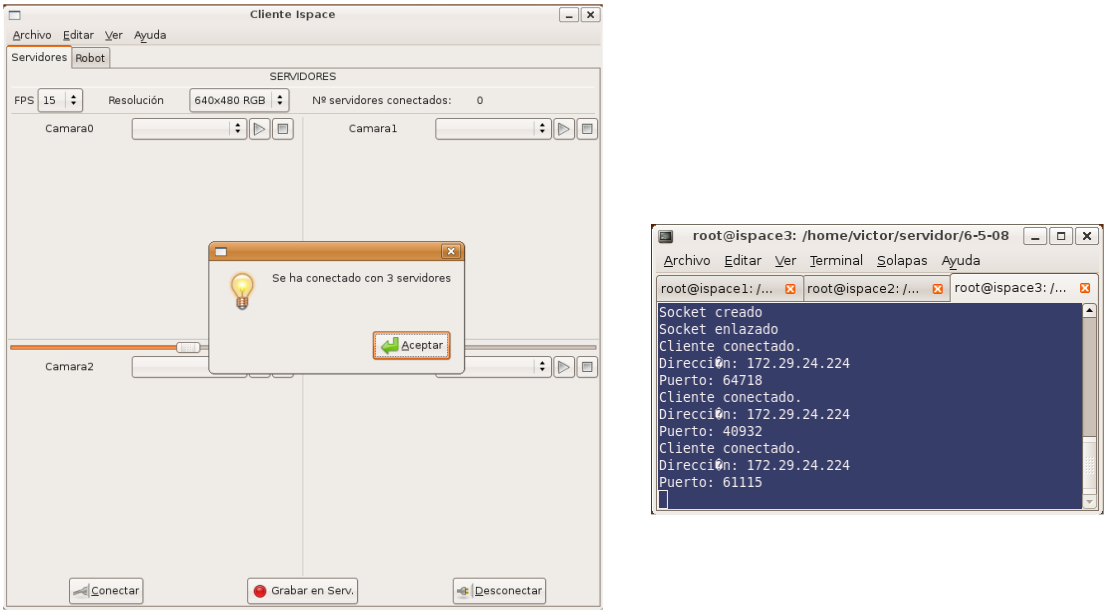


Figura 7.7. Izquierda: mensaje de confirmación al conectarse a los servidores, 3 en este caso. Derecha: mensajes de confirmación proporcionados por uno de los servidores.

Por otra parte, para configurar y conectar el robot al cliente, es necesario ir primero a Edición > Robot, con lo que aparece una ventana emergente como la que se muestra en la Figura 7.8, que permite configurar la dirección del robot, el paso de las flechas de movimiento

de la Figura 7.5 y las máximas velocidades, tanto lineal como angular, con las que se desea que se mueva el robot:

Figura 7.8. Ventana de propiedades del robot.

Pulsando el botón "Conectar Robot" éste quedará conectado mediante el protocolo correspondiente que se haya arrancado en el servidor del robot (Player o Aria). A partir de este momento se recibe continuamente odometría de forma sincronizada a las cámaras.

Una vez hecha la configuración mediante el procedimiento ya descrito, el aspecto de la aplicación es el que se muestra en la Figura 7.9:

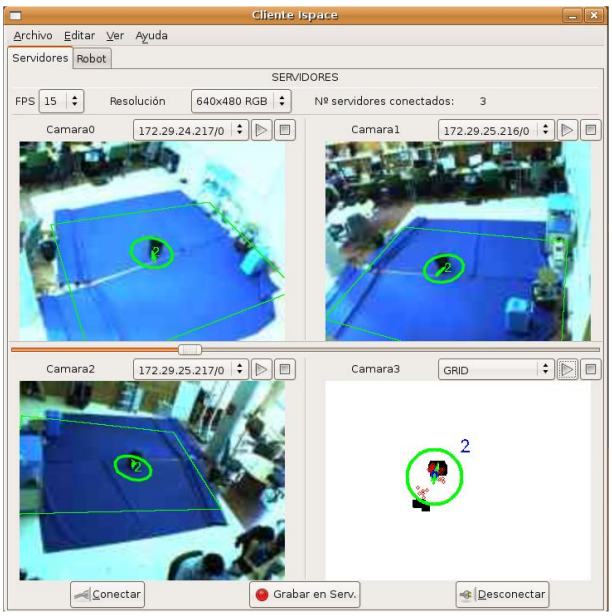


Figura 7.9. Aspecto de la aplicación ejecutándose, una vez conectadas las cámaras y visualizando el grid.

Las distintas clases de salida (es decir, de partículas) que genera el XPFPC se etiquetan mediante círculos de distintos colores y el identificador de clase asociado, tanto en la vista bidimensional como en las imágenes capturadas desde las cámaras. En las últimas además se dibuja el límite de la rejilla de ocupación, mediante un rectángulo verde.

En la representación del GRID, como se muestra en la Figura 7.10, aparecen las medidas obtenidas a través del Visual Hull en negro sobre fondo blanco. Las partículas se representan mediante pequeños círculos rojos. El centroide del grupo correspondiente se representa mediante un círculo azul, y el vector velocidad de la clase mediante una línea verde. Tanto en la representación del GRID como en la de las imágenes obtenidas por las cámaras se dibuja la trayectoria de cada clase a lo largo de los últimos *TAMBUFFER* instantes, en el mismo color que el círculo asociado. Se dibuja en gris la trayectoria estimada a partir de la odometría. En la clase cuya distancia entre trayectoria real y estimada se dibuja la palabra "ROBOT".

Todo esto puede verse en la Figura 7.10:

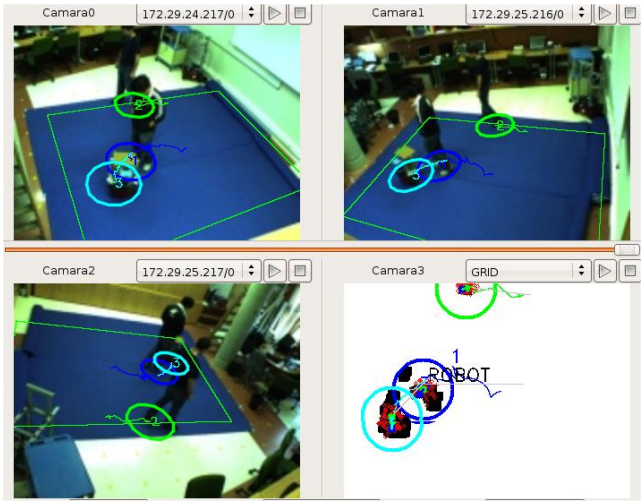


Figura 7.10. Aspecto de la aplicación cuando se está realizando el seguimiento, mostrando tres cámaras y el GRID.

7.2. Seguimiento tridimensional

Los algoritmos relativos al seguimiento tridimensional están implementados en el entorno de simulación Matlab. A continuación se hace un resumen de la estructura del sistema, los distintos ficheros fuente que forman parte del mismo, y de los resultados que se generan en tiempo de ejecución.

7.2.1. Estructura del sistema

El sistema de seguimiento está completamente simulado. Como entrada recibe una serie de medidas tridimensionales obtenidas offline aplicando el algoritmo de Visual Hull visto en el capítulo 2, a partir de imágenes ya segmentadas. A partir de esta entrada, la estructura del sistema se muestra de forma esquemática en la Figura 7.11:

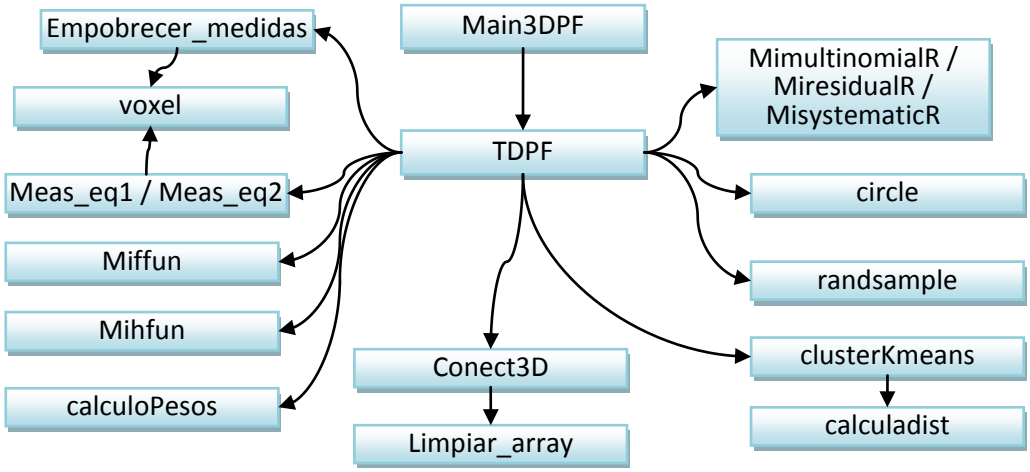


Figura 7.11. Relación entre distintos ficheros del sistema.

7.2.2. Archivos fuente

El algoritmo está implementado en Matlab mediante los siguientes ficheros:

a) General

- **Main3DPF.m:** Fichero principal, que se encarga de llamar a todos los demás, que se debe ejecutar desde la línea de comandos de Matlab, con la siguiente sintaxis:

```
main3DPF('C:\directorio_datos_vh', plotOn, videoOn, saveOn);
```

  Donde *plotOn* determina si se muestran figuras de ciertos resultados intermedios, *videoOn* se utiliza para grabar o no un vídeo con algunos de estos resultados, y *saveOn* para guardar o no los resultados de distintas variables del sistema en un fichero .mat que permita analizar el funcionamiento del experimento a posteriori en Matlab. Los datos de las observaciones son recogidos del directorio *C:\directorio_datos_vh*.
- **circle.m:** Función para dibujar círculos bidimensionales.
- **limpiar_array.m:** Ordena de menor a mayor y elimina elementos repetidos en un array.
- **voxel.m:** Función para representar prismas tridimensionales.
- **empobrecer_medidas.m:** Función que empobrece de forma artificial el grid 3D obtenido en relación a un objeto a determinar del video de pruebas, para comprobar la robustez del filtro frente al sensado desigual de objetos.

b) Funciones relacionadas con el algoritmo de seguimiento

- **TDPF.m:** Es llamado por la función *main3DPF* después de cargar las observaciones. Contiene el código del algoritmo de seguimiento 3DPF expuesto en el capítulo 4.
- **calculoPesos.m:** Contiene el algoritmo utilizado en el paso de corrección del filtro de partículas para asignar los pesos a las partículas.
- **clusterKMeans.m:** Algoritmo de clasificación k-medias para un número variable de dimensiones.
- **conect3d.m:** Algoritmo de conectividad para aplicar sobre los centroides obtenidos a partir de la clasificación de partículas por niveles.
- **meas_eq1.m:** Algoritmo de ecualización de medidas basado en histograma (ver 4.1.1).
- **meas_eq2.m:** Algoritmo de ecualización de medidas basado en grupos (ver 4.1.1).
- **MimultinomialR.m:** Algoritmo de selección multinomial, mencionado en el capítulo 4.
- **MiresidualR.m:** Algoritmo de selección residual, mencionado en el capítulo 4.
- **MisystematicR.m:** Algoritmo de selección sistemático, mencionado en el capítulo 4.

c) Funciones matemáticas

- **calculaDist.m:** Utilidad para calcular la distancia euclidea en el algoritmo de clasificación k-medias.
- **Miffun.m:** Función para procesar el modelo de actuación propuesto.
- **Mihfun.m:** Función para procesar el modelo de observación propuesto.
- **randsample.m:** Función para obtener un conjunto aleatorio de muestras de una matriz.

Una vez lanzada la aplicación, aparecerán por pantalla una serie de figuras. Aparecen figuras adicionales si el parámetro *plotOn* está activado. Las básicas son las siguientes, que se muestran en la Figura 7.12 y cuya funcionalidad se describe al pie de ésta:

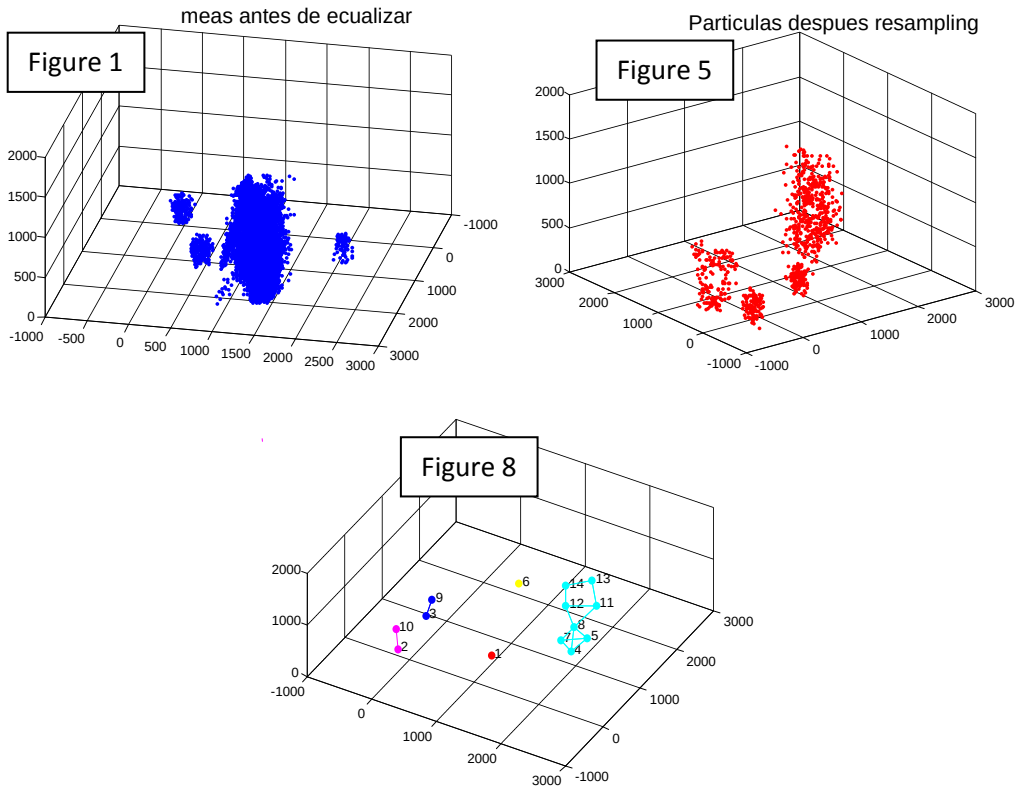


Figura 7.12. Figure 1: Representación mediante puntos azules de las medidas tridimensionales obtenidas directamente del fichero de entrada a main3DPF. Figure 5: Distribución de partículas en el entorno tridimensional después de realizar el paso de selección del 3DPF. Figure 8: Salida final del algoritmo 3DPF, clases finales. Cada una aparece representada en un color distinto, mediante un esqueleto que conecta centroides cercanos, como se ha explicado en el capítulo 4. Existe una clase asociada a la persona (cyan), una para el robot (azul), y una para cada papelera (magenta, amarillo y rojo). Todas las figuras se corresponden con la misma iteración.

Si se activa el flag de *plotOn* en la llamada a main3DPF, se dispone de una serie de figuras extra correspondientes a pasos intermedios del proceso. Estas figuras se muestran en la Figura 7.13**Error! Reference source not found.**, suponiendo que se está utilizando el algoritmo de ecualización de medidas basado en grupos.

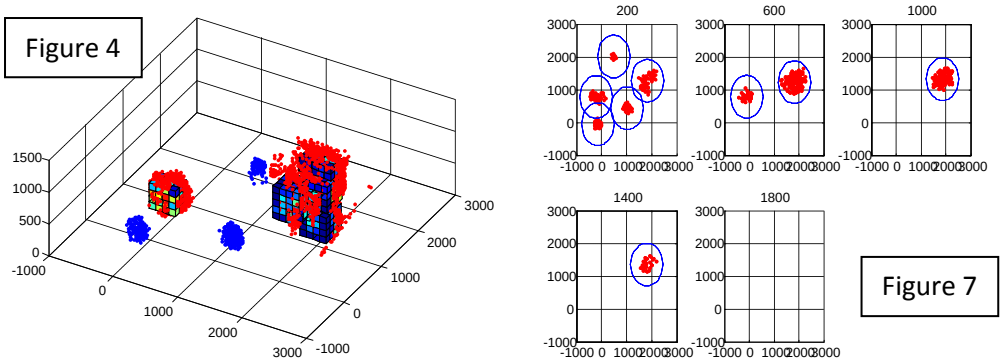


Figura 7.13. Figure 4: Mediante puntos azules se representan los grupos de medidas en los que no existen medidas redundantes. Con puntos rojos los grupos en que sí. Superpuestos a los grupos con medidas redundantes se representa mediante prismas la discretización del volumen en escala de colores según la densidad de medidas en esa parte del volumen (ver capítulo 4). Figure 7: Partículas por alturas (cada altura se muestra en milímetros como título de cada subplot) y clasificación por alturas de las mismas con el algoritmo k-medias. Como se explicó en el capítulo 4, esto genera unos centroides (representados mediante circunferencias azules en la figura) sobre los que se aplica el algoritmo de conectividad, dando lugar a la Figure 7 ya vista en la Figura 7.12.

Si se utiliza el algoritmo de ecualización de medidas basado en histograma, aparecerán los gráficos que se muestran en la Figura 7.14 en lugar de la "Figure 4" de la Figura 7.13, que se muestran a continuación:

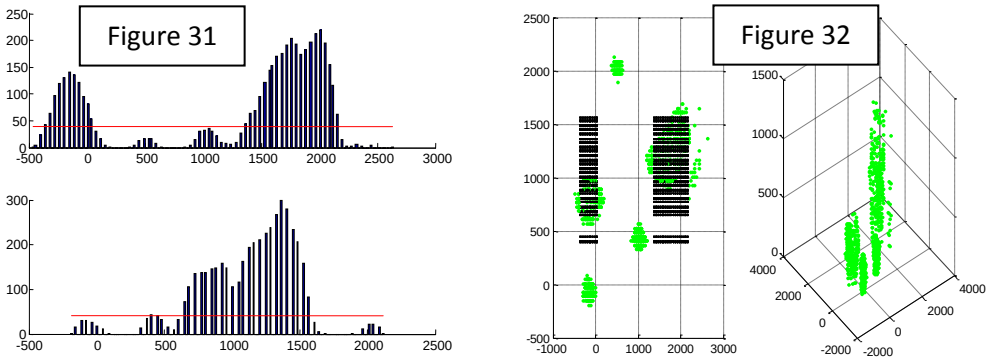


Figura 7.14. Figure 31: Histogramas de medidas en la dimensión X (arriba) y Z (abajo). El umbral a partir del cual se considera que la densidad de medidas es redundante se representa mediante una línea roja horizontal. Figure 32: A la izquierda se muestran las regiones de medidas sobre las que se ha realizado la ecualización, representadas mediante pequeños cuadrados. A la derecha, una vista tridimensional de las medidas ya ecualizadas.

CAPÍTULO 8

PLIEGO DE CONDICIONES

En el funcionamiento del sistema final se ven implicados tanto recursos hardware como software. En su última versión se emplean los numerados a continuación:

8.1. Hardware

- Router gigabit ethernet D-link.
- Cuatro ordenadores de sobremesa equipados con puerto firewire y gigabit ethernet.
- Portátil EasyNote SL65-M-003SP con procesador Intel Core2 Duo, 500 GB de disco duro y 4 GB de RAM.
- Punto de acceso inalámbrico.
- Cuatro cámaras Sony Marlin V392 IEEE1394.
- Robot Pioneer P3-DX.
- Mando analógico de Playstation y adaptador a PC.

8.2. Software

- Sistema operativo Ubuntu en sus versiones 5.04 y 7.10.
- Librerías para control del interfaz IEEE1394.
- Librerías OpenCV.
- Entorno de simulación Matlab

CAPITULO 9

PRESUPUESTO

En esta sección se estima el importe de la ejecución del proyecto. Para ello se realiza un estudio agrupando los gastos en función de su origen.

9.1. Costes de ejecución material

Los costes de ejecución incluyen tres elementos:

- Costes de equipos.
- Costes del software.
- Costes de mano de obra por tiempo empleado.

9.1.1. Costes de equipos

CONCEPTO	PRECIO UNITARIO	CANTIDAD	SUBTOTAL
Ordenador portátil	650 €	1	650 €
Ordenador MacMini	600 €	4	2.400 €
Cámara Marlin	800 €	4	3.200 €
Router gigabit	160 €	1	160 €
Punto de acceso	50 €	1	50 €
Robot Pioneer P3-DX	6.000 €	1	6.000 €
Impresora HP LaserJet	70 €	1	70€
Subtotal			12.530 €

Tabla 9.1. Costes de equipos.

9.1.2. Costes de software

Los distintos equipos que han sido utilizados en este trabajo son los siguientes:

CONCEPTO	PRECIO UNITARIO	CANTIDAD	SUBTOTAL
Sistema operativo Ubuntu	0 €	5	0 €
Coriander 1.0.1	0 €	1	0 €
Matlab 7.0	2.500 €	1	2.500 €
Glade	0 €	1	0 €
Microsoft Office XP	200 €	1	200 €
Photoshop CS 3	1.000 €	1	1.000 €
Microsoft Windows XP	135 €	1	135 €
Subtotal			3.835 €

Tabla 9.2. Costes de software.

9.1.3. Costes por tiempo empleado

FUNCIÓN	PRECIO UNITARIO	Nº HORAS	SUBTOTAL
Ingeniería	25 €	1440	36.000 €
Mecanografiado	15 €	240	3.600 €
Subtotal			39.600 €

Tabla 9.3. Costes por tiempo empleado

El número de horas de ingeniería corresponde a un trabajo de seis horas diarias durante un año. El mecanografiado se ha estimado como seis horas diarias durante dos meses.

9.1.4. Coste total del presupuesto de ejecución material

CONCEPTO	SUBTOTAL
Costes de equipos	12.530 €
Costes de software	3.835 €
Costes por tiempo empleado	39.600 €
Subtotal	55.965 €

Tabla 9.4. Coste total del presupuesto de ejecución material.

9.2. Gastos generales y beneficio industrial

Se incluyen los gastos derivados de la utilización de las instalaciones de trabajo y el beneficio industrial. Se estima como el 16% del coste de ejecución material

CONCEPTO	SUBTOTAL
Gastos generales y beneficio industrial	8.954,4 €
Subtotal	8.954,4 €

Tabla 9.5. Gastos generales y beneficio industrial.

9.3. Importe total del presupuesto

CONCEPTO	SUBTOTAL
Coste total del presupuesto de ejecución material	55.965,0 €
Gastos generales y beneficio industrial	8.954,4 €
Subtotal	64.838,2 €
IVA 16%	10.374,1 €
TOTAL IVA INCLUIDO	75.283,5 €

Tabla 9.6. Importe total del presupuesto.

El importe total del proyecto suma la cantidad de:

Setenta y cinco mil doscientos ochenta y tres euros con cincuenta céntimos.

Alcalá de Henares a Julio de 2009
Fdo: Álvaro Marcos Ramiro
Máster

CAPÍTULO 10

PLANOS

En este capítulo se incluyen las partes del código de relacionadas con esta tesis: tanto los ficheros correspondientes del cliente en la aplicación en tiempo real (para el seguimiento bidimensional) como los scripts del entorno de simulación Matlab (para el seguimiento tridimensional).

10.1. Seguimiento bidimensional

10.1.1. f_particulas.c

```
E #include "ispace.h"

void miXpf(int iter,int doResult,float At,XpfMeas Meas,float *xPartic,XpfCluster *cluster,
                    XpfCluster *clusterOut,int *ks,int *cParams,int *cParamsO,int *xParams,
                    IplImage *iplXZProyColor,FILE *fResult);
void miLiveIdent(IplImage *iplXZProyColor, float At, odometry odo);

void filtro_particulas(IplImage * iplgrid,float At,odometry odo)
{
        int     a,b,k;

        // Variables para guardar grid en jpg y odometria (y asi poder simular offline)
        static FILE *file_odom;
        static int fichero_abierto=0;
        char cadena1[16]="grid-", cadena2[5]=".jpg";
        static int frame=0;
        static int guardar=0;


        if (guardar)
        {
                // Guardar Odometria
                if (!fichero_abierto)
                {
                        file_odom = fopen("odometria.txt", "w");
                        if (file_odom)
                        {
                                printf("Fichero abierto\n");
                                fichero_abierto = 1;
                        }
                        else
                                printf("El fichero no ha podido abirse\n");
                }

                if (fichero_abierto)
                        fprintf(file_odom, "frame %d - linspeed %f rotspeed %f x %f y %f phi %f\n",
                        frame, odo.linspeed, odo.rotspeed, odo.x, odo.y, odo.phi);

                // Imagen
                sprintf(&cadena1[5],"%d",frame);
                strcat(cadena1,cadena2);
                printf("Imagen a guardar: %s\n", cadena1);
                cvSaveImage(cadena1,iplgrid);
```

```
44.         }
45.
46.
47.        cvDilate(iplgrid,iplgrid,NULL,3);
48.
49.
50.         k       = 0;
51.         Meas.ny = 0;
52.
53.
54.         for( a=0; a<(iplgrid->width*3); a += 12)        //Ancho (320)
55.         {
56.                 for( b=0; b<(iplgrid->height*3); b += 12) //Alto        (240)
57.                 {
58.                         if( iplgrid->imageData[b*iplgrid->width+a] != 0 )
59.                         {
60.                                 if(k < NVISION*NMEAS3D)  //No podemos desbordar el array Meas.y
61.                                 {
62.                                         Meas.y[k]   =(float)(a*(float)4);
63.                                         Meas.y[k+1] =(float)(b*(float)4);
64.                                         Meas.ny++;
65.                                         k+=2;
66.                                 }
67.                         }
68.                 }
69.         }
70.
71.        cvThreshold( iplgrid, iplgrid,60,355,CV_THRESH_BINARY_INV);
72.
73.         cParams[1] = Meas.ny;
74.
75.         // Reinitializing the algorithm
76.         if ((ks[0]==0)&&(xParams[0]==0))  iter = INTINF;
77.         if ((iter==INTINF)&&(Meas.ny!=0)) iter = 0;
78.
79.         if(iter!=INTINF)
80.         {
81.                 // Calling now xpf
82.                 printf("At = %f\n", At);
83.                 miXpf(iter,doResult,At,Meas,xPartic,cluster,clusterOut,ks,cParams,cParamsO,xParams,iplgrid,fResult);
84.
85.                 miLiveIdent(iplgrid, At, odo);
86.
87.                 if (iter<2) iter++;
88.         }
89.
90.         frame++;
91.
92. }
```

10.1.2. miXpfcp.h

```
1. /*************************************************************************
2.  * miXpfcp.h
3.  * PURPOSE  : Defines para ejecutar en misMaths.c Todas dist en mm.
4.  * AUTHORS  : Marta Marron Romera (marta@depeca.uah.es)
5.  * DATE     : 1 May 2007 (ojo cambios rpto abril)
6.  * INNOVATIONS: for miXpfcp + miCluster + misMaths
7.  *************************************************************************/
8.
9. #if !defined( _MIXPFCP_H_ )
10.#define _MIXPFCP_H_
11.
12./* ------------------------------------------------------ */
13./* Fijos de la app concreta                                                */
14.#define NMEAS 2                                          // [x z]
15.#define NMEAS3D 2                                        // [x z]
16.#define NPROCESS3D 4                                     // [x z vx vz]
17.#define NPROCESSOR 5               // Número de estados con orientación añadida [x z vx vz teta]
18.#define NPROCESSLOC 3                                 // [x z teta]
19.#define NPROCESSOD  2                                 // [vang vlineal]
20.#define NPARAMKMEANS 8
21.#define NPARAMSUBSTR 10
22.
23.#define NVISION 1000
24.#define MAXCLUSTER NVISION/10
25.#define MAXCLUSTEROUT NVISION/10
26.#define MAXBUFFER NVISION/10
27.
28.#define KMEANS 0
29.#define SUBSTRACT 1
30.#define INVALID -1
31.#define INTINF 32000
32.#define MAXITER 70
33.#define MAXSTR 250
34.#define MAXKLIK 10
35.#define PI2 3.14159
36.#define TAMBUFFER 40
37.#define ORBUFFER 15                    // Buffer circular para el calculo de la teta filtrada
38.#define VARV 10                        // Varianza del error de la velocidad lineal (10 ó 5)
39.#define VART 1*PI2/180                 // Varianza del error de la velocidad angular (1 grado)
40./* ------------------------------------------------------ */
41.
42./* ------------------------------------------------------ */
43./* Por Parametrizar                                                        */
44.#define NPARTICMAX 900
45.#define NPARTIC 900
46.#define NMRELATIONMAX 100
47.#define NMRELATION 40
48.
49.#define COVMEASSQRT 70
50.#define COVMEAS (70*70)
51.#define DOBLEPISQRT 2.5066
```

```
52. #define NORMFACT (COVMEASSQRT*DOBLEPISQRT)
53. /* ------------------------------------------------------ */
54.
55. /* ------------------------------------------------------ */
56. /* Parametrizados a traves de trackbar                                  */
57. #define MD 850                          //Substr:500 //KMeans:700
58. #define MDMAX 2000
59. #define MDMIN 10
60. #define MDO 800
61. #define MDOMAX 2000
62. #define MDOMIN 10
63. #define MR 700
64. #define MRMAX 2000
65. #define MRMIN 10
66. #define MRO 700
67. #define MROMAX 2000
68. #define MROMIN 10
69.
70. #define UM1 2                                                 //Substr:0 //KMeans:1
71. #define UM1MAX 10
72. #define UM2 2
73. #define UM2MAX 10
74. #define UM1O 2
75. #define UM1OMAX 10
76. #define UM2O 2
77. #define UM2OMAX 10
78.
79. #define HIST 5
80. #define HISTMAX 10
81. #define HISTO 5
82. #define HISTOMAX 10
83.
84. #define FACTOLV 8
85. #define FACTOLVMAX 10
86. #define FACTOLVO 8
87. #define FACTOLVOMAX 10
88.
89. #define MC 0                                                  //Substr:4 //KMeans:3
90. #define MCMAX 5
91. #define MCO 2
92. #define MCOMAX 5
93. /* ------------------------------------------------------ */
94. #define TRUE 1
95. #define FALSE 0
96.
97. #endif
```

10.1.3. miXpfcp.c

```
1.  /*************************************************************************
2.  * miXpfcp.c
3.  * PURPOSE  : The extended particle filter with vision data and clustering pre-process
```

```
 * AUTHORS  : Marta Marron Romera (marta@depeca.uah.es)
 * DATE     : 1 May 2007
 * SINTAXIS : xpfVision nfile Ts(fps) tC(0->KM,1->S) tCO(0->KM,1->S)
 *                                          doAdjust(1->trackbar xpf,1->clusterIn,2->clusterOut,0->NADA) doResult
 * INNOVATIONS :Global app. from xpfVisionGlobal7. To compile with misMaths and miCluster
 *                            state_vector=x=[x z y vx vz vy] data float in 32bits
 * MODIFICACION:cambiamos vbles globales por locales al main, eliminamos ipls para pintar solo en XZ
 ************************************************************************/

#include "ispace.h"


// ----------------------------------------------------------------------
// Definition of external functions
//      ATT: Orden del array de params en clustering
//              params[0] = ns; params[1] = ne; params[2] = mc; params[3] = hist; params [4] = factOlv;
//              params[5] = md; params[6] = mr (solo substract) params[7] = umbral1;
//              params[8] = umbral2 (solo substract); params[9] = doResult;
// ----------------------------------------------------------------------

void extern miResidual(int *,int,int *,int,float *);
void extern matrixActualA(float *,int,int,float,float *);
void extern matrixActualC(float *,float *,int,int,float);
void extern matrixSum(float *,int,int,int,float *,float *);
void extern DrawingXZProy(IplImage *, float *,CvScalar,int,int,char*);
int extern miSubstract(float *,XpfCluster *,XpfCluster *,float *,int *,int *,float,int *,FILE *,IplImage *);
extern int miKmeans(float *,XpfCluster *,XpfCluster *,float *,int *,int *, float,int *,FILE *,IplImage *);
const int array_R[]={0,  0,     0,     0,     255,     105,    125,   0,   125,   125,   0};
const int array_G[]={0,  0,     255,    255,   0,   105,   125,   125,   0,   125,    125};
const int array_B[]={0, 255,   0,   255,   255,   0,   0,    125,     125,     200,     0};

// ----------------------------------------------------------------------
// Xpf algorithm: Exectute the xpf algorithm with the sensor measurements
// The output will be graphic and the centroid of each detected obstacle
// ----------------------------------------------------------------------

void miXpf(int iter,int doResult,float At,XpfMeas Meas,float *xPartic,XpfCluster *cluster,
                   XpfCluster *clusterOut,int *ks,int *cParams,int *cParamsO,int *xParams,
                   IplImage *iplXZProyColor,FILE *fResult)
{
  register int i,j,l,indexAux,h;
  static int validated[MAXCLUSTER];
  int k,kOut,kValidated,kValidatedOut,nPartic,nParticAux,nParticAsig,nParticRest,*indexIn,*indexOut;
  static float centroidIni[MAXCLUSTER*NMEAS];
  float dist,wAcum,wMax,*pNoise,*wPartic,*xParticPred;
  char str[MAXSTR], ident[3];
  static XpfCluster buffer[MAXCLUSTER];
  XpfCluster bufferAux;
  CvFont fuente;
  int cvPx,cvPz,ident_cluster;
  CvScalar color;

  double aux1[TAMBUFFER],p1[2],p2[2];


```

```
59.  // ============================================================
60.  //  INITIALIZING:
61.  // ============================================================
62.
63.  if (iter==0)
64.  {
65.
66.                  // Initializing
67.                  k = kOut = kValidated = kValidatedOut = nPartic = 0; //METI TB K
68.                  memset(centroidIni,0,MAXCLUSTER*NMEAS*sizeof(float));
69.                  memset(centroidIniOut,0,MAXCLUSTER*NMEAS*sizeof(float));
70.                  memset(buffer,0,MAXCLUSTER*sizeof(XpfCluster));
71.                  for(i=0;i<MAXCLUSTER;i++)
72.                  {
73.                          cluster[i].identify=0;
74.                          cluster[i].candidate=0;
75.                          cluster[i].validated=0;
76.                          cluster[i].nMembers=0;
77.                          cluster[i].probability=0;
78.                          memset(cluster[i].centroid,0,NPROCESS3D*sizeof(float));
79.                          memset(cluster[i].members,0,NPARTICMAX*NPROCESS3D*sizeof(float));
80.                  }
81.                  for(i=0;i<MAXCLUSTEROUT;i++)
82.                  {
83.                          clusterOut[i].identify=0;
84.                          clusterOut[i].candidate=0;
85.                          clusterOut[i].validated=0;
86.                          clusterOut[i].nMembers=0;
87.                          clusterOut[i].probability=0;
88.                          memset(clusterOut[i].centroid,0,NPROCESS3D*sizeof(float));
89.                          memset(clusterOut[i].members,0,NPARTICMAX*NPROCESS3D*sizeof(float));
90.                  }
91.                  //memset(cluster,0,MAXCLUSTER*sizeof(XpfCluster));
92.                  //memset(clusterOut,0,MAXCLUSTEROUT*sizeof(XpfCluster));
93.                  //memset(xPartic,0,NPARTICMAX*NPROCESS3D*sizeof(float));
94.
95.                  // Calling the cluster algorithm as initialization
96.                  if (xParams[3]==SUBSTRACT)
97.                  {
98.                          //printf("\nmiSubstract\n");
99.                          kValidated = miSubstract(Meas.y,cluster,buffer,centroidIni,validated,&k,At,cParams,NULL,
100.                                 iplXZProyColor);
101.                 }
102.                 else
103.                 {
104.                         //printf("\nmiKMeans\n");
105.                         kValidated = miKmeans(Meas.y,cluster,buffer,centroidIni,validated,&k,At,cParams,NULL,
106.                                 iplXZProyColor);
107.                 }
108.
109.                 // Generating the first pdf
110.                 if (kValidated>0)
111.                 {
112.                         nParticAux = (int)floor((xParams[2]/kValidated)+0.5);
113.                         nParticRest = xParams[2];
```

```
114.                            //xParams[0] = 0;
115.                            for (i=0;i<kValidated;i++)
116.                            {
117.                                    indexAux = cluster[validated[i]].identify;
118.                                    buffer[indexAux] = cluster[validated[i]];
119.                                    nParticAsig = nParticRest-(kValidated-i-1)*nParticAux;
120.
121.                                    // Asigning particles
122.
123.                                    j = 0;
124.                                    if (buffer[indexAux].nMembers<=nParticAsig)
125.                                    {
126.                                      while (j<buffer[indexAux].nMembers)
127.                                      {
128.                                                    memcpy(&xPartic[(nPartic+j)*NPROCESS3D],&buffer[indexAux].members[j*NPROCESS3D],
129.                                                            NMEAS3D*sizeof(float));
130.                                                    j++;
131.                                      }
132.                                    }
133.                                    else
134.                            {
135.                                      while (j<nParticAsig)
136.                                      {
137.                                                    l = rand()%(buffer[indexAux].nMembers+1);
138.                                                    memcpy(&xPartic[(nPartic+j)*NPROCESS3D],&buffer[indexAux].members[l*NPROCESS3D],
139.                                                                    NMEAS3D*sizeof(float));
140.                                                    j++;
141.                                      }
142.                                    }
143.                                    nPartic += j;
144.                                    nParticRest -= j;
145.                      }
146.                    }
147.
148.                    //printf("\nk:%d, kValid:%d,",k,kValidated);
149.                    xParams[0]=nPartic;
150.                    ks[0]=k;
151.                    ks[1]=kValidated;
152.                    ks[2]=kOut;
153.                    ks[3]=kValidatedOut;
154.                    printf("k=%d, kValidated=%d, kOut=%d, kValidatedOut=%d\n", ks[0], ks[1], ks[2], ks[3]);
155.      }
156.
157.      // ============================================================
158.      //  REINITIALIZING:
159.      // ============================================================
160.
161.      else
162.      {
163.
164.                    // Getting params
165.                    k = ks[0]; kValidated = ks[1]; kOut = ks[2]; kValidatedOut = ks[3];
166.                    nPartic = xParams[0];
167.
168.                    // Actualizing the buffer of cluster-dependent points
```

```
169.                    if (kValidated>0)
170.                    {
171.                            nParticRest = (int)floor(xParams[2]*xParams[1]/100);
172.                            nParticAux = (int)floor((nParticRest/kValidated)+0.5);
173.                            for (i=0;i<kValidated;i++)
174.                            {
175.                                    indexAux = cluster[validated[i]].identify;
176.                                    bufferAux = cluster[validated[i]];
177.                                    nParticAsig = nParticRest-(kValidated-i-1)*nParticAux;
178.
179.                                    // Refilling the buffers AQUI NO TIENE QUE ENTRAR SI ES NUEVO.. COMPROBAR
180.                                    wAcum = cParams[5];
181.                                    j = 0;
182.                                    while ((j<buffer[indexAux].nMembers)&&(bufferAux.nMembers<MAXBUFFER))
183.                                    {
184.                                            dist = 0;
185.                                            for (l=0;l<NMEAS;l++)
186.                                                    dist += (buffer[indexAux].members[j*NPROCESS3D+l]-bufferAux.centroid[l])*
187.                                                            (buffer[indexAux].members[j*NPROCESS3D+l]-bufferAux.centroid[l]);
188.                                            if (dist<wAcum)
189.                                            {
190.                                                    memcpy(&bufferAux.members[bufferAux.nMembers*NPROCESS3D],
191.                                                                    &buffer[indexAux].members[j*NPROCESS3D],NMEAS3D*sizeof(float));
192.                                                    bufferAux.nMembers++;
193.                                            }
194.                                            j++;
195.                                    }
196.                                    buffer[indexAux] = bufferAux;
197.
198.                                    // Inserting new particles
199.                                    j = 0;
200.                                    if (buffer[indexAux].nMembers<=nParticAsig)
201.                                    {
202.                                            while (j<buffer[indexAux].nMembers)
203.                                            {
204.                                                    memcpy(&xPartic[(nPartic+j)*NPROCESS3D],&buffer[indexAux].members[j*NPROCESS3D],
205.                                                            NMEAS3D*sizeof(float));
206.                                                    memcpy(&xPartic[((nPartic+j)*NPROCESS3D)+NMEAS3D],&buffer[indexAux].centroid[NMEAS3D],
207.                                                            (NPROCESS3D-NMEAS3D)*sizeof(float));
208.                                                    j++;
209.                                            }
210.                                    }
211.                                    else
212.                                    {
213.                                            while (j<nParticAsig)
214.                                            {
215.                                                    l = rand()%(buffer[indexAux].nMembers+1);
216.                                                    memcpy(&xPartic[(nPartic+j)*NPROCESS3D],&buffer[indexAux].members[l*NPROCESS3D],
217.                                                            NMEAS3D*sizeof(float));
218.                                                    memcpy(&xPartic[((nPartic+j)*NPROCESS3D)+NMEAS3D],&buffer[indexAux].centroid[NMEAS3D],
219.                                                            (NPROCESS3D-NMEAS3D)*sizeof(float));
220.                                                    j++;
221.                                            }
222.                                    }
223.                                    nPartic += j;
```

```
224.                                    nParticRest -= j;
225.
226.                                    // Here all members buffer[indexAux] are validated with its centroid dx,dy+T. No me gusta n?s
227.                                    buffer[indexAux].centroid[0] = buffer[indexAux].centroid[0]+
228.                                            (buffer[indexAux].centroid[(NPROCESS3D-NMEAS3D)]*At);
229.                                    buffer[indexAux].centroid[1] = buffer[indexAux].centroid[1]+
230.                                            (buffer[indexAux].centroid[(NPROCESS3D-NMEAS3D+1)]*At);
231.                                    matrixActualC(buffer[indexAux].members,&buffer[indexAux].centroid[(NPROCESS3D-NMEAS3D)],
232.                                                    buffer[indexAux].nMembers,NPROCESS3D,At);
233.                            }
234.                    }
235.
236.                    // ============================================================
237.                    // PREDICTION STEP:
238.                    // ============================================================
239.
240.                    // Obtaning xParticPred
241.                    if ((xParticPred = (float *)malloc(nPartic*NPROCESS3D*sizeof(float)))==NULL)
242.                            printf("Error de malloc\n");
243.                    matrixActualA(xPartic,nPartic,NPROCESS3D,At,xParticPred);
244.                    if ((pNoise = (float *)malloc(nPartic*NPROCESS3D*sizeof(float)))==NULL) printf("Error de malloc\n");
245.                    for (i=0;i<nPartic*NPROCESS3D;i++) pNoise[i] = COVMEASSQRT*((((float)rand())-RAND_MAX/2)/RAND_MAX);
246.                            matrixSum(xParticPred,nPartic,NPROCESS3D,NPROCESS3D,pNoise,xParticPred);
247.                    free(pNoise);
248.
249.                    // ============================================================
250.                    // EVALUATE IMPORTANCE WEIGHTS:
251.                    // ============================================================
252.
253.                    // Organizing new meas first of all with kMeans: Estimating the centroids firstly
254.                    for (i=0;i<k;i++)
255.                    {
256.                            memcpy(&centroidIni[cluster[i].identify*NMEAS],cluster[i].centroid,NMEAS*sizeof(float));
257.                            matrixActualC(cluster[i].centroid,&cluster[i].centroid[(NPROCESS3D-NMEAS3D)],1,NPROCESS3D,At);
258.                    }
259.
260.                    // Estimating the rest of a-vector for validated and candidate clusters. No gusta n?s
261.                    if (xParams[3]==SUBSTRACT)
262.                    {
263.                            printf("miSubstract\n");
264.                            kValidated = miSubstract(Meas.y,cluster,buffer,centroidIni,validated,&k,At,cParams,NULL,
265.                                    iplXZProyColor);
266.                    }
267.                    else
268.                    {
269.                            kValidated = miKmeans(Meas.y,cluster,buffer,centroidIni,validated,&k,At,cParams,NULL,
270.                                    iplXZProyColor);
271.                            //printf("miKMeans %d\n",kValidated);
272.                    }
273.                    j = (int)sqrt(cParams[5]);
274.                    for (i=0;i<k;i++)
275.                    {
276.                            cluster[i].centroid[(NPROCESS3D-NMEAS3D)] = (cluster[i].centroid[0]-
277.                                    centroidIni[cluster[i].identify*NMEAS])/At;
278.                            cluster[i].centroid[(NPROCESS3D-NMEAS3D)] /= 2;
```

```
279.                    cluster[i].centroid[(NPROCESS3D-NMEAS3D+1)] = (cluster[i].centroid[1]-
280.                            centroidIni[(cluster[i].identify*NMEAS)+1])/At;
281.                    cluster[i].centroid[(NPROCESS3D-NMEAS3D+1)] /= 2;
282.
283.                    // Pintando. no gusta n?s
284.                    //printf("cl[%d].ce={%.2f %.2f %.2f %.2f} .id=%d .ca=%d .va=%d .nM=%d .pr=%.4f ce={%.2f %.2f}\n"
285.                    //              ,i,cluster[i].centroid[0],cluster[i].centroid[1],cluster[i].centroid[(NPROCESS3D-NMEAS3D)],
286.                    //              cluster[i].centroid[(NPROCESS3D-NMEAS3D+1)],cluster[i].identify,cluster[i].candidate,
287.                    //              cluster[i].validated,cluster[i].nMembers,cluster[i].probability,
288.                    //              centroidIni[cluster[i].identify*NMEAS],centroidIni[cluster[i].identify*NMEAS+1]);
289.                    if (cluster[i].validated==TRUE)
290.                    {
291.                            wAcum = 255*cluster[i].probability;
292.                            //DrawingXZProy(iplXZProyColor,cluster[i].centroid,CV_RGB(0,255,0),CIRC,j,NULL);
293.                    }
294.            }
295.
296.            // Obtaining the variable for the sampling
297.            if ((wPartic = (float *)malloc(nPartic*sizeof(float)))==NULL) printf("Error de malloc\n");
298.            if ((indexIn = (int *)malloc(nPartic*sizeof(int)))==NULL) printf("Error de malloc\n");
299.            wAcum = wMax = 0;
300.            for (j=0;j<nPartic;j++)
301.            {
302.                    indexIn[j] = INVALID;
303.                    wPartic[j] = cParams[5];
304.                    for (i=0;i<kValidated;i++)
305.                    {
306.                            dist = 0;
307.                            for (l=0;l<NMEAS3D;l++)
308.                              dist += (xParticPred[j*NPROCESS3D+l]-cluster[validated[i]].centroid[l])*
309.                                     (xParticPred[j*NPROCESS3D+l]-cluster[validated[i]].centroid[l]);
310.                            //for (l=NMEAS3D;l<NPROCESS3D;l++)
311.                              //dist += ((xParticPred[j*NPROCESS3D+l]-cluster[validated[i]].centroid[l])*
312.                                     //(xParticPred[j*NPROCESS3D+l]-cluster[validated[i]].centroid[l])*At*At);
313.                            if (dist<wPartic[j])
314.                            {
315.                              wPartic[j] = dist;
316.                              indexIn[j] = i;
317.                            }
318.                    }
319.                    wPartic[j] = (exp(-0.5*wPartic[j]/(float)(COVMEAS)))/(float)(NORMFACT);
320.                    if (wPartic[j]>wMax) wMax = wPartic[j];
321.            }
322.
323.            // Another loop to obtain the weigths using umbral / umbral2
324.            //if (xParams[3]==KMEANS) dist = (float)cParams[6]/10;
325.            //else dist = (float)cParams[8]/10;
326.            for (j=0;j<nPartic;j++)
327.            {
328.                    //if ((indexIn[j]!=INVALID))    //&&(cluster[indexIn[j]].probability>dist))
329.                    //{
330.                            //wPartic[j] = wPartic[j]*(cluster[indexIn[j]].probability); // /dist);
331.                            //if (wPartic[j]>wMax) wPartic[j] = wMax;
332.                    //}
333.                    //else printf("peso no asignado a clase!!! w=%.2f\n",wPartic[j]);
```

```
334.                        wAcum += wPartic[j];
335.                }
336.
337.                // Obtaining the weights and preparing the index array
338.                dist = 0;
339.                j = 0;
340.                if (wAcum==0) printf("!!Error division por 0, wAcum\n");
341.                for (i=0;i<nPartic;i++)
342.                {
343.                        wPartic[i] = wPartic[i]/wAcum;
344.                        if (wPartic[i]>(0.1/nPartic)) j++;
345.                        indexIn[i] = i;
346.                        dist += (wPartic[i]*wPartic[i]);
347.                }
348.                //printf("Neff=%.2f > nPartic/2=%.2f ?? wPartic>0==%d\n",(float)(1/dist),(float)(nPartic/2),i);
349.
350.                // ========================================================================
351.                // SELECTION STEP:
352.                // ========================================================================
353.
354.                // Calling the resampling algorithm
355.                nParticAux = xParams[2]-(int)floor(xParams[2]*xParams[1]/100);
356.                if ((indexOut = (int *)malloc(nParticAux*sizeof(int)))==NULL) printf("Error de malloc\n");
357.                if ((Meas.ny==0)&&(k==0))
358.                {
359.                        miResidual(indexIn,nPartic,indexOut,0,wPartic);
360.                        nPartic = 0;
361.                }
362.                else
363.                {
364.                        miResidual(indexIn,nPartic,indexOut,nParticAux,wPartic);
365.                        nPartic = nParticAux;
366.                }
367.
368.                // Writing results
369.                if (doResult)
370.                {
371.                        if (!fwrite("w=[",3,1,fResult)) printf("Error en fwrite init w\n");
372.                        for (i=0;i<nPartic;i++)
373.                        {
374.                                if (sprintf(str,"%.2f ",wPartic[indexOut[i]])==INVALID) printf("Error de sprintf\n");
375.                                if (!fwrite(str,strlen(str),1,fResult)) printf("Error en fwrite w\n");
376.                        }
377.                        if (!fwrite("];\n",3,1,fResult)) printf("Error en fwrite fin w\n");
378.                        if (!fwrite("x=[",3,1,fResult)) printf("Error en fwrite init x\n");
379.                }
380.
381.                // Writing the particles
382.                for (i=0;i<nPartic;i++)
383.                {
384.                        memcpy(&xPartic[i*NPROCESS3D],&xParticPred[indexOut[i]*NPROCESS3D],NPROCESS3D*sizeof(float));
385.
386.                        // Writing result. no gusta n?s
387.                        if (doResult)
388.                        {
```

```
389.                            if (sprintf(str,"%.2f %.2f %.2f %.2f %.2f;\n",xPartic[i*NPROCESS3D],xPartic[i*NPROCESS3D+1],
390.                                    xPartic[i*NPROCESS3D+2],xPartic[i*NPROCESS3D+3],xPartic[i*NPROCESS3D+4])==INVALID)
391.                            printf("Error de sprintf\n");
392.                            if (!fwrite(str,strlen(str),1,fResult)) printf("Error en fwrite x\n");
393.                    }
394.            }
395.
396.            // Writing results
397.            if (doResult)
398.            if (!fwrite("];\n",3,1,fResult)) printf("Error en fwrite fin x\n");
399.
400.            // Freeing space
401.            free(wPartic);
402.            free(indexIn);
403.            free(indexOut);
404.            free(xParticPred);
405.
406.            // ========================================================================
407.            // OBTAINING THE OUTPUTS:
408.            // ========================================================================
409.
410.            // Time actualization of the clusterOut centroids
411.            for (i=0;i<kOut;i++)
412.            {
413.              memcpy(&centroidIniOut[clusterOut[i].identify*NMEAS],clusterOut[i].centroid,NMEAS*sizeof(float));
414.              matrixActualC(clusterOut[i].centroid,&clusterOut[i].centroid[(NPROCESS3D-NMEAS3D)],1,NPROCESS3D,At);
415.            }
416.
417.            // Calling the clustering process. nPartic has actualized since Xpf was called (without validatedOut)
418.            cParamsO[1] = nPartic;
419.            if (xParams[4]==SUBSTRACT)
420.            {
421.                    printf("miSubstractOut\n");
422.                    kValidatedOut = miSubstract(xPartic,clusterOut,NULL,centroidIniOut,NULL,&kOut,At,cParamsO,
423.                            fResult,iplXZProyColor);
424.            }
425.            else
426.            {
427.                    kValidatedOut = miKmeans(xPartic,clusterOut,NULL,centroidIniOut,NULL,&kOut,At,cParamsO,
428.                            fResult,iplXZProyColor);
429.                    //printf("miKMeansOut %d\n",kValidatedOut);
430.            }
431.            j = (int)sqrt(cParamsO[5]);
432.            for (i=0;i<kOut;i++)
433.            {
434.                    clusterOut[i].centroid[(NPROCESS3D-NMEAS3D)] = (clusterOut[i].centroid[0]-
435.                            centroidIniOut[clusterOut[i].identify*NMEAS])/At;
436.                    clusterOut[i].centroid[(NPROCESS3D-NMEAS3D)] /= 2;
437.                    clusterOut[i].centroid[(NPROCESS3D-NMEAS3D+1)] = (clusterOut[i].centroid[1]-
438.                            centroidIniOut[(clusterOut[i].identify*NMEAS)+1])/At;
439.                    clusterOut[i].centroid[(NPROCESS3D-NMEAS3D+1)] /= 2;
440.                    sprintf(ident,"%d",clusterOut[i].identify);
441.                    cvInitFont(&fuente,CV_FONT_HERSHEY_DUPLEX,0.7,0.7,0,1,8);
442.                    cvPx=(int)clusterOut[i].centroid[0]/12+30;
443.                    cvPz=(int)clusterOut[i].centroid[1]/12-30;
```

```
444.                            cvPutText(iplXZProyColor,ident,cvPoint(cvPx,cvPz),&fuente,CV_RGB(255,0,0));
445.
446.                            // Pintando. No gusta n?s
447.                            /*printf("cl[%d].ce={%.2f %.2f %.2f %.2f} .id=%d .ca=%d .va=%d .nM=%d .pr=%.4f ce={%.2f %.2f}\n"
448.                                    ,i,clusterOut[i].centroid[0],clusterOut[i].centroid[1],clusterOut[i].centroid[(NPROCESS3D-NMEAS3D)],
449.                                    clusterOut[i].centroid[(NPROCESS3D-NMEAS3D+1)],clusterOut[i].identify,
450.                                    clusterOut[i].candidate,clusterOut[i].validated,clusterOut[i].nMembers,clusterOut[i].probability,
451.                                    centroidIniOut[clusterOut[i].identify*NMEAS],centroidIniOut[clusterOut[i].identify*NMEAS+1]);*/
452.                            if (clusterOut[i].validated==TRUE)
453.                            {
454.                                    wAcum = 255*clusterOut[i].probability;
455.                                    ident_cluster=clusterOut[i].identify%10;
456.                                    color=CV_RGB(array_B[ident_cluster],array_G[ident_cluster],array_R[ident_cluster]);
457.
458.
        DrawingXZProy(iplXZProyColor,clusterOut[i].centroid,CV_RGB(array_B[ident_cluster],array_G[ident_cluster],array_R[ident_cluster]),CIRC,j,NULL);
459.                                    /////////MOSTRAR TRAYECTORIAS//////////////////////
460.                                    for (h=0; h<TAMBUFFER; h++)
461.                                            aux1[h] = trayX[clusterOut[i].identify][h];
462.                                    for (h=0; h<TAMBUFFER-1; h++)
463.                                            trayX[clusterOut[i].identify][h+1] = aux1[h];
464.                                    for (h=0; h<TAMBUFFER; h++)
465.                                            aux1[h] = trayZ[clusterOut[i].identify][h];
466.                                    for (h=0; h<TAMBUFFER-1; h++)
467.                                            trayZ[clusterOut[i].identify][h+1] = aux1[h];
468.                                    trayX[clusterOut[i].identify][0] = clusterOut[i].centroid[0] / CONVX;
469.                                    trayZ[clusterOut[i].identify][0] = clusterOut[i].centroid[1] / CONVZXZ;
470.                                    // pintando la trayectoria
471.                                    for (h=0; h<TAMBUFFER-1; h++)
472.                                    {
473.                                            p1[0] = trayX[clusterOut[i].identify][h];
474.                                            p1[1] = trayZ[clusterOut[i].identify][h];
475.                                            p2[0] = trayX[clusterOut[i].identify][h+1];
476.                                            p2[1] = trayZ[clusterOut[i].identify][h+1];
477.
478.                                    if (p1[0] && p1[1] && p2[0] && p2[1]) // && clusterOut[i].validated == TRUE)
479.                                            cvLine(iplXZProyColor, cvPoint((int)p1[0],(int)p1[1]), cvPoint((int)p2[0],(int)p2[1]), color,1,4,0);
480.
481.                                    }
482.
483.
484.                            }
485.
486.                     }
487.
488.                    // Writing the call to the Matlab processing function
489.                    if (doResult)
490.                    {
491.                            if (!fwrite("plotting;\n",10,1,fResult)) printf("Error de fwrite fin de fResult\n");
492.                            if (sprintf(str,"k:%d, kValid:%d, kOut:%d, kValidOut:%d",k,kValidated,kOut,kValidatedOut)==INVALID)
493.                            printf("Error de sprintf\n");
494.                            DrawingXZProy(iplXZProyColor,NULL,CV_RGB(0,0,0),TEXTO,0,str);
495.                            printf("\nk:%d, kValid:%d, kOut:%d, kValidOut:%d,",k,kValidated,kOut,kValidatedOut);
496.                    }
497.                    //ks[]={k,kValidated,kOut,kValidatedOut};
```

```
498.                    xParams[0]=nPartic;
499.                    ks[0]=k;
500.                    ks[1]=kValidated;
501.                    ks[2]=kOut;
502.                    ks[3]=kValidatedOut;
503.            }
504.    }
505.
506.
507.
508.    void miLiveIdent(IplImage *iplXZProyColor, float At, odometry odo)
509.    {
510.            register int i,j,l,m,p,indexAux;        // Several indexes to use in loops. indexAux stores the identifier of a validated class
511.            static int eval[MAXCLUSTEROUT],index[MAXCLUSTEROUT];
512.            int idaux;
513.            static float angulos[MAXCLUSTEROUT*ORBUFFER];
514.            static int bufferVot[MAXCLUSTEROUT];
515.            float orient, orientacion[5],sen,coseno,orient_filt,acum,radsg,gradsg,vl;
516.            float aux,aux2,Jaux[NPROCESSLOC*NPROCESSOD],Jx[NPROCESSLOC*NPROCESSLOC],JxT[NPROCESSLOC*NPROCESSLOC];
517.            float E[(NPROCESSLOC*NPROCESSLOC)*(TAMBUFFER+1)],Eu[NPROCESSOD*NPROCESSOD];
518.            float mAux[NPROCESSLOC*NPROCESSLOC],mAux2[NPROCESSLOC*NPROCESSLOC],mAuxT[NPROCESSLOC*NPROCESSLOC],mAuxT2[NPROCESSLOC*NPROCESSLOC];
519.            float vel_acum[NPROCESS3D-NMEAS3D];
520.            static float velocidad[MAXCLUSTEROUT*ORBUFFER*(NPROCESS3D-NMEAS3D)],vel_filt[NPROCESS3D-NMEAS3D];
521.            char ident[3];
522.            static XpfOrient bufferOrient[MAXCLUSTEROUT];
523.            static odometry bufferOdometry[TAMBUFFER];              // Buffer de estructuras de odometría
524.            CvFont fuente;
525.            int cvPx,cvPz;
526.            static int h,idmaxdMah,idmindMah,maxVot;
527.            float mDest[NPROCESSLOC*NPROCESSLOC],mPrueba[9],mP2[6],mAux3x1[NPROCESSLOC],mAux1x3[NPROCESSLOC],mDest3x1[NPROCESSLOC],mDest1x1[1];
528.            float xdM,zdM,tdM,dMah[MAXCLUSTEROUT],mAux3[NPROCESSLOC*NPROCESSLOC],mindMah,maxdMah;
529.            static float reg_orient[] = {0,0};      // Stores the two latest filtered orientations, to calculate the rotational speed, also filtered
530.            static int contador=0;
531.            static int eltos_dmah;  // Número de elementos sobre los cuales se hace la media de dMah
532.
533.            static int init=0;
534.            CvScalar color;
535.            int ident_cluster;
536.            double aux1[TAMBUFFER],p1[2],p2[2];     // Para pintar trayectoria estimada
537.
538.            static double pose_odo_x[TAMBUFFER][2];
539.            static double pose_odo_z[TAMBUFFER][2];
540.            static double angu_odo[TAMBUFFER][2];
541.
542.
543.
544.            // Inicialización
545.            if (!init)
546.            {
547.                    memset(centroidOr,0,MAXCLUSTER*NPROCESSOR*sizeof(float));
548.                    memset(angulos,0,MAXCLUSTEROUT*ORBUFFER*sizeof(float));
549.                    memset(velocidad,0,MAXCLUSTEROUT*ORBUFFER*(NPROCESS3D-NMEAS3D)*sizeof(float));

550.                    memset(eval,0,MAXCLUSTEROUT*sizeof(int));
551.                    memset(index,0,MAXCLUSTEROUT*sizeof(int));
```

```
552.                    memset(bufferOrient,0,MAXCLUSTEROUT*sizeof(XpfOrient));
553.                    memset(E,0,NPROCESSLOC*NPROCESSLOC*(TAMBUFFER+1)*sizeof(float));
554.                    memset(bufferVot,0,MAXCLUSTEROUT*sizeof(int));
555.
556.                    init=1;
557.            }
558.
559.            for (i=0;i<ks[2];i++)   // For each class
560.            {
561.                    if (clusterOut[i].validated==TRUE)      // If it's validated
562.                    {
563.                            // ========================================================================
564.                            // CALCULATE THE FILTERED ORIENTATION
565.                            // ========================================================================
566.
567.                            // For clarity purposes
568.                            idaux=clusterOut[i].identify;
569.
570.                            // Store the x z centroid measurements in bufferOrient
571.                            // Store the x z centroid measurements in centroidOr, and also vx and vz
572.                            memcpy(&bufferOrient[idaux].xmeas[index[idaux]*NPROCESSLOC],clusterOut[i].centroid,NMEAS3D*sizeof(float));      // Store x, z
573.                            memcpy(&centroidOr[idaux*(NPROCESSOR)],clusterOut[i].centroid,(NPROCESSOR-1)*sizeof(float));    // Store x, z, vx, vz
574.
575.                            // Calculate the current orientation
576.                            sen=-(clusterOut[i].centroid[1]-centroidIniOut[(idaux*NMEAS)+1]);
577.                            coseno=(clusterOut[i].centroid[0]-centroidIniOut[idaux*NMEAS]);
578.                            orient = atan(sen/coseno);      // Rad
579.                            orient=(orient*180)/PI;         // Deg
580.
581.                            if (sen<0 && coseno>0)
582.                                    orient=360-orient;
583.                            if ((sen<0 && coseno<0)||(sen>0 && coseno<0))
584.                                    orient+=180;
585.
586.                            while (orient > 360)
587.                                    orient -= 360;
588.
589.                            memcpy(&angulos[(idaux)*ORBUFFER+eval[idaux]],&orient,1*sizeof(float));         // Store theta
590.                            memcpy(&velocidad[((idaux)*ORBUFFER*(NPROCESS3D-NMEAS3D))+(2*eval[idaux])],&clusterOut[i].centroid[(NPROCESS3D-
    NMEAS3D)],NMEAS3D*sizeof(float)); // Store vx, vz
591.
592.                            eval[idaux]++;
593.                            if (eval[idaux]==ORBUFFER)
594.                                    eval[idaux]=0;
595.
596.                            // Calculate the filtered orientation and speed components
597.                            acum=0;
598.                            vel_acum[0]=vel_acum[1]=0;
599.                            for (j=0;j<ORBUFFER;j++)
600.                            {
601.                                    acum+=angulos[idaux*ORBUFFER+j];
602.                                    vel_acum[0]+=velocidad[idaux*ORBUFFER*(NPROCESS3D-NMEAS3D)+(NMEAS3D*j)];
603.                                    vel_acum[1]+=velocidad[(idaux*ORBUFFER*(NPROCESS3D-NMEAS3D))+1+(NMEAS3D*j)];
604.                            }
605.
```

```
606.                            acum/=ORBUFFER;                 // acum era la forma de obtener la orientación offline, que como te comenté funciona no muy bien. Lo
        cambíe para calcularla
607.                                                            // a partir de vx, vz filtradas
608.
609.                            vel_acum[0]/=ORBUFFER;
610.                            vel_filt[0]=vel_acum[0];        // vx
611.                            vel_acum[1]/=ORBUFFER;
612.                            vel_filt[1]=vel_acum[1];        // vz
613.
614.                            orient_filt =atan(vel_acum[1]/vel_acum[0])*180/PI;
615.                            if (vel_filt[1]<0 && vel_acum[0]>0)
616.                                    orient_filt =360+atan(vel_acum[1]/vel_acum[0])*180/PI;
617.                            if ((vel_filt[1]<0 && vel_acum[0]<0)||(vel_filt[1]>0 && vel_acum[0]<0))
618.                                    orient_filt =180 + atan(vel_acum[1]/vel_acum[0])*180/PI;
619.                            while (orient_filt > 360)
620.                                    orient_filt -= 360;
621.
622.                            orient_filt = 360 - orient_filt;        // Done like this so the angles increase anti-clockwise
623.
624.                            // Get the linear and rotational speed from the odometry
625.                            //vl = odo.linspeed;
626.                            //radsg = odo.rotspeed;
627.                            // En pose_odo_x[idaux][0] se guarda la posicion en x en t. En pose_odo_x[idaux][1], en t-1
628.                            aux = pose_odo_x[idaux][0];
629.                            pose_odo_x[idaux][0] = odo.x;
630.                            pose_odo_x[idaux][1] = aux;
631.                            aux = pose_odo_z[idaux][0];
632.                            pose_odo_z[idaux][0] = odo.y;
633.                            pose_odo_z[idaux][1] = aux;
634.
635.                            aux = angu_odo[idaux][0];
636.                            angu_odo[idaux][0] = odo.phi;
637.                            angu_odo[idaux][1] = aux;
638.
639.                            vl = sqrt( (pose_odo_x[idaux][0]-pose_odo_x[idaux][1])*(pose_odo_x[idaux][0]-pose_odo_x[idaux][1]) + (pose_odo_z[idaux][0]-
        pose_odo_z[idaux][1])*(pose_odo_z[idaux][0]-pose_odo_z[idaux][1]) );
640.                            radsg = angu_odo[idaux][0] - angu_odo[idaux][1];
641.
642.
643.                            // Insert a new filtered orientation value in the array, to calculate the filtered rotational speed
644.                            reg_orient[0] = reg_orient[1];                  // Angular position in t-1
645.                            reg_orient[1] = orient_filt;                    // Angular position in t
646.                            gradsg = (reg_orient[1] - reg_orient[0]) / At;  // Rotational speed in degrees
647.
648.                            // Store the filtered orientation in the cluster and array
649.                            memcpy(&angulos[(idaux)*ORBUFFER+eval[idaux]],&orient_filt,1*sizeof(float));
650.                            memcpy(&centroidOr[idaux*(NPROCESSOR)+NPROCESS3D],&orient_filt,1*sizeof(float));        // Store theta (in deg) in centroidOr
651.
652.                            // Store the orientation in degrees (from 0 to 360)
653.                            memcpy(&bufferOrient[idaux].xmeas[(index[(idaux)]*NPROCESSLOC)+NMEAS3D],&orient_filt,1*sizeof(float));
654.
655.                            // Store the linear and rotational speed
656.                            memcpy(&bufferOrient[idaux].vang[index[idaux]],&radsg,1*sizeof(float));
657.                            memcpy(&bufferOrient[idaux].vlineal[index[idaux]],&vl,1*sizeof(float));
658.
```

```
659.                            // Draw the orientation
660.                            //if (clusterOut[i].validated==TRUE)
661.                            //{
662.
663.                                    ident_cluster=clusterOut[i].identify%10;
664.                                    color=CV_RGB(array_B[ident_cluster],array_G[ident_cluster],array_R[ident_cluster]);
665.                                    memcpy(&orientacion,&centroidOr[clusterOut[i].identify*NPROCESSOR],NPROCESSOR*sizeof(float));
666.                                    orientacion[2]=vel_filt[0];
667.                                    orientacion[3]=vel_filt[1];
668.                                    if (contador>ORBUFFER)
669.
670.                                            DrawingXZProy(iplXZProyColor,orientacion,color,FLECHA,0,NULL);
671.                            //}
672.
673.                            // ================================================================================
674.                            // CALCULATE THE ESTIMATED CLASS BEHAVIOUR, USING THE CLASS CENTROID AND THE ODOMETRY
675.                            // ================================================================================
676.
677.
678.                            /*if (index[idaux]>0)
679.                                    aux=bufferOrient[idaux].xmeas[((index[idaux]-1)*NPROCESSLOC)+NMEAS3D]*PI/180;                  // Angular position in t-1
680.
681.                            aux+=bufferOrient[idaux].vang[index[idaux]]*PI/180;                                                    // Add the rotational speed
682.                            aux+=(float)VART;                                                                                  // Rotational speed error variance
683.
684.                            aux2 = bufferOrient[idaux].xmeas[((index[idaux])*NPROCESSLOC)+NMEAS3D]*PI/180 + bufferOrient[idaux].vang[index[idaux]]*PI/180;  //
        Rotational position in t + rotational speed
685.
686.                            if (index[idaux]>0)
687.                            {
688.                                    bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC]=bufferOrient[idaux].xmeas[(index[idaux]-
        1)*NPROCESSLOC]+(bufferOrient[idaux].vlineal[index[idaux]]+VARV)*cos(aux2);
689.                                    bufferOrient[idaux].xest[(index[idaux]*NPROCESSLOC)+1]=bufferOrient[idaux].xmeas[((index[idaux]-
        1)*NPROCESSLOC)+1]+(bufferOrient[idaux].vlineal[index[idaux]]+VARV)*(-1)*sin(aux2);
690.                            }
691.                            bufferOrient[idaux].xest[(index[idaux]*NPROCESSLOC)+2]=aux;
692.
693.                            while (bufferOrient[idaux].xest[(index[idaux]*NPROCESSLOC)+2] > 2*PI)
694.                                    bufferOrient[idaux].xest[(index[idaux]*NPROCESSLOC)+2] -= 2*PI;
695.
696.                            // Results visualization
697.                            if (index[idaux]>0)
698.                            {
699.                                    printf("\n[%d] Datos centroide\t\t\t\t\t[%d] Vector de estado estimado\n", idaux, idaux);
700.                                    printf("-------------------------------\t\t\t\t-------------------------------\n");
701.                                    printf("x medida: %f\t\t\t\t\tx estimada: %f\n", bufferOrient[idaux].xmeas[(index[idaux]-
        1)*NPROCESSLOC],bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC] );
702.                                    printf("z medida: %f\t\t\t\t\tz estimada: %f\n", bufferOrient[idaux].xmeas[(index[idaux]-
        1)*NPROCESSLOC+1],bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC+1] );
703.                                    printf("t medida (rad): %f, vang (rad/sg): %f\tt estimada (rad): %f\n", bufferOrient[idaux].xmeas[((index[idaux]-
        1)*NPROCESSLOC)+NMEAS3D]*PI/180 , bufferOrient[idaux].vang[index[idaux]]*PI/180, bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC+2]);
704.                            }*/
705.
706.
707.                            if (index[idaux]==0)    // Si estamos en el primer elemento del buffer, el estimado es igual a la medida actual
```

```
708.                            {
709.                                    bufferOrient[idaux].xest[0] = bufferOrient[idaux].xmeas[0];
710.                                    bufferOrient[idaux].xest[1] = bufferOrient[idaux].xmeas[1];
711.                                    bufferOrient[idaux].xest[2] = bufferOrient[idaux].xmeas[2];
712.                            }
713.
714.                            for (h=1; h<TAMBUFFER; h++)     // Desde h=1 porque h=0 es la medida del pf a la q hacemos caso, a partir de ahi estimamos con
        velocidad
715.                            {
716.                                    // Calcular posicion angular actual + velocidad angular -> integrar la posición angular
717.                                    aux2 = bufferOrient[idaux].xmeas[(h*NPROCESSLOC)+NMEAS3D]*PI/180 + bufferOrient[idaux].vang[h]*PI/180;
718.
719.                                    // Estimar X
720.                                    bufferOrient[idaux].xest[h*NPROCESSLOC] = bufferOrient[idaux].xest[(h-1)*NPROCESSLOC] +
        (bufferOrient[idaux].vlineal[h])*cos(aux2);
721.                                    // Estimar Z
722.                                    bufferOrient[idaux].xest[h*NPROCESSLOC+1] = bufferOrient[idaux].xest[(h-1)*NPROCESSLOC+1] +
        (bufferOrient[idaux].vlineal[h])*sin(aux2)*(-1);
723.                                    // Estimar phi
724.                                    aux=bufferOrient[idaux].xest[((h-1)*NPROCESSLOC)+NMEAS3D]*PI/180;                        // Angular position in t-1
725.                                    aux+=bufferOrient[idaux].vang[h]*PI/180;                                                // Add the rotational speed
726.
727.
728.                                    bufferOrient[idaux].xest[h*NPROCESSLOC+2] = aux;
729.
730.                            }
731.
732.
733.                            /////////MOSTRAR TRAYECTORIAS//////////////////////
734.                            for (h=0; h<TAMBUFFER; h++)
735.                                    aux1[h] = trayX_est[clusterOut[i].identify][h];
736.                            for (h=0; h<TAMBUFFER-1; h++)
737.                                    trayX_est[clusterOut[i].identify][h+1] = aux1[h];
738.                            for (h=0; h<TAMBUFFER; h++)
739.                                    aux1[h] = trayZ_est[clusterOut[i].identify][h];
740.                            for (h=0; h<TAMBUFFER-1; h++)
741.                                    trayZ_est[clusterOut[i].identify][h+1] = aux1[h];
742.                            trayX_est[clusterOut[i].identify][0] = bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC] / CONVX;
743.                            trayZ_est[clusterOut[i].identify][0] = bufferOrient[idaux].xest[index[idaux]*NPROCESSLOC+1] / CONVZXZ2;
744.                                    // pintando la trayectoria
745.                            for (h=0; h<TAMBUFFER-1; h++)
746.                            {
747.                                    p1[0] = trayX_est[clusterOut[i].identify][h];
748.                                    p1[1] = trayZ_est[clusterOut[i].identify][h];
749.                                    p2[0] = trayX_est[clusterOut[i].identify][h+1];
750.                                    p2[1] = trayZ_est[clusterOut[i].identify][h+1];
751.
752.                                    if (p1[0] && p1[1] && p2[0] && p2[1]) // && clusterOut[i].validated == TRUE)
753.                                            cvLine(iplXZProyColor, cvPoint((int)p1[0],(int)p1[1]), cvPoint((int)p2[0],(int)p2[1]), CV_RGB(0,0,0),1,4,0);
754.
755.                            }
756.
757.
758.                            // ===========================================================================
759.                            // CALCULATE J y Jx: -> Llamarlos de otra manera (con el significado fisico)
```

```
760.                            // ========================================================================
761.                            Jx[0]=1;
762.                            Jx[1]=0;
763.                            if (index[idaux]>0)
764.                                    Jx[2]= -bufferOrient[idaux].vlineal[index[idaux]] * sin(((bufferOrient[idaux].xmeas[((index[idaux]-
        1)*NPROCESSLOC)+NMEAS3D])*PI/180)+bufferOrient[idaux].vang[index[idaux]]*PI/180);
765.                            else
766.                                    Jx[2]= -bufferOrient[idaux].vlineal[index[idaux]] *
        sin(((bufferOrient[idaux].xmeas[((TAMBUFFER)*NPROCESSLOC)+NMEAS3D])*PI/180)+bufferOrient[idaux].vang[index[idaux]]*PI/180);
767.                            Jx[3]=0;
768.                            Jx[4]=1;
769.                            if (index[idaux]>0)
770.                                    Jx[5] = bufferOrient[idaux].vlineal[index[idaux]] * cos(((bufferOrient[idaux].xmeas[((index[idaux]-
        1)*NPROCESSLOC)+NMEAS3D])*PI/180)+bufferOrient[idaux].vang[index[idaux]]*PI/180);
771.                            else
772.                                    Jx[5] = bufferOrient[idaux].vlineal[index[idaux]] *
        cos(((bufferOrient[idaux].xmeas[((TAMBUFFER)*NPROCESSLOC)+NMEAS3D])*PI/180)+bufferOrient[idaux].vang[index[idaux]]*PI/180);
773.                            Jx[6]=0;
774.                            Jx[7]=0;
775.                            Jx[8]=1;
776.
777.                            // Jx transpose
778.                            matrixTras(Jx,3,3,JxT);
779.
780.                            // Compose the J matrix and copy it to the cluster
781.                            Jaux[0]=cos(aux);
782.                            Jaux[1]=-(bufferOrient[idaux].vlineal[index[idaux]]+VARV)*sin(aux);
783.                            Jaux[2]=sin(aux);
784.                            Jaux[3]=(bufferOrient[idaux].vlineal[index[idaux]]+VARV)*cos(aux);
785.                            Jaux[4]=0;
786.                            Jaux[5]=1;
787.                            memcpy(&bufferOrient[idaux].J[index[idaux]*NPROCESSLOC*NPROCESSOD],Jaux,NPROCESSLOC*NPROCESSOD*sizeof(float));
788.
789.
790.                            // ========================================================================
791.                            // CALCULATE E MATRICES (error covariance propagation)
792.                            // ========================================================================
793.
794.                            // E0 is fixed
795.                            E[0]=50;        // x error covariance
796.                            E[4]=50;        // y error covariance
797.                            E[8]=5;         // theta error covariance
798.
799.                            // Eu stores the rotational speed and angular position covariance errors
800.                            Eu[0]=VARV;
801.                            Eu[1]=0;
802.                            Eu[2]=0;
803.                            Eu[3]=VART;
804.
805.                            for (h=0;h<TAMBUFFER;h++)
806.                            {
807.                                    matrixMult(Jx,3,&E[(NPROCESSLOC*NPROCESSLOC)*(h)],mAux);                                        // Jx(3x3ci) * E(3x3) =
        mAux(3x3)
808.                                    matrixMult(mAux,3,JxT,mAux2);                                                                   // mAux2(3x3) = mAux(3x3) *
        (Jx)^t(3x3)        (J*E*J^t)
```

```
809.                                    matrixMult2(&bufferOrient[idaux].J[(h)*NPROCESSLOC*NPROCESSOD],3,2,Eu,mAux);                    // mAux(3x2) = J(3x2) *
        Eu(2x2)
810.                                    memcpy(mAuxT,&bufferOrient[idaux].J[(h)*NPROCESSLOC*NPROCESSOD],NPROCESSLOC*NPROCESSOD*sizeof(float));
811.                                    matrixTras(mAuxT,3,2,mAuxT2);
812.                                    matrixMult3(mAux,3,2,mAuxT2,mAuxT);                                                             // mAuxT = J*Eu*J^t
813.                                    matrixSum(mAux2,NPROCESSLOC,NPROCESSLOC,NPROCESSLOC,mAuxT,&E[(NPROCESSLOC*NPROCESSLOC)*(h+1)]); // E = Jx*E*Jx^t + J*Eu*J^t
814.                            }
815.
816.                            // ========================================================================
817.                            // CALCULATE MAHALANOBIS DISTANCE
818.                            // ========================================================================
819.
820.                            dMah[idaux] = 0;
821.                            eltos_dmah = 0;
822.                            for (h=1;h<TAMBUFFER;h++)
823.                            {
824.                                    // Euclid distance
825.                                    xdM=(bufferOrient[idaux].xmeas[h*NPROCESSLOC]-bufferOrient[idaux].xest[h*NPROCESSLOC]);
826.                                    zdM=(bufferOrient[idaux].xmeas[(h*NPROCESSLOC)+1]-bufferOrient[idaux].xest[(h*NPROCESSLOC)+1]);
827.                                    tdM=((bufferOrient[idaux].xmeas[(h*NPROCESSLOC)+2])*PI/180-bufferOrient[idaux].xest[(h*NPROCESSLOC)+2]);               //
        Radians
828.
829.                                    // Obtain the covariance matrix inverse (if it were and identity matrix, we would be calculating the euclid distance)
830.                                    matrixInv(&E[(NPROCESSLOC*NPROCESSLOC)*(h+1)],3,3,mDest);
831.
832.                                    // ############ Temporal: poner la matriz de covarianza como identidad
833.                                    for (m=0;m<9; m++)
834.                                    {
835.                                            mDest[m] = 0;
836.                                            if(m==0 || m==4 || m==8)
837.                                                    mDest[m]=1;
838.                                    }
839.
840.                                    // Build the euclid distance vectors
841.                                    mAux[0]=xdM;
842.                                    mAux[1]=zdM;
843.                                    mAux[2]=tdM;
844.                                    mAux1x3[0] = xdM;
845.                                    mAux1x3[1] = zdM;
846.                                    mAux1x3[2] = tdM;
847.
848.                                    // Calculate the mahalanobis distance in each loop iteration. Ponderate the euclid distances with the inverted covariance
        matrix
849.                                    matrixMult4(mDest,3,3,mAux,mAux3x1);                    // E-1(3x3) * mAux(3x1) = mAux3x1 (3x1)
850.                                    matrixMult5(mAux1x3,1,3,mAux3x1,mDest1x1);
851.
852.                                    if (mDest1x1[0])
853.                                            eltos_dmah++;
854.
855.                                    // Accumulate the distances in dMah
856.                                    dMah[idaux]+=sqrt(mDest1x1[0]);
857.                            }
858.
859.                            // Average the accumulated distances
860.                            dMah[idaux]/=eltos_dmah;
```

```
861.                            printf("\n[%d] distancia mahalanobis = %f\n", idaux, dMah[idaux]);
862.
863.                            // Increase the circular buffer index, for the next iteration
864.                            index[idaux]++;
865.                            if (index[idaux]==TAMBUFFER)
866.                                    index[idaux]=0;
867.
868.                            /*if (doResult)         // Y esto?
869.                                    if (sprintf(str,"%.2f %.2f %.2f %.2f %.2f;\n",centroidOr[clusterOut[i].identify*NPROCESSOR],
870.                                                                    centroidOr[cluster[i].identify*NPROCESSOR+1],
871.                                                                    centroidOr[clusterOut[i].identify*NPROCESSOR+2],
872.                                                                    centroidOr[clusterOut[i].identify*NPROCESSOR+3],
873.                                                                    centroidOr[clusterOut[i].identify*NPROCESSOR+4])==INVALID)
874.                                            printf("Error de sprintf\n");*/
875.                    }
876.            }
877.
878.            // ================================================================================================
879.            // VOTATION PROCESS
880.            // ================================================================================================
881.
882.            printf("\n");
883.
884.            if(contador>ORBUFFER)
885.            {
886.                    // Calculate dMah,min and dMah,max, and the associated classes
887.                    mindMah=dMah[clusterOut[0].identify];                   // Minimum Mahalanobis distance
888.                    maxdMah=dMah[clusterOut[0].identify];                   // Maximum Mahalanobis disntace
889.                    idmindMah=idmaxdMah=clusterOut[0].identify;             // Associated ids
890.                    for (i=0;i<ks[3];i++)
891.                    {
892.                            idaux=clusterOut[i].identify;                   // For clarity purposes
893.                            if (dMah[idaux]<mindMah)
894.                            {
895.                                    mindMah=dMah[idaux];
896.                                    idmindMah=idaux;
897.                            }
898.                            if (dMah[idaux]>maxdMah)
899.                            {
900.                                    maxdMah=dMah[idaux];
901.                                    idmaxdMah=idaux;
902.                            }
903.                    }
904.
905.                    // Votation system: the class associated with the lowest Mahalanobis distance is awarded (+1)
906.                    // the class associated with the highest Mahalanobis distance in punished (-4). The minimum votation is 0. Currently there is no maximum
        value.
907.                    idmaxVot=clusterOut[0].identify;
908.                    maxVot=bufferVot[0];
909.                    for (i=0;i<ks[3];i++)
910.                    {
911.                            idaux=clusterOut[i].identify;                           // For clarity purposes
912.                            if (idaux==idmaxdMah)
913.                            {
914.                                    bufferVot[idaux]=bufferVot[idaux]-4;
```

```
915.                                    if (bufferVot[idaux]<0)
916.                                            bufferVot[idaux]=0;
917.                                    if (bufferVot[idaux]>maxVot)
918.                                    {
919.                                            maxVot = bufferVot[idaux];
920.                                            idmaxVot=i;
921.                                    }
922.                                    printf("   [%d] -4 -> %d\n", idaux, bufferVot[idaux]);
923.                            }
924.                            if (idaux==idmindMah)
925.                            {
926.                                    bufferVot[idaux]++;
927.                                    if (bufferVot[idaux]>maxVot)
928.                                    {
929.                                            maxVot = bufferVot[idaux];
930.                                            idmaxVot=i;
931.                                    }
932.                                    printf("   [%d] +1 -> %d\n", idaux, bufferVot[idaux]);
933.                            }
934.                            printf("[%d] vale %d\n", idaux, bufferVot[idaux]);
935.                    }
936.
937.                    // Display the word "ROBOT" next to the most voted class
938.                    cvInitFont(&fuente,CV_FONT_HERSHEY_DUPLEX,0.7,0.7,0,1,8);
939.                    cvPx=(int)clusterOut[idmaxVot].centroid[0]/CONVX+40;
940.                    cvPz=(int)clusterOut[idmaxVot].centroid[1]/CONVZXZ2-40;
941.                    cvPutText(iplXZProyColor,"ROBOT",cvPoint(cvPx,cvPz),&fuente,CV_RGB(0,0,0));
942.                    printf("\nEl robot es [%d]\n", clusterOut[idmaxVot].identify);
943.            }
944.
945.            contador++;
946.    }// end of miLiveIdent
```

10.1.4. misMath.h

```
1. /*********************************************************************
2.  * misMaths.h
3.  * PURPOSE  : Defines para ejecutar en misMaths.c
4.  * AUTHORS  : Marta Marron Romera
5.  * DATE     : 1 Mayo 2007
6.  * INNOVATIONS: for miXpfcp + miCluster + misMaths
7.  *********************************************************************/
8.
9. #if !defined( _MISMATHS_H_ )
10.#define _MISMATHS_H_
11.
12./* ---------------------------------------------- *******/
```

```
/* parametros imagen                                                    */
#define WIDTH 320
#define HEIGHT 240
#define FILE_SIZE (WIDTH*HEIGHT)
/* ------------------------------------------- */

/* ------------------------------------------- *******/
/* limites X-Z (en mm). se puede parametriz o mejor hacer adapt.*/
/* OJO: ctes de proyec dependen de ellos, si se parametriza hay que obtenerlas on-line*/
#define XMIN 0
#define XMAX 320*12
#define ZMIN 0
#define ZMAX 240*12
#define ZMAX2 240*12
/* ------------------------------------------- */

/* ------------------------------------------- *******/
/* Obteccion de proyecciones                                        */
#define CONVX ((float)(XMAX-XMIN)/WIDTH)
#define CONVZXZ ((float)(ZMAX-ZMIN)/HEIGHT)
#define CONVZXZ2 ((float)(ZMAX2-ZMIN)/HEIGHT)
#define PI 3.14159
/* ------------------------------------------- */

/* ------------------------------------------- *******/
/* Forma del dibujo                                                    */
#define PTOUV 0
#define PTO 1
#define CIRC 2
#define RECT 3
#define TEXTO 4
#define FLECHA 5
/* ------------------------------------------- */

#endif
```

10.1.5. misMath.c

```
 /*************************************************************************
  * xpVisionMaths.c
  * PURPOSE  : Math functions to work with xpfVisionGlobal
  * AUTHORS  : Marta Marron Romera (marta@depeca.uah.es)
 * DATE     : 1 May 2007
  *************************************************************************/

 #include "ispace.h"

```

```
10.  /*#ifdef _CH_
11.  #pragma package <opencv>
12.  #endif
13.  #ifndef _EiC
14.  #include "cv.h"
15.  #include "highgui.h"
16.  #include <stdio.h>
17.  #include <ctype.h>
18.  #endif
19.  #include <stdio.h>
20.  #include <stdlib.h>
21.  #include <math.h>
22.  #include "miXpfcp.h"
23.  #include "misMaths.h"*/
24.  float invH_0[] = {-1.785621,3.589879,708.791661,-0.810318,-1.158418,791.969197,0.00641468,0.006607866,3.6439302};
25.    float invH_1[] = {4204.905387,383.960128,193058.684109,-67.143048,1553.782785,515112.419867,5.097124,-7.825283,5111.461765};
26.    float invH_2[] = {2839.366697,-2743.757747,643467.588835,-152.711152,1195.868284,692744.426273,-4.400157,-8.967132,6733.856001};
27.    float invH_3[] = {-4253.789073,257.648395,1313636.974660,-678.845373,-1557.273350,832748.062806,-4.850559,8.485926,5388.019972};
28.  // ----------------------------------------------------------------------
29.  // Xpf algorithm fcn: miResidual -> To select particles at the resample
30.  // ----------------------------------------------------------------------
31.  void miResidual(int *indexIn,int nIndexIn,int *indexOut,int nIndexOut,float *q)
32.  {
33.    register int i,j;
34.    int *nBabies,nRes=0,acumI=0;
35.    float *qRes,*dist,acumD;
36.
37.    // Obtaining space
38.    qRes = (float *)malloc(sizeof(float)*nIndexIn);
39.    nBabies = (int *)malloc(sizeof(int)*nIndexIn);
40.    dist = (float *)malloc(sizeof(float)*nIndexIn);
41.    if ((qRes==NULL)||(nBabies==NULL)||(dist==NULL)) printf("Error en malloc");
42.
43.    // First integer part
44.    for (i=0;i<nIndexIn;i++)
45.    {
46.                  qRes[i] = nIndexOut*q[i];
47.                  nBabies[i] = (int)floor(qRes[i]);
48.                  nRes += nBabies[i];
49.    }
50.
51.    // Residual number of particles to sample
52.    nRes = nIndexOut-nRes;
53.
54.    // Generate nRes ordered random variables uniformly distributed in [0,1]
55.    if (nRes>0)
56.    {
57.                  acumD = 0;
58.                  for (i=0;i<nIndexIn;i++)
59.                  {
60.                    qRes[i] = (qRes[i]-nBabies[i])/nRes;
61.                    acumD += qRes[i];
62.                    dist[i] = acumD;
63.                  }
64.                  acumD = 1;
```

```
65.                 j = nIndexIn-1;
66.                 for (i=1;i<=nRes;i++)
67.                 {
68.                         acumD *= pow(((float)rand())/RAND_MAX,(1/(float)(nRes-i+1)));
69.                         while (acumD<dist[j])   j--;
70.                         nBabies[j+1]++;
71.                 }
72.    }
73.
74.    // Copy resampled trajectories
75.    for(i=0;i<nIndexIn;i++)
76.    {
77.                 if (nBabies[i]>0) for (j=acumI;j<acumI+nBabies[i];j++) indexOut[j] = indexIn[i];
78.                 acumI += nBabies[i];
79.    }
80.
81.    // Free space
82.    free(qRes);
83.    free(nBabies);
84.    free(dist);
85.  }
86.
87.  // ------------------------------------------------------------------------
88.  // Xpf matrix mathem functions
89.  // ------------------------------------------------------------------------
90.  void matrixActualC(float *mX,float *mVX,int rX,int cX,float T)
91.  {
92.    register int i;
93.
94.    for (i=0;i<rX;i++)
95.    {
96.                 mX[i*cX]=mX[i*cX]+mVX[0]*T;
97.                 mX[i*cX+1]=mX[i*cX+1]+mVX[1]*T;
98.    }
99.  }
100.
101. void matrixActualA(float *mX,int rX,int cX,float T,float *mR)
102. {
103.   register int i;
104.
105.   for (i=0;i<rX;i++)
106.   {
107.                mR[i*cX+3]=mX[i*cX+3];
108.                mR[i*cX+2]=mX[i*cX+2];
109.                mR[i*cX]=mX[i*cX]+mR[i*cX+2]*T;
110.                mR[i*cX+1]=mX[i*cX+1]+mR[i*cX+3]*T;
111. //             mR[i*cX+2]=mX[i*cX+2];
112.   }
113. }
114.
115.
116. void matrixSum(float *mA,int rA,int cA,int nA,float *mB,float *mC)
117. {
118.   register int i,j;
119.
```

```
120.   for (i=0;i<rA;i++) for (j=0;j<nA;j++) mC[i*cA+j]=mA[i*cA+j]+mB[i*nA+j];
121. }
122.
123. void matrixMult(float *mA,int d,float *mB,float *mC) //multiplica dos matrices 3x3
124. {
125.   register int i,j;
126.   for (i=0;i<d;i++){
127.         j=0;
128.         mC[i*d]=(mA[i*d]*mB[j])+(mA[(i*d)+1]*mB[j+d])+(mA[(i*d)+2]*mB[j+6]);
129.         mC[i*d+1]=(mA[i*d]*mB[j+1])+(mA[(i*d)+1]*mB[j+4])+(mA[(i*d)+2]*mB[j+7]);
130.         mC[i*d+2]=(mA[i*d]*mB[j+2])+(mA[(i*d)+1]*mB[j+5])+(mA[(i*d)+2]*mB[j+8]);
131.   }
132. }
133.
134. void matrixMult2(float *mA,int rA,int cA,float *mB,float *mC) //multiplica matriz 3x2 por matriz 2x2
135. {
136.   register int i;
137.   for (i=0;i<rA;i++){
138.         mC[i*cA]=(mA[i*cA]*mB[0])+(mA[(i*cA)+1]*mB[cA]);
139.         mC[i*cA+1]=(mA[i*cA]*mB[1])+(mA[(i*cA)+1]*mB[3]);
140.   }
141.
142. }
143.
144. void matrixMult3(float *mA,int rA,int cA,float *mB,float *mC){  //multiplica matrix 3x2 por matriz 2x3
145.   register int i;
146.   for (i=0;i<rA;i++){
147.         mC[i*rA]=(mA[i*cA]*mB[0])+(mA[(i*cA)+1]*mB[3]);
148.         mC[i*rA+1]=(mA[i*cA]*mB[1])+(mA[(i*cA)+1]*mB[4]);
149.         mC[i*rA+2]=(mA[i*cA]*mB[2])+(mA[(i*cA)+1]*mB[5]);
150.   }
151. }
152.
153. void matrixMult4(float *mA,int rA,int cA,float *mB,float *mC){ //multiplica matriz 3x3 por matriz 3x1
154.   register int i;
155.         mC[0]=(mA[0]*mB[0])+(mA[1]*mB[1])+(mA[2]*mB[2]);
156.         mC[1]=(mA[3]*mB[0])+(mA[4]*mB[1])+(mA[5]*mB[2]);
157.         mC[2]=(mA[6]*mB[0])+(mA[7]*mB[1])+(mA[8]*mB[2]);
158. }
159.
160. void matrixMult5 (float *mA,int rA,int cA,float *mB,float *mC){ //multiplica matriz 1x3 por matriz 3x1
161.   register int i;
162.         mC[0]=(mA[0]*mB[0])+(mA[1]*mB[1])+(mA[2]*mB[2]);
163. }
164.
165. void matrixTras (float *mT,int r,int c,float *mC){
166.   register i,j;
167.   for (i=0;i<c;i++){
168.         for (j=0;j<r;j++){
169.                mC[(i*r)+j]=mT[(j*c)+i];
170.         }
171.   }
172. }
173. void matrixInv (float *mA,int r,int c,float *mC){
174.   float det;
```

```
175.   det=((mA[0]*mA[4]*mA[8])+(mA[3]*mA[7]*mA[2])+(mA[6]*mA[5]*mA[1]))-((mA[6]*mA[4]*mA[2])+(mA[7]*mA[5]*mA[0])+(mA[8]*mA[3]*mA[1]));
176.   mC[0]=(1/det)*((mA[8]*mA[4])-(mA[7]*mA[5]));mC[1]=-(1/det)*((mA[1]*mA[8])-(mA[2]*mA[7]));mC[2]=(1/det)*((mA[1]*mA[5])-(mA[4]*mA[2]));
177.   mC[3]=-(1/det)*((mA[8]*mA[3])-(mA[5]*mA[6]));mC[4]=(1/det)*((mA[0]*mA[8])-(mA[6]*mA[2]));mC[5]=-(1/det)*((mA[0]*mA[5])-(mA[3]*mA[2]));
178.   mC[6]=(1/det)*((mA[3]*mA[7])-(mA[4]*mA[6]));mC[7]=-(1/det)*((mA[0]*mA[7])-(mA[6]*mA[1]));mC[8]=(1/det)*((mA[0]*mA[4])-(mA[1]*mA[3]));
179. }
180.
181. // ---------------------------------------------------------------------
182. // DrawingProyXZ fcn: DrawPoint
183. // ---------------------------------------------------------------------
184. void DrawingXZProy(IplImage *ipl,float *XZ,CvScalar color,int type,int width,char *text)
185. {
186.   CvFont font;
187.   float u1,v1,u2,v2,angulo,seno,coseno,mod;
188.   char str[MAXSTR];
189.
190.   // Dibujando segn tipo
191.   switch (type)
192.   {
193.
194.                 case PTOUV:
195.
196.                         // Dibujo circulo
197.                         cvCircle(ipl,cvPoint((int)XZ[0],(int)XZ[1]),2,color,1,8,0);
198.                         break;
199.
200.                 case PTO:
201.
202.                         // El centro. No gusta n?s
203.                         u1 = (XZ[0]-(float)XMIN)/CONVX;
204.                         v1 = XZ[1]/CONVZXZ2;
205.
206.                         // Dibujo circulo
207.                         cvCircle(ipl,cvPoint((int)u1,(int)v1),2,color,1,8,0);
208.                         break;
209.
210.                 case CIRC:
211.
212.                         // El centro y el radio. No gusta n?s
213.                         u1 = (XZ[0]-(float)XMIN)/CONVX;
214.                         v1 = XZ[1]/CONVZXZ2;
215.                         u2 = (float)width/(2*CONVX);
216.
217.                         // Dibujo circulo
218.                         cvCircle(ipl,cvPoint((int)u1,(int)v1),(int)(u2),color,3,8,0);
219.                         break;
220.
221.                 case RECT:
222.
223.                         // Esquina inferior izq. No gusta n?s
224.                         u1 = (XZ[0]-(float)width/2-(float)XMIN)/CONVX;
225.                         v1 = XZ[1]/CONVZXZ2;
226.
227.                         // Esquina superior der. No gusta n?s
228.                         u2 = (XZ[0]+(float)width/2-(float)XMIN)/CONVX;
229.                         v2 = XZ[1]/CONVZXZ2;
```

```
230.
231.                     // Dibujo rectangulo
232.                     cvRectangle(ipl,cvPoint((int)u1,(int)v1),cvPoint((int)u2,(int)v2),color,1,8,0);
233.                     break;
234.
235.             case TEXTO:
236.
237.                     // El inicio. No gusta n?s
238.                     if (XZ!=NULL)
239.                     {
240.                             u1 = (XZ[0]-(float)XMIN)/CONVX;
241.                             v1 = XZ[1]/CONVZXZ2;
242.                     }
243.                     else
244.                     {
245.                             u1 = 100;
246.                             v1 = 230;
247.                     }
248.
249.                     // Pongo texto. No gusta n?s
250.                     cvInitFont(&font,CV_FONT_VECTOR0,0.4f,0.4f,0.0f,1,8);
251.                     if (text==NULL)
252.                     {
253.                             if (sprintf(str,"%d",width)==INVALID) printf("Error de sprintf\n");
254.                             cvPutText(ipl,str,cvPoint((int)u1,(int)v1),&font,color);
255.                     }
256.                     else
257.                             cvPutText(ipl,text,cvPoint((int)u1,(int)v1),&font,color);
258.                     break;
259.             case FLECHA:
260.
261.                     // Esquina inferior. No gusta n?s
262.                       u1 = XZ[0]/CONVX;
263.                       v1 = XZ[1]/CONVZXZ2;
264.
265.                       // Esquina superior. No gusta n?s
266.                       angulo=(XZ[4]*PI)/180;
267.                       seno=sin(angulo);
268.                       coseno=cos(angulo);
269.                       if (XZ[2]<0) XZ[2]=XZ[2]*(-1);
270.                       if (XZ[3]<0) XZ[3]=XZ[3]*(-1);
271.                       u2 = (int)floor((XZ[0]+(XZ[2]*MD*coseno))/(float)CONVX);
272.                       v2 = (int)floor((XZ[1]-(XZ[3]*MD*seno))/(float)CONVZXZ2);
273.
274.                       // Dibujo linea
275.
276.                       cvCircle(ipl,cvPoint((int)u1,(int)v1),2,CV_RGB(0,255,0),5,8,0);
277.                       cvCircle(ipl,cvPoint((int)u2,(int)v2),2,CV_RGB(255,0,0),5,8,0);
278.
279.                       cvLine(ipl,cvPoint((int)u1,(int)v1),cvPoint((int)u2,(int)v2),color,2,8,0);
280.                       break;
281.
282.
283.             default:
284.                     printf("Tipo de forma a dibujar no implementado (solo circulo o rect?gulo)\n");
```

```
285.                         break;
286.   }
287. }
288.
289.
290.
291. void DrawingXZ(IplImage *ipl,float *centro,float radio,CvScalar color,int ispace, int classID, int esRobot)     // El centro y radio son pixels (la imagen es de
     320x240)
292. {
293.   float grid[4][2];
294.   float grid_uv[4][2];
295.
296.   // Matrices H inversas
297.
298.   float invH[9];
299.
300.   float invH_xy1[3];
301.   float vector_xy[3];
302.
303.   float centro_uv[3];
304.
305.   int i,lados;
306.   float x1,z1,x2,z2;
307.
308.   // Aproximacion del circulo por una polilinea de 10 lados
309.   float poly_x[15];
310.   float poly_y[15];
311.   float poly_u[15];
312.   float poly_v[15];
313.
314.   CvFont fuente;        // Para dibujar las clases
315.   char ident[3];
316.
317.
318.   // Dibujar el grid
319.   // ---------------------------------------
320.
321.   // Matriz a utilizar en función de la cámara sobre la que realizamos la perspectiva
322.   if (ispace == 1)
323.         for (i=0; i<9; i++)
324.                invH[i] = invH_1[i];
325.   else if (ispace == 2)
326.         for (i=0; i<9; i++)
327.                invH[i] = invH_2[i];
328.   else if (ispace == 3)
329.         for (i=0; i<9; i++)
330.                invH[i] = invH_3[i];
331.   else if (ispace == 0)
332.         for (i=0; i<9; i++)
333.                invH[i] = invH_0[i];
334.
335.   // Definir las esquinas, en metros
336.   grid[0][0] = 0; grid[0][1] = 0;       // Primera esquina
337.   grid[1][0] = 320; grid[1][1] = 0;     // Segunda esquina
338.   grid[2][0] = 320; grid[2][1] = 240;   // Tercera esquina
```

```
339.   grid[3][0] = 0; grid[3][1] = 240;     // Cuarta esquina
340.
341.   // Proyectar cada una
342.   vector_xy[2] = 1;
343.   for (i=0; i<4; i++)
344.   {
345.         vector_xy[0] = grid[i][0];
346.         vector_xy[1] = grid[i][1];
347.         matrixMult4(invH,3,3,vector_xy,invH_xy1);
348.         grid_uv[i][0] = invH_xy1[0] / invH_xy1[2];
349.         grid_uv[i][1] = invH_xy1[1] / invH_xy1[2];
350.   }
351.
352.   // Representarlo
353.   cvLine(ipl,cvPoint((int)grid_uv[0][0],(int)grid_uv[0][1]),cvPoint((int)grid_uv[1][0],(int)grid_uv[1][1]),CV_RGB(0,255,0),1,4,0);
354.   cvLine(ipl,cvPoint((int)grid_uv[1][0],(int)grid_uv[1][1]),cvPoint((int)grid_uv[2][0],(int)grid_uv[2][1]),CV_RGB(0,255,0),1,4,0);
355.   cvLine(ipl,cvPoint((int)grid_uv[2][0],(int)grid_uv[2][1]),cvPoint((int)grid_uv[3][0],(int)grid_uv[3][1]),CV_RGB(0,255,0),1,4,0);
356.   cvLine(ipl,cvPoint((int)grid_uv[3][0],(int)grid_uv[3][1]),cvPoint((int)grid_uv[0][0],(int)grid_uv[0][1]),CV_RGB(0,255,0),1,4,0);
357.
358.
359.   // Dibujar el círculo
360.   // ---------------------------------------
361.   vector_xy[0] = centro[0];     // x
362.   vector_xy[1] = centro[1];     // z
363.   vector_xy[2] = 1;
364.
365.   radio=400;
366.   lados = 15;
367.
368.   // Transformar el circulo en un poligono
369.   for (i=0; i<lados; i++)
370.   {
371.         poly_x[i] = centro[0] + (radio * cos(2*PI/lados * i));
372.         poly_y[i] = centro[1] + (radio * sin(2*PI/lados * i));
373.
374.         vector_xy[0] = poly_x[i] / CONVX;
375.         vector_xy[1] = poly_y[i] / CONVZXZ2;
376.         matrixMult4(invH,3,3,vector_xy,invH_xy1);
377.         poly_u[i] = invH_xy1[0] / invH_xy1[2];
378.         poly_v[i] = invH_xy1[1] / invH_xy1[2];
379.   }
380.
381.   // Dibujarlo
382.   for (i=0; i<lados; i++)
383.   {
384.         if (i==(lados-1))
385.                cvLine(ipl,cvPoint((int)poly_u[0],(int)poly_v[0]),cvPoint((int)poly_u[i],(int)poly_v[i]),color,2,18,0);
386.         else
387.                cvLine(ipl,cvPoint((int)poly_u[i],(int)poly_v[i]),cvPoint((int)poly_u[i+1],(int)poly_v[i+1]),color,2,18,0);
388.   }
389.
390.   // Dibujar el id de la clase
391.   // ---------------------------------------
392.
393.   // Calcular su centro proyectado
```

```
394.   vector_xy[0] = centro[0] / CONVX;     // x
395.   vector_xy[1] = centro[1] / CONVZXZ2;  // z
396.   vector_xy[2] = 1;
397.   matrixMult4(invH,3,3,vector_xy,centro_uv);
398.   centro_uv[0] = centro_uv[0] / centro_uv[2];
399.   centro_uv[1] = centro_uv[1] / centro_uv[2];
400.
401.
402.
403.   sprintf(ident,"%d", classID);
404.   cvInitFont(&fuente, CV_FONT_HERSHEY_DUPLEX, 0.5, 0.5, 0, 1, 8);
405.   cvPutText(ipl,ident,cvPoint((int)centro_uv[0]+0,(int)centro_uv[1]+5),&fuente,color);
406.   if (esRobot)
407.         cvPutText(ipl,"ROBOT",cvPoint((int)centro_uv[0]-30,(int)centro_uv[1]-30),&fuente,color);
408. }
409.
410. void perspLine(IplImage *ipl, double *p1, double *p2, CvScalar color, int ispace)
411. {
412.         float p1_uv[2];
413.         float p2_uv[2];
414.
415.         // Matrices H inversas
416.         float invH[9];
417.
418.         float invH_xy1[3];
419.         float vector_xy[3];
420.
421.         int i;
422.
423.         // Matriz a utilizar en función de la cámara sobre la que realizamos la perspectiva
424.         if (ispace == 1)
425.                 for (i=0; i<9; i++)
426.                         invH[i] = invH_1[i];
427.         else if (ispace == 2)
428.                 for (i=0; i<9; i++)
429.                         invH[i] = invH_2[i];
430.         else if (ispace == 3)
431.                 for (i=0; i<9; i++)
432.                         invH[i] = invH_3[i];
433.         else if (ispace == 0)
434.                 for (i=0; i<9; i++)
435.                         invH[i] = invH_0[i];
436.
437.         // Proyectar
438.         vector_xy[2] = 1;
439.
440.         // P1
441.         vector_xy[0] = p1[0];
442.         vector_xy[1] = p1[1];
443.         matrixMult4(invH,3,3,vector_xy,invH_xy1);
444.         p1_uv[0] = invH_xy1[0] / invH_xy1[2];
445.         p1_uv[1] = invH_xy1[1] / invH_xy1[2];
446.
447.         // P2
448.         vector_xy[0] = p2[0];
```

```
449.          vector_xy[1] = p2[1];
450.          matrixMult4(invH,3,3,vector_xy,invH_xy1);
451.          p2_uv[0] = invH_xy1[0] / invH_xy1[2];
452.          p2_uv[1] = invH_xy1[1] / invH_xy1[2];
453.
454.          // Representar
455.          cvLine(ipl, cvPoint((int)p1_uv[0],(int)p1_uv[1]) , cvPoint((int)p2_uv[0],(int)p2_uv[1]) ,color,1,4,0);
456. }
```

10.1.6. timeouts.c

```
1. E #include "ispace.h"
2.
3. void timeout_redibujar(void)
4. {
5.          //GtkWidget * imagen_odometria_robot = lookup_widget(GTK_WIDGET(__window1), "imagen_odometria_robot");
6.
7.          odometry odo_ahora = copiar_odometria(__odo);                   //nos copiamos la odometria en este instante
8.
9.          GtkWidget * area_trayectoria_robot = lookup_widget(GTK_WIDGET(__window1), "area_trayectoria_robot");
10.         GtkWidget * label_posicion_x_robot = lookup_widget(GTK_WIDGET(__window1), "label_posicion_x_robot");
11.         GtkWidget * label_posicion_y_robot = lookup_widget(GTK_WIDGET(__window1), "label_posicion_y_robot");
12.         GtkWidget * label_velocidad_robot  = lookup_widget(GTK_WIDGET(__window1), "label_velocidad_robot");
13.         GtkWidget * label_angulo_robot     = lookup_widget(GTK_WIDGET(__window1), "label_angulo_robot");
14.
15.         char odo_x_char[15],odo_y_char[15];
16.         char velocidad_char[15];
17.         char angulo_char[6];
18.
19.         if(__Ipl_fondo_odometria!=NULL)
20.         {
21.
22.
23.
24.                 int x=(int)( (odo_ahora.x)/__mm_por_pixel    +     __punto_partida_robot_x_pixels);
25.                 int y=(int)( (-(odo_ahora.y))/__mm_por_pixel +     __punto_partida_robot_y_pixels);
26.
27.         //      RELLENA LA INFORMACION DE LA NAVEGACION
28.
29.                 sprintf(odo_x_char,"%.0f",odo_ahora.x);
30.                 gtk_label_set_text(GTK_LABEL(label_posicion_x_robot),odo_x_char);
31.
32.                 sprintf(odo_y_char,"%.0f",odo_ahora.y);
33.                 gtk_label_set_text(GTK_LABEL(label_posicion_y_robot),odo_y_char);
34.
35.                 sprintf(velocidad_char,"%.0f",odo_ahora.linspeed);
36.                 gtk_label_set_text(GTK_LABEL(label_velocidad_robot),velocidad_char);
37.
38.                 sprintf(angulo_char,"%.1f",odo_ahora.phi);
```

```
39.                gtk_label_set_text(GTK_LABEL(label_angulo_robot),angulo_char);
40.
41.        //      HASTA AQUI LA INFORMACION DE NAVEGADOR
42.
43.
44.                if(__Ipl_trayectoria_odometria==NULL)
45.                        __Ipl_trayectoria_odometria=cvCloneImage(__Ipl_fondo_odometria);
46.
47.
48.                if(x<__Ipl_fondo_odometria->width && x>0)               //nos aseguramos de que esté dentro de los margenes que si no error
49.                        if(y<__Ipl_fondo_odometria->height && y>0)
50.                                cvSet2D(__Ipl_trayectoria_odometria,y,x,CV_RGB(__rastro_robot.red,__rastro_robot.green,__rastro_robot.blue));
51.
52.
53.                if(__Ipl_direccion_odometria!=NULL)
54.                        cvReleaseImage(&__Ipl_direccion_odometria);
55.
56.                __Ipl_direccion_odometria = cvCloneImage(__Ipl_trayectoria_odometria);
57.
58.
59.
60.                //el color que es funcion de la velocidad
61.                cvEllipse       (__Ipl_direccion_odometria,
62.                                cvPoint(x,y),
63.                                cvSize(__alto_robot_pixels,__ancho_robot_pixels),
64.                                odo_ahora.phi,
65.                                0,360,
66.                                CV_RGB(abs(odo_ahora.linspeed/3),255-abs(odo_ahora.linspeed/3),0),
67.                                -1,
68.                                16,
69.                                0);
70.
71.                //el triangulito
72.                cvEllipse       (__Ipl_direccion_odometria,
73.                                cvPoint(x,y),
74.                                cvSize(__alto_robot_pixels,__ancho_robot_pixels),
75.                                odo_ahora.phi,
76.                                160,200,
77.                                CV_RGB(255,255,255),
78.                                -1,
79.                                16,
80.                                0);
81.
82.                //el borde
83.                cvEllipse       (__Ipl_direccion_odometria,
84.                                cvPoint(x,y),
85.                                cvSize(__diametro_maximo_robot_pixels,__diametro_maximo_robot_pixels),
86.                                odo_ahora.phi,
87.                                0,360,
88.                                CV_RGB(__borde_robot.red,__borde_robot.green,__borde_robot.blue),
89.                                0,
90.                                16,
91.                                0);
92.
93.                asociar_ipl2pixbuf(&__pixbuf_odometria  ,__Ipl_direccion_odometria);
```

```
94.
95.                cvCvtColor( __Ipl_direccion_odometria, __Ipl_direccion_odometria, CV_BGR2RGB );
96.
97.                //gtk_image_set_from_pixbuf(GTK_IMAGE(imagen_odometria_robot),__pixbuf_odometria);
98.                //printf("pintnado\n");
99.
100.                     gdk_draw_pixbuf(__pixmap_odometria,
101.                                     area_trayectoria_robot->style->white_gc,
102.                                     __pixbuf_odometria,
103.                                     0,0,
104.                                     0,0,
105.                                     gdk_pixbuf_get_width(__pixbuf_odometria),gdk_pixbuf_get_height(__pixbuf_odometria),
106.                                     1,
107.                                     0,
108.                                     0);
109.
110.                     gtk_widget_queue_draw_area      (area_trayectoria_robot,
111.                                                      0,0,
112.                                                      area_trayectoria_robot->allocation.width,area_trayectoria_robot->allocation.height);
113.
114.            }
115.
116.            printf("\r");   //si no hacemos nada en el hilo ya no vuelve a entrar, por eso hacemos eso
117.
118.    }
119.
120.
121.
122.    void timeout_modificar_controles_robot(void)
123.    {
124.            GtkWidget * robot_barra_horizontal = lookup_widget(GTK_WIDGET(__window1), "robot_barra_horizontal");
125.            GtkWidget * robot_barra_vertical   = lookup_widget(GTK_WIDGET(__window1), "robot_barra_vertical");
126.
127.            static int seguridad=0;
128.
129.            int velocidad_angular_porcentaje = __valor_joystick.velocidad_angular_porcentaje;
130.            int velocidad_lineal_porcentaje  = __valor_joystick.velocidad_lineal_porcentaje;
131.
132.            seguridad++;
133.
134.            if(seguridad>25)
135.            {
136.                    gtk_range_set_value(GTK_RANGE(robot_barra_horizontal),0);
137.                    gtk_range_set_value(GTK_RANGE(robot_barra_vertical),0);
138.                    seguridad=0;
139.            }
140.
141.            if(__valor_joystick.modificado)
142.            {
143.                    __valor_joystick.modificado = 0;
144.                    seguridad = 0;
145.                    gtk_range_set_value(GTK_RANGE(robot_barra_horizontal),velocidad_angular_porcentaje);
146.                    gtk_range_set_value(GTK_RANGE(robot_barra_vertical),velocidad_lineal_porcentaje);
147.
148.            }
```

```
149.
150.            printf(" \r");          //si no hacemos nada en el hilo ya no vuelve a entrar
151.
152.    }
153.
154.
155.    void timeout_representar_imagenes_de_servidores(void)
156.    {
157.            odometry odo_ahora;
158.            int i,a,ident_cluster,j,h;
159.            GtkWidget * imagen_camara[ZONAS_DE_DIBUJO];
160.            GdkPixbuf * pixbuf_camara[ZONAS_DE_DIBUJO];             //la imagen que plasmaremos sobre los 4 cuadrados de vista de servidores
161.            GdkPixbuf * pixbuf_camara_160_120[ZONAS_DE_DIBUJO];
162.            GdkPixbuf * im_contorno_pixbuf;
163.            IplImage  * im_contorno_ipl_temp;
164.            int representa_grid_en = -1;
165.            IplImage  * camara_temp;        // nuevo
166.            CvScalar color;                 // nuevo
167.            int ispace;             // nuevo
168.            const int array_R[]={0,  0,     0,     0,     255,     105,    125,   0,   125,   125,   0};
169.            const int array_G[]={0,  0,     255,    255,   0,   105,   125,   125,   0,   125,    125};
170.            const int array_B[]={0, 255,   0,   255,   255,   0,   0,    125,     125,     200,     0};
171.            imagen_camara[0] = lookup_widget(GTK_WIDGET(__window1), "_0_imagen_camara");
172.            imagen_camara[1] = lookup_widget(GTK_WIDGET(__window1), "_1_imagen_camara");
173.            imagen_camara[2] = lookup_widget(GTK_WIDGET(__window1), "_2_imagen_camara");
174.            imagen_camara[3] = lookup_widget(GTK_WIDGET(__window1), "_3_imagen_camara");
175.
176.            double p1[2],p2[2];
177.
178.            // Para calcular tiempo entre frame y frame
179.            struct timezone tz;
180.            struct timeval tiempo;
181.            static unsigned long t1,t2,At;
182.
183.
184.    //PINTAMOS LOS FRAMES EN DIRECTO
185.            for( i=0; i<ZONAS_DE_DIBUJO; i++)                       //para cada zona de dibujo "imagen_camara[i]" representamos su imagen
186.            {
187.                    if(__zona_dibujo[i].usada == TRUE && strcmp(__zona_dibujo[i].direccion_quien_usa,"GRID") != 0)
188.                    {
189.
190.                            if(pthread_mutex_trylock(&__sem_rx_imagenes[i]) == 0 )  //P NO BLOQUEANTE
191.                            {
192.                                    __loader_cam[i] = gdk_pixbuf_loader_new();      //Inicializamos los loader (consume mucha cpu,MAL!!)
193.
194.                                    gdk_pixbuf_loader_write(__loader_cam[i],(unsigned char *)__gchar_camara[i],__longitud_gchar_camara[i],NULL);//buffer->loader
195.                                    gdk_pixbuf_loader_close (__loader_cam[i], NULL);                                        //cierra el loader
196.                                    pixbuf_camara_160_120[i] = gdk_pixbuf_loader_get_pixbuf(__loader_cam[i]);               //loader->pixbuf
197.
198.                                    pthread_mutex_unlock(&__sem_rx_imagenes[i]);    //V SEMAFORO
199.
200.
201.                                    if( G_IS_OBJECT(pixbuf_camara_160_120[i]) && pixbuf_camara_160_120[i] != NULL )
202.                                    {
203.                                            gdk_pixbuf_ref (pixbuf_camara_160_120[i]);
```

```
204.
205.                                 pixbuf_camara[i] = gdk_pixbuf_scale_simple      (pixbuf_camara_160_120[i],
206.                                                                                         320,240,
207.                                                                                                 GDK_INTERP_BILINEAR);
208.
209.
210.                                //////AQUI SE PINTAN LOS OBJETOS EN LAS IMAGENES
211.
212.
213.                                 pixbuf2ipl(&camara_temp, pixbuf_camara[i]);
214.                                 cvConvertImage(camara_temp, camara_temp, CV_CVTIMG_SWAP_RB);
215.
216.                                 cvConvertImage(camara_temp, camara_temp, CV_CVTIMG_SWAP_RB);
217.
218.                                 //printf("__zona_dibujo[%d].direccion_quien_usa = %s\n", i,__zona_dibujo[i].direccion_quien_usa);
219.                                 if( strcmp(__zona_dibujo[i].direccion_quien_usa,"172.29.24.217") == 0)
220.                                         ispace = 1;
221.                                 else if ( strcmp(__zona_dibujo[i].direccion_quien_usa,"172.29.25.216") == 0)
222.                                         ispace = 2;
223.                                 else if ( strcmp(__zona_dibujo[i].direccion_quien_usa,"172.29.25.217") == 0)
224.                                         ispace = 3;
225.                                 else if ( strcmp(__zona_dibujo[i].direccion_quien_usa,DIR_ISPACE0) == 0)
226.                                         ispace = 0;
227.
228.
229.                                 for (j=0; j<ks[3]; j++)
230.                                 {
231.                                         // Mejor hacer el switch case con el identificador de cada clase, no con j
232.                                         // Y si es idmaxvot, poner el texto "robot"
233.                                         ident_cluster=clusterOut[j].identify%10;
234.                                         color=CV_RGB(array_B[ident_cluster],array_G[ident_cluster],array_R[ident_cluster]);
235.
236.                                         //printf("validated[%d] = %d\n", j, clusterOut[j].validated);
237.                                         if (clusterOut[j].validated == TRUE)
238.                                         {
239.                                                 DrawingXZ(camara_temp,clusterOut[j].centroid,800,color,ispace,clusterOut[j].identify, 0);
240.                                                 for (h=0; h<TAMBUFFER-1; h++)
241.                                                         {
242.                                                                 p1[0] = trayX[clusterOut[j].identify][h];
243.                                                                 p1[1] = trayZ[clusterOut[j].identify][h];
244.                                                                 p2[0] = trayX[clusterOut[j].identify][h+1];
245.                                                                 p2[1] = trayZ[clusterOut[j].identify][h+1];
246.
247.                                                                 if (p1[0] && p2[0] && p1[1] && p2[1])
248.                                                                         perspLine(camara_temp,p1,p2,color,ispace);
249.                                                         }
250.                                         }
251.                                 }
252.                                 gdk_pixbuf_unref (pixbuf_camara[i]);
253.                                 ipl2pixbuf(&pixbuf_camara[i], camara_temp);
254.                                  cvReleaseImage(&camara_temp);
255.
256.                                 // ########## fin
257.
258.                                 ////////////////////////////////////////////////
```

```
259.
260.
261.
262.                                        gtk_image_set_from_pixbuf(GTK_IMAGE(imagen_camara[i]),pixbuf_camara[i]);//pinta el pixbuf
263.
264.                                        gdk_pixbuf_unref (pixbuf_camara_160_120[i]);                            //libera los pixbuf
265.                                        gdk_pixbuf_unref (pixbuf_camara[i]);                                    //libera los pixbuf
266.
267.                                }
268.
269.                                g_object_unref (__loader_cam[i]);                                               //libera los loaders
270.                        }
271.
272.
273.                }
274.
275.                if( strcmp(__zona_dibujo[i].direccion_quien_usa,"GRID") == 0)
276.                {
277.                        representa_grid_en = i;
278.
279.                }
280.
281.
282.
283.        }
284.
285.
286.    //AHORA PINTAMOS EL GRID
287.
288.        if(__im_contorno_ipl != NULL && representa_grid_en >= 0 && representa_grid_en <= ZONAS_DE_DIBUJO)
289.        {
290.                if(pthread_mutex_trylock(&__sem_genera_grid) == 0 )     //P NO BLOQUEANTE
291.                {
292.                        im_contorno_ipl_temp = cvCloneImage(__im_contorno_ipl);
293.                        cvReleaseImage(&__im_contorno_ipl);
294.
295.                        pthread_mutex_unlock(&__sem_genera_grid);               //V SEMAFORO
296.
297.                        odo_ahora = copiar_odometria(__odo);                                            //nos copiamos la odometria en este instante
298.
299.
300.                        // Terminar de contar el tiempo entre frame y frame
301.                        gettimeofday(&tiempo,&tz);
302.                        t2 = tiempo.tv_sec * 1000000 + tiempo.tv_usec;
303.                        At = (t2-t1)/1000;
304.
305.                        filtro_particulas(im_contorno_ipl_temp,(float)At,odo_ahora);    //aplicamos el filtro de particulas(f_particulas.h)
306.
307.                        // Empezar a contar el tiempo
308.                        gettimeofday(&tiempo,&tz);
309.                        t1 = tiempo.tv_sec * 1000000 + tiempo.tv_usec;
310.
311.                        if( ipl2pixbuf(&im_contorno_pixbuf ,im_contorno_ipl_temp) )
312.                        {
313.                                if(imagen_camara[representa_grid_en] != NULL)
```

```
314.                                            gtk_image_set_from_pixbuf(GTK_IMAGE(imagen_camara[representa_grid_en]),im_contorno_pixbuf);
315.
316.                                    gdk_pixbuf_unref (im_contorno_pixbuf);
317.                            }
318.
319.                            else
320.                            {
321.                                    printf("no se pudo representar la imagen del grid xq no se pudo componer en ipl2pixbuf\n");
322.                            }
323.
324.
325.                            cvReleaseImage(&im_contorno_ipl_temp);
326.                    }
327.
328.                    else
329.                            printf("No se representa un frame del GRID, semaforo bloqueado\n");
330.
331.            }
332.
333.            printf(" \r");          //si no hacemos nada en el timer ya no vuelve a entrar
334.
335.    }
```

10.1.7. variables.h

```
1. E #if !defined( _VARIABLES_H_ )
2. #define _VARIABLES_H_
3.
4.
5.
6.
7. /*-------------------------------------------------------------------------------------*/
8. /*----------------------------FILTRO DE PARTICULAS-------------------------------------*/
9. /*-------------------------------------------------------------------------------------*/
10.
11. FILE            *fMeas,*fResult;
12. XpfCluster      cluster[MAXCLUSTER],clusterOut[MAXCLUSTEROUT];
13. float           xPartic[NPARTICMAX*NPROCESS3D];
14. int             *cParams, *cParamsO;
15. int             iter,doResult,ks[4],xParams[5];         //[nPartic,nmRelation,nParticBase,tC,tCO]
16. XpfMeas         Meas;
17. double trayX[MAXCLUSTEROUT][TAMBUFFER],trayZ[MAXCLUSTEROUT][TAMBUFFER];
18. double trayX_est[MAXCLUSTEROUT][TAMBUFFER],trayZ_est[MAXCLUSTEROUT][TAMBUFFER];
19. ////VARIABLES NECESARIAS PARA LA IDENTIFICACION
20. int idmaxVot;
21. float           centroidOr[MAXCLUSTER*NPROCESSOR],centroidIniOut[MAXCLUSTEROUT*NMEAS];
22.
23.
24. /*-------------------------------------------------------------------------------------*/
25. /*-----------------------------------GRID----------------------------------------------*/
```

```
26. /*-------------------------------------------------------------------------------------------*/
27.
28. IplImage * __im_contorno_ipl;                          //La imagen con la suma de todo lo recibido desde los servidores
29.
30.
31. /*-------------------------------------------------------------------------------------------*/
32. /*---------------------------------COLORES---------------------------------------------------*/
33. /*-------------------------------------------------------------------------------------------*/
34.
35. GdkColor __rastro_robot;                        //el color del rastro del robot
36. GdkColor __borde_robot;                         //el color del borde del robot
37.
38.
39. /*-------------------------------------------------------------------------------------------*/
40. /*----------------------------PUNTEROS A VENTANAS--------------------------------------------*/
41. /*-------------------------------------------------------------------------------------------*/
42.
43. GtkWidget * __window1;                           //ventana principal
44. GtkWidget * __window_edicion_robot;             //ventana de propiedades del robot
45. GtkWidget * __window_edicion_de_servidores;     //ventana de edicion de servidores
46. GtkWidget * __window_propiedades_odometria;     //ventana de propiedades odometria
47. GtkWidget * __window_colores_odometria;         //ventana de los colores de la odometria
48. GtkWidget * __window_edicion_joystick;          //ventana para la edicion del joystick
49.
50. /*-------------------------------------------------------------------------------------------*/
51. /*---------------------------------ODOMETRIA-------------------------------------------------*/
52. /*-------------------------------------------------------------------------------------------*/
53.
54. IplImage * __Ipl_fondo_odometria;                //imagen que guarda el fondo de la odometria
55. IplImage * __Ipl_trayectoria_odometria;          //imagen que guarda el fondo con la trayectoria
56. IplImage * __Ipl_direccion_odometria;           //imagen que guarda el fondo, trayectoria y direccion
57.
58. GdkPixbuf * __pixbuf_odometria;                 //aqui podemos pintar lo que queramos para luego representtarlo en la odometria
59.
60. GdkDrawable * __pixmap_odometria;               //esta es la imagen que se puede representar en la ventana (haremos la conversion)
61.
62. double  __ancho_fondo_mm;                       //tamaño de la imagen a representar (fondo)
63. double  __alto_fondo_mm;
64. int     __ancho_fondo_pixels;
65. int     __alto_fondo_pixels;
66.
67. float   __mm_por_pixel;                         //el numero de mm representados en cada pixel
68.
69. int     __punto_partida_robot_x_pixels;         //la posicion inicial del robot en pixels
70. int     __punto_partida_robot_y_pixels;
71. double  __punto_partida_robot_x_mm;             //la posicion inicial del robot en mm
72. double  __punto_partida_robot_y_mm;
73.
74.
75. /*-------------------------------------------------------------------------------------------*/
76. /*---------------------------------JOYSTICK--------------------------------------------------*/
77. /*-------------------------------------------------------------------------------------------*/
78.
79. int __boton_horizontal_js;              //el boton en el mando que controlará la velocidad angular
80. int __boton_vertical_js;                //el boton en el mando que controlará la velocidad lineal
```

```
81.
82. tjoystick __valor_joystick;             //estructura que guarda informacion del mando (ver estructuras.h)
83.
84. int __usando_joystick;                  //indica en cada momento si estamos usando el joystick o no
85.
86. char __ruta_joystick[255];              //contendra la ruta para el dispositivo joystick
87.
88.
89. /*-------------------------------------------------------------------------------------------*/
90. /*-----------------------------------ROBOT---------------------------------------------------*/
91. /*-------------------------------------------------------------------------------------------*/
92.
93. char __direccion_robot[17];             //guarda la direccion del robot
94.
95. int __robot_conectado;                  //nos dice si esta o no conectado el robot
96.
97. odometry __odo;                         //estructura donde guardamos la odometria del robot(ver estructuras.h)
98.
99. testado_robot __estado_deseado_robot;   //las variables que definen como deberia estar el robot en cada momento (ver estructuras.h)
100.
101.     int __paso_flechas_robot;              //valor que incrementaremos en cada toque a las flechas de movimiento
102.     int __max_vel_lineal;
103.     int __max_vel_angular;
104.
105.     double __ancho_robot_mm;               //tamaño en mm del robot
106.     double __alto_robot_mm;
107.
108.     int __ancho_robot_pixels;              //tamaño en pixels del robot
109.     int __alto_robot_pixels;
110.     int __diametro_maximo_robot_pixels;    //maximo tamaño (alto/ancho) del robot (nos sirve para la circunferencia que describe)
111.
112.     //float linspeed,rotspeed;             //Variables globales control robot
113.
114.
115.     /*-------------------------------------------------------------------------------------------*/
116.     /*-----------------------------------ENTORNO-------------------------------------------------*/
117.     /*-------------------------------------------------------------------------------------------*/
118.
119.     int __estado_actual;           //el estado en el que nos encontramos(estados definidos en constantes.h)
120.
121.
122.     /*-------------------------------------------------------------------------------------------*/
123.     /*-----------------------------------CAMARA--------------------------------------------------*/
124.     /*-------------------------------------------------------------------------------------------*/
125.
126.     int __resolucion_escogida;                    //la resolucion que aplicaremos a todas las camaras
127.     int __fps_escogida;                           //los fps que aplicaremos a todas las camaras
128.
129.     /*-------------------------------------------------------------------------------------------*/
130.     /*-----------------------------------SERVIDORES----------------------------------------------*/
131.     /*-------------------------------------------------------------------------------------------*/
132.
133.     tservidor * __cabecera_servidores;             //la cabecera de los servidores (lista enlazada) (ver estructuras.h)
134.
135.     char __direc_servi[MAX_SERVIDORES][17];        //array donde guardaremos las direccion de los servidores
```

```
136.
137.
138.    //int __puerto_comandos[MAX_SERVIDORES];              //array con los puertos de los servidores para comandos
139.    //int __puerto_datos[MAX_SERVIDORES];          //array con los puertos de los servidores para datos
140.
141.    tcomando __comando_enviado[MAX_SERVIDORES];     //el comando
142.
143.    tdispositivo __dispositivos[MAX_SERVIDORES];
144.
145.    int __num_servidores;                           //numero de servidores al que nos vamos a conectar(finalmente guarda el nº de servidores conectados)
146.
147.    int __sock_comandos[MAX_SERVIDORES];    // Sockets de comandos
148.    int __sock_datos[MAX_SERVIDORES];               // Sockets de datos
149.    int __sock_imagenes[MAX_SERVIDORES];    // Sockets para tx imagenes
150.    int __sock_grid[MAX_SERVIDORES];                // Sockets para recibir el contorno
151.
152.    char            * __gchar_camara          [ZONAS_DE_DIBUJO];
153.    int               __longitud_gchar_camara [ZONAS_DE_DIBUJO];
154.    GdkPixbufLoader * __loader_cam            [ZONAS_DE_DIBUJO];    //los loaders para cada camara
155.    tzona_dibujo      __zona_dibujo           [ZONAS_DE_DIBUJO];    //variable que nos indica que zonas de dibujo de las imagenes son utilizadas
156.
157.
158.    /*-----------------------------------------------------------------------------------------*/
159.    /*-------------------------------------HILOS-----------------------------------------------*/
160.    /*-----------------------------------------------------------------------------------------*/
161.
162.    pthread_t __hilo_sincronizacion_robot_camara;                           //hilo encargado de sincronizar las camaras y la odometria
163.    pthread_t __hilo_sondeo_joystick;                                       //hilo que sondea continuamente el joystick
164.    pthread_t __hilo_recepcion_imagenes_de_servidores[ZONAS_DE_DIBUJO];     //hilo encargado de la recepcion de imagenes de los servidores
165.    pthread_t __hilo_robot;                                                 //hilo encargado de controlar el robot(pide odometria y manda ordenes)
166.
167.    /*-----------------------------------------------------------------------------------------*/
168.    /*-------------------------------------TIMERS----------------------------------------------*/
169.    /*-----------------------------------------------------------------------------------------*/
170.
171.    int __timer_redibujar;                          //el timer encargado de redibujar la odometria
172.    int __timer_modificar_controles_robot;          //timer encargado de modificar los controles del robot debido al joystick
173.    int __timer_representar_imagenes_de_servidores; //encargado de representar las imagenes que lleguen desde los servidores
174.
175.
176.    /*-----------------------------------------------------------------------------------------*/
177.    /*-------------------------------------SEMAFOROS-------------------------------------------*/
178.    /*-----------------------------------------------------------------------------------------*/
179.
180.    pthread_mutex_t __sem_rx_imagenes[ZONAS_DE_DIBUJO];
181.    pthread_mutex_t __sem_genera_grid;
182.    pthread_mutex_t __sem_fichero_eventos;
183.    pthread_mutex_t __sem_estado_servidores;
184.
185.
186.    #endif
```

10.1.8. estructuras.h

```
1. E #if !defined( _ESTRUCTURAS_H_ )
2. #define _ESTRUCTURAS_H_
3.
4. // Estructura de las zonas de dibujo
5. typedef struct _zona_dibujo_            //se generara una lista enlazada que contendrá todos los servidores, tanto activos como no.
6. {
7.         int usada;                      //indica si la zona de dibujo está siendo o no usada por algun servidor
8.         char direccion_quien_usa[17];   //la direccion del servidor que esta usando esa zona de dibujo
9.         int num_camara;
10.
11.} tzona_dibujo;
12.
13.
14.// Estructuras del filtro de particulas
15.typedef struct
16.{
17.        float y[NVISION*NMEAS3D];
18.        int ny;
19.}XpfMeas;
20.
21.
22.typedef struct
23.{
24.        float xmeas[NPROCESSLOC*TAMBUFFER];
25.        float xest[NPROCESSLOC*TAMBUFFER];
26.        float vang[TAMBUFFER];
27.        float vlineal[TAMBUFFER];
28.        float J[NPROCESSLOC*NPROCESSOD*TAMBUFFER];
29.}XpfOrient;
30.
31.
32.typedef struct
33.{       int identify;
34.        int candidate;
35.        int validated;
36.        int nMembers;
37.        float probability;
38.        float centroid[NPROCESS3D];
39.        float members[NPARTICMAX*NPROCESS3D];
40.}XpfCluster;
41.
42.
43.// Estructura de servidor
44.typedef struct _servidor_               //se generara una lista enlazada que contendrá todos los servidores, tanto activos como no.
45.{
46.        int activo;                     //para saber si a ese servidor hay que conectarse o no (en la ventana tendrá o no 'v')
```

```
47.         char direccion[17];             //la direccion del servidor
48.         int port_comandos;              //el puerto de comandos del servidor (debería ser igual para todos 'PUERTO_COMANDOS_PARA_TODOS_IGUAL')
49.         int port_datos;                 //el puerto de datos del servidor (debería ser igual para todos 'PUERTO_DATOS_PARA_TODOS_IGUAL')
50.         struct _servidor_ * next;       //para apuntar al siguiente servidor (el ultimo tiene NULL)
51.
52. } tservidor;
53.
54. // Estructura de comando
55. typedef struct _comando_
56. {
57.         short tipo_comando;             //el tipo de comando que le estamos mandando al servidor
58.
59.                                                 // DEFINIDAS EN 'constantes.h'
60.                                                 /*      #define ENVIAR_VIDEO_COMPLETO   20
61.                                                         #define LOCALIZAR               0
62.                                                         #define CALIBRAR                1
63.                                                         #define CAPTURAR                2
64.                                                         #define DESCONECTAR             3
65.                                                         #define ERROR_COMANDO           4
66.                                                         #define NO_ERROR_COMANDO        5
67.
68.                                                         #define CAPTURAR_FRAME          6
69.                                                         #define CAPTURAR_GRABAR_FRAME   7
70.                                                         #define FRAME_CAPTURADO         8
71.                                                         #define SALIR_CAPTURA           9       */
72.
73.         short transmite_imagenes;       //si el servidor tiene o no que mandar imagenes al cliente (TRUE o FALSE)
74.         short calidad_imagen;           //la calidad con lo que los servidores nos mandan las imagenes
75.         short ack;                      //si se ha recibido todo correcto el valor sera 'NO_ERROR_COMANDO'
76.         short fps;                      //los fps a los que debe ir la camara
77.         short resolucion;               //la resolucion a la que capturamos
78.         int numImagenes;                //lleva la cuenta de las imagenes grabadas por el servidor
79.         int puerto_imagenes;            //el puerto asignado al servidor para que tx las imagenes
80.         short num_camara;               //el numero de camara del servidor al que va dirigido el comando
81.
82. } tcomando;
83.
84. // Estructura de los dispositivos
85. typedef struct _dispositivo_
86. {
87.         int numPuertos;
88.         int numCamaras;
89.         int error;
90.
91. } tdispositivo;
92.
93.
94. // Odometria del robot
95. typedef struct
96. {
97.         double linspeed;        //velocidad lineal
98.         double rotspeed;        //velocidad angular
99.         double x;               //posicion x respecto la posicion inicial
100.            double y;               //posicion y respecto la posicion inicial
101.            double phi;             //angulo respecto la posicion inicial
```

```
102.            double num_orden;       //el numero del frame al que corresponde la odometria del robot cuando grabamos
103.
104.    }odometry;
105.
106.
107.    // el estado del robot
108.    typedef struct _estado_robot_
109.    {
110.            int desconectar;                        //a 1 si hay que desconectarlo
111.            double linspeed;                        //el valor que debe tener la velocidad lineal
112.            double rotspeed;                        //el valor que debe tener la velocidad angular
113.            double x;                                       //la posicion x en la que debe estar
114.            double y;                                       //la posicion y en la que debe estar
115.            double phi;                             //el angulo que debe tener
116.
117.    } testado_robot;
118.
119.
120.    // estructura para joystick
121.    typedef struct _joystick_
122.    {
123.            struct js_event estructura;             //estructura con valores del mando(ya hecha en el nucleo)
124.
125.                                                            /*struct js_event {
126.                                                                    __u32 time;     // event timestamp in milliseconds
127.                                                                    __s16 value;    // value
128.                                                                    __u8 type;      // event type
129.                                                                    __u8 number;    // axis/button number
130.                                                            };*/
131.
132.            int modificado;                         //si hemos modificado el valor de velocidad (ahorramos tiempo)
133.            int velocidad_angular_porcentaje;       //la velocidad que queremos
134.            int velocidad_lineal_porcentaje;        //la velocidad que queremos
135.
136.    } tjoystick;
137.
138.    #endif
```

10.2. Seguimiento tridimensional

10.2.1. main3DPF.m

```
 function main3DPF(file,plotOn,videoOn,saveOn)
% Initialization for XPFCP execution.
% Author  : Marta Marron Romera and Álvaro MArcos
% Date    : 12-06-06 / 12-12-06
% Sintaxis: mainXPFCPsinBuff(file,plotOn,videoOn,saveOn);
% Inputs  : - file: The path of the datos file to segment
%           - plotOn: If plotting XPF execution info (pdf,weights,etc) set to 1... time consuming?
%           - videoOn: If plotting in the image is wanted set to 1... very time consuming
%           - saveOn: If wanted to store the results in a video set to 1... very time consuming
% Outputs :  All info generated is stored in a file named resutltXPFCP.mat
% K number of clusters,and nY is almost-half of number of states (3=>X,Z,Y of 5=>X,Z,Y,VX,VZ)
% CAREFUL: WE ARE ALWAYS WORKING ONLY WITH X AND Z. Y IS ONLY USED TO PLOT IN THE IMAGE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

warning off MATLAB:colon:operandsNotRealScalar;
warning off MATLAB:divideByZero;

%====================================
% Global variables
%====================================
global color_
color_=('.r.m.b.c.y.g.k.r.m.b.c.y.g.k.r.m.b.c.y.g.k');
global color2
color2=('-r-m-b-c-y-g-k-r-m-b-c-y-g-k-r-m-b-c-y-g-k');
global espacio
espacio=[-1000 3000 -1000 3000 0 2000]; % xmin xmax zmin zmax ymin ymax de la habitacion

%====================================
% Init
%====================================

% ks: array of info about the number of classes [kOut,kOutValid,kOut3D,kOut3DValid]
ks=[0 0 0 0];

% cluster: K-1 array of structures of info about the output classes TO MODIFY IN 3D
%       * I: number to identify a cluster and to do clustering association over time
%       * C: 1-nY/1-nX array with the state variables (the position in xz
%       plane) of the centroid
%       * P: number in %1 presenting the cluster probability
%       * candidate: number presenting if the cluster is a candidate
%       * validated: to 1 if the cluster is validated,to 0 otherwise
%       * M: nM-1 cluster M
%       * nM: number informing about the number cluster M
clusterOut=[];
```

```
IOut=[]; COut=[]; POut=[]; canOut=[]; valOut=[]; nmOut=[];

% xPred,xPredTotal: nN-nX array with the particles value after the pred/corr step,before resampling
% wPred,wPredTotal: nN-1 array with the particles' weigh after the pred/corr step,before resampling
% x,xTotal: nN-nX array with the particles value at the end of the XPF
xPredTotal=[];
wPredTotal=[];
xTotal=[];
x=[];

% SeffTPTotal: num partic eff,wNews: weights of partic inserted at reinit
SeffTPTotal=[];
wNews=[];

% Timings
tTotal=[];

%====================================
% Inizializing param-structs
%====================================

% Initializing Sistema Params. - Sistema: Stucture with system information
%       * y: Array of measurements [x,z,y]
%       * Params: Params of the system
%           - nX: Number of states of the estimated obstacle [x,z,y,vz,vx]
%           - nY3D: Number of elements of the observation vector [x,z,y]
%           - cX: Standard deviation (NO Variance) of the Gaussian process noise 1e-3. SEE IN C
%           - cY: Standard deviation (NO Variance) of the Gaussian measurement noise 1e-3. SEE C
Sistema.Params=[5 3 20 20];   %Sistema.Params=[5 3 100 150];

% Initializing Algorithm Params Array - AlgParams
%   * 3DPF: Params of the Filter
%       - iter: the state variable of the XPF state machine
%       - nN: (200) Number total of particles
%       - nM: (30) Number of samples inserted directly from observation in the pdf reinit %
%       - nB: (>=.nP) Number of meas can be in the buffer. NOT USEd. used to reinforce init %
%       - rS: possible choices: systematic sampling (2),residual (1),and multinomial (3) NORMAL 1
%       - distM: DIFFERENT USES. Fixed here to 8*Sistema.cY; (10*) THINK AND SEE
%   * Eq: Params of the Meas Equalization
%       - regionesHist: nº of cells when gridding the space. Should be multiple of sizeSpace/tamCelda
%       - umbral: min nº meas to clean each hist cell
%       - porcentaje: %1 in which the cell to clean is in fact cleaned
%   * Weigh: Params of the particle weighting
%       - radio: (200mm) Radio dentro del cual se buscan medidas alrededor
%       de partic
%       - variacionAltura: (40) Tamaño del grid
%       - algoritmo: 1: Cúbico, 2: Esférico
%       - porcentajeSup: (20) Porcentaje de saturación de pesos superior
%       - porcentajeInf: (0) Porcentaje de saturación de pesos inferior
iter=0;
distM=650;
AlgParams.TDPF=[iter 1500 20 10 3 distM^2]; %
AlgParams.Eq=[150 40 0.05 100];             %
AlgParams.Weigh=[200 40 1 20 0 1];          %
```

```
% Initializing Clases Array - Clases: Structure with cluster inform. SIN SUBTRACTIVE
%       * in: Params of the equalization clustering
%           - nXY: Lenght of the data vector to cluster,can be nX or nY,or a mixeture
%           - iterM: Num max of iterations at the clustering algorithm NOT NEEDED IN SUBTRACT
%           - canM: NUMBER of iter. that cluster is invalid/valid before valida/invalid
%           - hist: histheresys to valid/invalid due to cluster lik | dist to pred centroide
%           - factOlv: forgeting factor in the cluster likelihood
%           - kLikM: normalizing factor to calc cluster lik according to the number clusters
%           - distM: DIFFERENT USES. Fixed here to 8*Sistema.cY; (10*) THINK AND SEE
%           - um: threshold to validate the cluster according to its likelihood
%           - distM2: DIFFERENT USES. MAINLY TO PLOT. without 2exp.
%           - nMmin: DIFFERENT USES. only for clusterIn
%       * out: IDEM THAN IN but for particles
%       * TD: Params for the connectivity algorithm
%           - distM3: Connectivity distance
%           - pond_xz: Distance factor in the XZ coords
%           - pond_y: Distance factor in the Y coord
%       * type: For in, out and 3D [typeIn,typeOut,type3D]. choices: "tree"(0),kMeans (1),subtractive (2)
distM2=350;
distM3=600;
Clases.in=[Sistema.Params(2) 100 0 0 0 0 800^2 0 distM2 300];
Clases.out=[Sistema.Params(2) 100 2 0.2 0.2 10 700^2 0.3 distM2 0];
Clases.TD=[distM3 1.1 1.3];
Clases.type=[1 1 0];

%====================================
% Mientras haya datos
%====================================

% Vbles a inicializar
nPartic=0;
nParticTotal=[];

% 1/FPS,cada cuanto hay nuevos datos... solo para actualizar velo
% No pongo At,seria para ejecutar tiempo real,pero va mas despacio. Tal y como esta simulo real
ts=1/15;

% Lazo de ejecution. FlagVideo va a valer para quitar solo las primeras iters
iterVideo=1;

%====================================
% Recojo datos y ejecuto
%====================================

% Specifying s short piece of the video %for iterVideo=400:660
for iterVideo=410:660
    if (iterVideo<414) | (iterVideo>483) %     if iterVideo > 483

        % Recojo datos
        datFile=sprintf('..\\datos_filtro\\frame0%d.dat',iterVideo);      % Datos a leer
        Sistema.y=load(datFile);
        nMeas=length(Sistema.y');

        % Empobrecer medidas para simular un mal sensado
%         objeto=1;         % Objeto sobre el que se van a empobrecer
```

```
%         porcentaje=30;    % Grado de empobrecimiento
%         iteracion=5;      % Grado de empobrecimiento
%         plotOn=1;         % Mostrar resultados activados / desactivado
%         drawVoxels=1;     % Dibujar zona donde se empobrecen activado / desactivado
%         Sistema.y=empobrecer_medidas(iterVideo,Sistema.y,objeto,porcentaje,iteracion,plotOn,drawVoxels);

        % Ejecutando lazo
        AlgParams.TDPF(1)=iter;
        tic
        if iter~=-1

            % Ejecutando filtro
            [xPred,wPred,x,clusterOut3D,clusterOut,ks]=TDPF(Sistema,AlgParams,Clases,x,clusterOut,ks,ts,plotOn);

            % For reliability test
            frame(iter+1).ifile=iterVideo
            frame(iter+1).nClases=clusterOut3D
            save frame frame

        end
        iter=iter+1;
        nPartic=size(x,1);

        %====================================
        % storing results
        %====================================

        % Time
        tTotal=[tTotal;toc];
        texto=sprintf('Texe=%f',tTotal(length(tTotal)));
        disp(texto)

        % Particles and weiths
        nParticTotal(end+1,1)=nPartic;
        nParticTotal(end,2)=length(wPred);
        if isempty(nParticTotal)==0
            wNews=[wNews;sum(wPred(nParticTotal(end,1)+1:end))];
        end
        if isempty(x)==0
            xTotal=[xTotal;x];
        end
        xPredTotal=[xPredTotal;xPred];
        wPredTotal=[wPredTotal;wPred];
        SeffTP=(1/sum(wPred.*wPred))*100/nParticTotal(end,2);
        SeffTPTotal=[SeffTPTotal;SeffTP];
        texto=sprintf('Número eficaz de partículas=%f',SeffTP);
        disp(texto)

        %====================================
        % Mas datos
        %====================================

        iterVideo=iterVideo+1;
        pause(0.01);
    end
```

```
end

%====================================
% Saliendo de la ejecucion del XPF
%====================================

% Guardando results
save resultXPFCP wNews *Total *Out;
```

10.2.2. empobrecer_medidas.m

```
function y=empobrecer_medidas(i,y,objeto,porcentaje,iteracion,plotOn,drawVoxels)
% ========================================================================
% Función empobrecer_medidas
% Autor: Alvaro Marcos 14/04/09
% Descripción: Empobrece artificialmente las medidas. Sólo vale para un
% video en concreto
% ------------------------------------
% Parámetros de entrada
%     y: medidas de entrda de la escena,previamente abiertas con y=load(File)
%     objeto:   Robot:1
%               Persona:2
%               Papelera1:3
%               Papelera2:4
%               Papelera3:5
%     porcentaje: probabilidad de que en cada iteración la medida sea borrada
%     iteracion: número de veces que se tratan las medidas del objeto para
%                borrarlas
%     plotOn: Si se pone a 1,se imprimen
%     drawVoxels: A 1 se dibujan
% Parámetros de salida
%     y: Medidas empobrecidas
% ========================================================================

% Frames válidas dentro del vídeo (el resto están rotas)
% for i=101:660
    if i<414 | i>483

        % ======================================
        % REPRESENTAR LAS MEDIDAS ** PLOTTING
        % ======================================

        if plotOn
            figure(20); clf;

            % Vista XY ?? y pq no pintas solo XY sin plot3D+view
            subplot(221)
            plot3(y(:,1),y(:,2),y(:,3),'g.');
            axis([-1000 3000 -1000 3000 0 2000]);
            xlabel('x in mm'); ylabel('z in mm'); zlabel('y in mm');
            titulo=sprintf('Representación XY. Frame %d',i);
```

```
title(titulo); grid on; view(0,0);
% Vista ZY ??
subplot(222)
plot3(y(:,1),y(:,2),y(:,3),'g.');
axis([-1000 3000 -1000 3000 0 2000]);
xlabel('x in mm'); ylabel('z in mm'); zlabel('y in mm');
titulo=sprintf('Representación ZY. Frame %d',i);
title(titulo); grid on; view(-90,0);

% Vista XZ
subplot(223)
plot(y(:,1),y(:,2),'g.');
axis([-1000 3000 -1000 3000]); xlabel('x in mm'); ylabel('z in mm');
titulo=sprintf('Representación XZ. Frame %d',i);
title(titulo); grid on; %view(2);
% Vista 3D
subplot(224)
plot3(y(:,1),y(:,2),y(:,3),'g.');
axis([-1000 3000 -1000 3000 0 2000]);
xlabel('x in mm'); ylabel('z in mm'); zlabel('y in mm');
titulo=sprintf('Representación 3D. Frame %d',i);
title(titulo); grid on;

% ======================================
% DEFINIR LIMITES DE LOS OBJETOS ** PLOTTING
% ======================================

if drawVoxels

    % Robot (rojo)
    % ======================================
    if objeto==1

        % Entre frames 101 y 318:
        %   X: -860,1500
        %   Z: -190,2300
        %   Y: 0,800
        if i<318
            subplot(221)
            voxel([-890 -190 0],[2390 2490 700],'r',0.5);
            subplot(222)
            voxel([-890 -190 0],[2390 2490 700],'r',0.5);
            subplot(223)
            voxel([-890 -190 0],[2390 2490 700],'r',0.5);
            subplot(224)
            voxel([-890 -190 0],[2390 2490 700],'r',0.5);
        end

        % Entre frames 318 y 525:
        %   X: -860,1500
        %   Z: -190,2300
        %   Y: 0,800
        elseif i>=318 & i<525
            subplot(221)
            voxel([-500 500 0],[750 600 700],'r',0.5);
```

```
        subplot(222)
        voxel([-500 500 0],[750 600 700],'r',0.5);
        subplot(223)
        voxel([-500 500 0],[750 600 700],'r',0.5);
        subplot(224)
        voxel([-500 500 0],[750 600 700],'r',0.5);
    end
end

% Papelera 1 (amarillo)
% =====================================
elseif objeto==3
    if i>325 & i<660
        subplot(221)
        voxel([-350 -200 0],[400 280 350],'y',0.5);
        subplot(222)
        voxel([-350 -200 0],[400 280 350],'y',0.5);
        subplot(223)
        voxel([-350 -200 0],[400 280 350],'y',0.5);
        subplot(224)
        voxel([-350 -200 0],[400 280 350],'y',0.5);
    end
end

% Papelera 2 (magenta)
% =====================================
elseif objeto==4
    if i>335 & i<660
        subplot(221)
        voxel([800 300 0],[400 280 350],'m',0.5);
        subplot(222)
        voxel([800 300 0],[400 280 350],'m',0.5);
        subplot(223)
        voxel([800 300 0],[400 280 350],'m',0.5);
        subplot(224)
        voxel([800 300 0],[400 280 350],'m',0.5);
    end
end

% Papelera 3 (cyan)
% =====================================
elseif objeto==5
    if i>355 & i<=660
        subplot(221)
        voxel([350 1850 0],[400 280 350],'c',0.5);
        subplot(222)
        voxel([350 1850 0],[400 280 350],'c',0.5);
        subplot(223)
        voxel([350 1850 0],[400 280 350],'c',0.5);
        subplot(224)
        voxel([350 1850 0],[400 280 350],'c',0.5);
    end
end

% Persona 3 (azul)
```

```
        % ====================================

        elseif objeto==2
            if i>318 & i<353
                subplot(221)
                voxel([-700+((i-318)*30) -1000+((i-318)*45) 0],[1100 900 1500],'b',0.5);
                subplot(222)
                voxel([-700+((i-318)*30) -1000+((i-318)*45) 0],[1100 900 1500],'b',0.5);
                subplot(223)
                voxel([-700+((i-318)*30) -1000+((i-318)*45) 0],[1100 900 1500],'b',0.5);
                subplot(224)
                voxel([-700+((i-318)*30) -1000+((i-318)*45) 0],[1100 900 1500],'b',0.5);
            end

            if i>=353 & i<368
                subplot(221)
                voxel([350 590+((i-353)*55) 0],[1100 1100 1500],'b',0.5);
                subplot(222)
                voxel([350 590+((i-353)*55) 0],[1100 1100 1500],'b',0.5);
                subplot(223)
                voxel([350 590+((i-353)*55) 0],[1100 1100 1500],'b',0.5);
                subplot(224)
                voxel([350 590+((i-353)*55) 0],[1100 1100 1500],'b',0.5);
            end

            if i>=368 & i<393
                subplot(221)
                voxel([350+((i-368)*70) 1415 0],[1100 900 1500],'b',0.5);
                subplot(222)
                voxel([350+((i-368)*70) 1415 0],[1100 900 1500],'b',0.5);
                subplot(223)
                voxel([350+((i-368)*70) 1415 0],[1100 900 1500],'b',0.5);
                subplot(224)
                voxel([350+((i-368)*70) 1415 0],[1100 900 1500],'b',0.5);
            end

            if i>=400 & i<420
                subplot(221)
                voxel([2205-((i-400)*80) 1400-((i-400)*68) 0],[1200 1100 1500],'b',0.5);
                subplot(222)
                voxel([2250-((i-400)*80) 1400-((i-400)*68) 0],[1200 1100 1500],'b',0.5);
                subplot(223)
                voxel([2250-((i-400)*80) 1400-((i-400)*68) 0],[1200 1100 1500],'b',0.5);
                subplot(224)
                voxel([2250-((i-400)*80) 1400-((i-400)*68) 0],[1200 1100 1500],'b',0.5);
            end
        end
    end
end

% ====================================
% ELIMINAR EL PORCENTAJE DESEADO DE MEDIDAS
% ====================================

% Formar el array con las medidas que queremos reducir
```

```
        k=1; j=1;
        nMeas=size(y,1);

        % Robot (rojo)
        % =====================================
        if objeto==1
            iteracion=1;
            while iteracion<5
                if i<318
                    for j=1:size(y,1)
                        if y(j,1)>-890 & y(j,1)<1500    % en X
                            if y(j,2)>-190 & y(j,2)<2300% en Z
                                if rand()<porcentaje    %
                                    y(j,:)=[];
%                                     borradas=borradas +1;
                                end
                            end
                        end
                    end
                end

                elseif i>=318 & i<525
                    for j=1:size(y,1)
                        if y(j,1)>-500 & y(j,1)<1250    % en X
                            if y(j,2)>500 & y(j,2)<1100 % en Z
                                if rand()<porcentaje
                                    y(j,:)=[];
 %                                    borradas=borradas +1;
                                end
                            end
                        end
                    end
                end
                iteracion=iteracion+1;
            end
        end

        % Papelera 1 (amarillo) !! ojo aqui te falta el for!!!
        % =====================================
        elseif objeto==3
            if i>325 & i<660
                if y(j,1)>-350 & y(j,1)<750     % en X
                    if y(j,2)>-200 & y(j,2)<480 % en Z
                        if rand()<porcentaje
                            y(j,:)=[];
%                             borradas=borradas +1;
                        end
                    end
                end
            end
        end

        % Papelera 2 (magenta)
        % =====================================
        elseif objeto==4
```

```
             if i>335 & i<660
                  for j=1:size(y,1)
                    if y(j,1)>-350 & y(j,1)<750     % en X
                        if y(j,2)>-200 & y(j,2)<480 % en Z
                            if rand()<porcentaje
                                y(j,:)=[];
%                                borradas=borradas +1;
                            end
                        end
                    end
                end
            end
        end

        % Papelera 3 (cyan)
        % =====================================
        elseif objeto==5
            if i>355 & i<660
                for j=1:size(y,1)
                    if y(j,1)>350 & y(j,1)<750      % en X
                        if y(j,2)>1850 & y(j,2)<2130% en Z
                            if rand()<porcentaje
                                y(j,:)=[];
%                                borradas=borradas +1;
                            end
                        end
                    end
                end
            end
        end

        % Persona 3 (azul)
        % =====================================
        elseif objeto==2

            if i>318 & i<353
                for j=1:size(y,1)                       % !! esto mas dificil hacer con find
                    x_inf=-700+((i-318)*30);
                    x_sup=x_inf+1100;
                    if y(j,1)>x_inf & y(j,1)<x_sup      % en X
                        z_inf=-1000+((i-318)*45);
                        z_sup=z_inf+900;
                        if y(j,2)>z_inf & y(j,2)<z_sup  % en Z
                            if rand()<porcentaje
                                y(j,:)=[];
%                                borradas=borradas +1;
                            end
                        end
                    end
                end
            end

            elseif i>=353 & i<368
                for j=1:size(y,1)
                    if y(j,1)>350 & y(j,1)<1450         % en X
```

```
                        z_inf=590+((i-353)*55);
                        z_sup=z_inf+1100;
                        if y(j,2)>z_inf & y(j,2)<z_sup  % en Z
                            if rand()<porcentaje
                                y(j,:)=[];
%                                borradas=borradas +1;
                            end
                        end
                    end
                end

                elseif i>=368 & i<393
                    for j=1:size(y,1)
                        x_inf=350+((i-368)*70);
                        x_sup=x_inf+1100;
                        if y(j,1)>x_inf & y(j,1)<x_sup  % en X
                            if y(j,2)>1415 & y(j,2)<2315% en Z
                                if rand()<porcentaje
                                    y(j,:)=[];
%                                    borradas=borradas +1;
                                end
                            end
                        end
                    end
                end

                elseif i>=400 & i<420
                    for j=1:size(y,1)
                        x_inf=2205-((i-400)*80);
                        x_sup=x_inf+1200;
                        if y(j,1)>x_inf & y(j,1)<x_sup      % en X
                            z_inf=1400-((i-400)*68);
                            z_sup=z_inf+1100;
                            if y(j,2)>z_inf & y(j,2)<z_sup   % en Z
                                if rand()<porcentaje
                                    y(j,:)=[];
%                                    borradas=borradas +1;
                                end
                            end
                        end
                    end
                end
            end

            % ======================================
            % plotting end
            % ======================================

            if plotOn
                figure(21)
                plot3(y(:,1),y(:,2),y(:,3),'b.');
                axis([-1000 3000 -1000 3000 0 2000]);
                grid on;
            end
```

```
        mostrar=sprintf('nMeas final=%d',size(y,1));
        disp(mostrar);
    end
% end
```

10.2.3. TDPF.m

```
 function [xParticPred,wParticPred,xPartic,clusterOut3D,clusterOut,kOut]=...
            TDPF(Sistema,AlgParams,Clases,xPartic,clusterOut,kOut,ts,plotOn)
% TDPF execution. Modified by Marta Marron Romera
% Author  : Alvaro Marcos
% Date    : 12-03-09 / 12-05-09
% Sintaxis: [xParticPred,wParticPred,xPartic,clusterOut3D,clusterOut,kOut]=...
%               TDPF(Sistema,AlgParams,Clases,xPartic,clusterOut,kOut,ts,plotOn)
% Inputs  :
%   - Sistema: Stucture with system information
%       * y: Array of measurements [x,z,y]
%       * Params: Params of the system
%           - nX: Number of states of the estimated obstacle [x,z,y,vz,vx]
%           - nY: Number of elements of the observation vector [x,z,y]
%           - cX: Standard deviation (NO Variance) of the Gaussian process noise 1e-3. SEE IN C
%           - cY: Standard deviation (NO Variance) of the Gaussian measurement noise 1e-3. SEE C
%   - Algorithm Params Array - AlgoParams
%       * TDPF: Params of the Filter
%           - iter: the state variable of the XPF state machine
%           - nN: (200) Number total of particles
%           - nM: (30) Number of samples inserted directly from observation in the pdf reinit %
%           - nB: (>=.nP) Number of meas can be in the buffer. NOT USEd. used to reinforce init %
%           - rS: possible choices: systematic sampling (2),residual (1),and multinomial (3) NORMAL 1
%           - distM: DIFFERENT USES. Fixed here to 8*Sistema.cY; (10*) THINK AND SEE
%       * Eq: Params of the Meas Equalization
%           - regionesHist: nº of cells when gridding the space. Should be multiple of sizeSpace/tamCelda
%           - umbral: min nº meas to clean each hist cell
%           - porcentaje: % in which the cell to clean is in fact cleaned
%       * Weigh: Params of the particle weighting
%           - radio: (200mm) Radio dentro del cual se buscan medidas
%           alrededor de partic
%           - variacionAltura: (40) Tamaño del grid
%           - algoritmo: 1: Cúbico, 2: Esférico
%           - porcentajeSup: (20) Porcentaje de saturación de pesos
%           superior
%           - porcentajeInf: (0) Porcentaje de saturación de pesos inferior
%   - Clases: Structure with clustering information
%       * out: Params of the particles clustering
%           - nXY: Lenght of the data vector to cluster,can be nX or nY,or a mixeture
%           - iterM: Num max of iterations at the clustering algorithm NOT NEEDED IN SUBTRACT
%           - canM: NUMBER of iter. that cluster is invalid/valid before valida/invalid
%           - hist: histheresys to valid/invalid due to cluster lik | dist to pred centroide
%           - factOlv: forgeting factor in the cluster likelihood
%           - kLikM: normalizing factor to calc cluster lik according to the number clusters
%           - distM: DIFFERENT USES. Fixed here to 8*Sistema.cY; (10*) THINK AND SEE
```

```
%           - um: threshold to validate the cluster according to its likelihood
%           - distRM: (subtract) Enable area for cluster intersection. 8*Sistema.cY? THINK AND SEE
%           - umOutliers: (subtract) pdf threshold to create new cluster
%       * 3D: IDEM THAN PARAMS but for "cluster clustering"
%       * type: For 3D and output [type3D,typeOut]. choices: kMeans (1),subtractive (2)
%   - xPartic: nN-nX array with the particles value at the end of the XPF
%   - clusterOut: K-1 array of structures of info about the output classes
%       * I: number to identify a cluster and to do clustering association over time
%       * C: 1-nY/1-nX array with the state variables (the position in xz plane) of the centroid
%       * P: number in %1 presenting the cluster probability
%       * candidate: number presenting if the cluster is a candidate
%       * validated: to 1 if the cluster is validated,to 0 otherwise
%       * M: nM-1 cluster M
%       * nM: number informing about the number cluster M
%   - clusterOut3D: K-1 array of structures of info about the 3D output classes TO MODIFY
%   - ks: array of info about the number of classes [kOut,kOutValid,kOut3D,kOut3DValid]
%   - ts: time increment from last execution time. To implement the state vector equation
%   - plotOn: If plotting XPF execution info (pdf,weights,ETC) is wanted to 1... time consuming?
% Outputs :
%   - xParticInit: nN-nX array with the particles value after the re-init step
%   - xParticPred: nN-nX array with the particles after the pred/corr step,before the resampl
%   - wParticPred: nN-1 array with the particles' weigh after the pred/corr step,before resampl
%   - xPartic: nN-nX array with the particles value at the end of the XPF
%   - clusterOut: K-1 array of structures of info about the output classes
%   - clusterOut3D: K-1 array of structures of info about the 3D output classes
%   - ks: array of info about the number of classes [k,kValid,kOut,kValidOut]
% K number of clusters,and nY is the almost-half number of states (2=>X,Z of 5=>X,Z,Y,VX,VZ)
% CAREFUL: WE ARE ALWAYS WORKING ONLY WITH X AND Z. Y IS ONLY USED TO PLOT IN THE IMAGE
% REVISAR AL FINAL,cluster.C 1x5,cluster.M nMx3,+PLOTS,cenAntIn/Out???
% kInx3?? kOutx5?? (no serian necesarios mas q 2 en ambos casos)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Llamadas a variables globales
% ================================================================
global espacio

% ================================================================
% Pintar medidas
figure(1)
plot3(Sistema.y(:,1),Sistema.y(:,2),Sistema.y(:,3),'.b');
view(-37.5+140,40); grid on; axis(espacio);
title('meas antes de ecualizar')
% ================================================================

% ========================================================
% Equalizing meas
% ========================================================
% Existen dos posibles algoritmo para realizar esto

% a) Umbralizando los histogramas de medidas en XZ
y=quitar_redundantes(Sistema.y,AlgParams.Eq,plotOn);

% b) Mediante un k-medias 3D, umbralización de histogramas y discretización
% del espacio.
% y=meas_eq(Sistema.y,Clases.in,AlgParams.Eq,ts,plotOn);
```

```
nMeas=size(y,1);

% ======================================================
% Determine initial uncertainty of every object possition
% ======================================================

if AlgParams.TDPF(1)==0

    % ======================================================
    % CALLING THE CLUSTERING PROCESS
    % ======================================================

    % Generating the first pdf
    if nMeas<=AlgParams.TDPF(2)
        xPartic=y;
        nPartic=nMeas;
    else
        xPartic=y(randsample(nMeas,AlgParams.TDPF(2)),:);
        nPartic=AlgParams.TDPF(2);
    end
    xPartic(:,[4:5])=zeros(nPartic,2);

    % To avoid errors in the returned variables
    xParticPred=[];
    wParticPred=[];
    clusterOut3D=[];

% ======================================
% PF LOOP:
% ======================================

else

    % ======================================
    % REINITIALIZATION STEP:
    % ======================================

    % Recuperando info
    nPartic=size(xPartic,1);

    % Distributing the measurements at t-1 within the nM particles
    nParticAsig=fix(AlgParams.TDPF(2)*AlgParams.TDPF(3)/100);
    if nMeas>nParticAsig
        xPartic(nPartic+1:nPartic+nParticAsig,1:3)=y(randsample(nMeas,nParticAsig),:);
        xPartic(nPartic+1:nPartic+nParticAsig,4:5)=zeros(nParticAsig,2);
    else
        medidas=y;
        while size(medidas,1)<nPartic
            medidas=cat(1,medidas,y);
        end
        xPartic(nPartic+1:nPartic+nParticAsig,1:3)=medidas(randsample(size(medidas,1),nParticAsig),:);
        xPartic(nPartic+1:nPartic+nParticAsig,4:5)=zeros(nParticAsig,2);
    end
    nPartic=nPartic+nParticAsig;
```

```
    % ======================================
    % PREDICTION STEP: We use the transition prior as proposal
    % ======================================

    % HERE important to insert noise in order to increase particles dispersion!!!
    processNoise(:,1:Sistema.Params(2))=Sistema.Params(3).*randn(nPartic,Sistema.Params(2));
    processNoise(:,Sistema.Params(2)+1:Sistema.Params(1))=Sistema.Params(3)/ts.*randn(nPartic,Sistema.Params(1)-Sistema.Params(2));
    xParticPred=MIffun(xPartic,processNoise,ts);

    % ============================================================
    % EVALUATE IMPORTANCE WEIGHTS:
    % ============================================================

    % OJO,FALTA PONDERACION W EN FCN DE cl.p. SE PODRIA INCLUIR PONDERAR SI CLUSTER VALIDO O NO,
    % AUQ YO SOY PARTIDARIA DE Q SI NO ES VALIDO NO SE CUENTE CON EL
    yPred=MIhfun(xParticPred,zeros(nPartic,Sistema.Params(2)));
    wParticPred=calculoPesos(y,yPred,AlgParams.Weigh);

    % ============================================================
    % Pintar las particulas como volumenes
%     figure(40); clf; hold on;
%     for i=1:nPartic
%
%         % Pintarlas como esferas
%         if AlgParams.Weigh(6)==2
%
%             % escalar la esfera en funcion de su peso (o no)
%             [xs,ys,zs]=sphere(10);
%             xPlot=xs.*wParticPred(i); %xPlot=xs.*radio;
%             yPlot=ys.*wParticPred(i); %yPlot=ys.*radio;
%             zPlot=zs.*wParticPred(i); %zPlot=zs.*radio;
%
%             % trasladarla a su posicion
%             xPlot=xPlot+xParticPred(i,1);
%             yPlot=yPlot+xParticPred(i,2);
%             zPlot=zPlot+xParticPred(i,3);
%
%             % dibujarla
%             surf(xPlot,yPlot,zPlot); colormap([1  0  0;1  0  0]);   % en rojo
%
%         % Pintarlas como cubos
%         elseif AlgParams.Weigh(6)==1
%             radio = AlgParams.Weigh(1);
%             voxel([xParticPred(i,1)-radio xParticPred(i,2)-radio xParticPred(i,3)-radio],...
%                 [2*radio 2*radio 2*radio],'r',0.5);
%         end
%     end
%     shading interp
%     light('Position',[1 0 0],'Style','infinite');
%     grid on; axis([espacio]); view(3);
%     title('particulas antes resampling')

%     % Pintar las particulas como ptos
%     figure(41);
```

```
%     plot3(xParticPred(:,1),xParticPred(:,2),xParticPred(:,3),'.r','MarkerSize',20);
%     axis(espacio)
%     title('particulas antes resampling')
%     % ============================================================

    % Normalise the weights
    auxRuido=sum(wParticPred);
    if auxRuido
        wParticPred=wParticPred./auxRuido;
    end

    % ============================================================
    % SELECTION STEP:
    % ============================================================

    nParticAsig=AlgParams.TDPF(2)-fix(AlgParams.TDPF(2)*AlgParams.TDPF(3)/100);
    idxOut=[];
    if auxRuido

        % Residual resampling
        if AlgParams.TDPF(5)==1
            idxOut=MIresidualR(1:nPartic,wParticPred,nParticAsig);

        % Systematic resampling
        elseif AlgParams.TDPF(5)==2
            idxOut=MIsystematicR(1:nPartic,wParticPred,nParticAsig);

        % Multinomial resampling
        elseif AlgParams.TDPF(5)==3
            idxOut=MImultinomialR(1:nPartic,wParticPred,nParticAsig);
        else
            disp('Error,ese tipo de resampling no esta implementado');
        end
    end

    % Keep particles with resampled indices
    xPartic=xParticPred(idxOut,:);
    nPartic=size(xPartic,1);

    % ========================================================
    % Plotting partics exist after resampling
    figure(5)
    plot3(xPartic(:,1),xPartic(:,2),xPartic(:,3),'.r');
    axis(espacio);
    title('Particulas despues resampling'); grid on;
    % ========================================================

    % ============================================================
    % OBTENCIÓN DE SALIDAS
    % ============================================================

    % Params de clasificacion salida
    nAlturas=5;
    alturaMax=1800;
    m=1;i=1;
```

```
if plotOn
    figure(7); clf;
end
while i<nAlturas

    % Definir limites de altura
    alturaInf=(i-1)*(alturaMax/nAlturas);
    if i~=nAlturas
        alturaSup=(i)*(alturaMax/nAlturas);
    else
        alturaSup=alturaMax;
    end

    % Buscar partículas dentro de esos límites
    k=1;
    tempPartic=[];
    for j=1:nPartic
        if (xPartic(j,3)>alturaInf) & (xPartic(j,3)<alturaSup)
            tempPartic(k,1:2)=xPartic(j,1:2);
            tempPartic(k,3)=(alturaSup+alturaInf)/2;
            k=k+1;
        end
    end
    if isempty(tempPartic)==0 & plotOn
        subplot(2,3,i)
        plot(tempPartic(:,1),tempPartic(:,2),'.r');
    end

    % Clustering at each slice
    [clusterOut,foo,kOut,validOut]=clusterKMeans3SinBuff(tempPartic,[],[],Clases.out,ts);

    % Taking centroids info
    for j=1:length(clusterOut)
        centSlices(m,:)=clusterOut(j).C;
        m=m+1;
    end

    % ================================
    % Dibujar particulas por alturas
    if plotOn
        figure(7)
        if i==1
            clf;
        end
        subplot(2,3,i)
        hold on
        if isempty(tempPartic) == 0
            plot(tempPartic(:,1),tempPartic(:,2),'.r');
        end
    end
    % ================================

    i=i+1;
end
    % ======================================================
```

```
    %  Dibujando puntos centroides por alturas
    punto=centSlices(:,1:3);
    if plotOn
        handler=figure(7);
        subplot(231)
        aux = punto(punto(:,3)==1*(alturaMax/(nAlturas))-180,:);
        for k=1:size(aux,1)
            [X Y] = circle(aux(k,1), aux(k,2),sqrt(AlgParams.TDPF(6)),40,'b');
            plot([X X],[Y Y]);
        end
        title('200'); axis(espacio(1:4)); grid on;
        subplot(232)
        aux = punto(punto(:,3)==2*(alturaMax/(nAlturas))-180,:);
        for k=1:size(aux,1)
            [X Y] = circle(aux(k,1), aux(k,2),sqrt(AlgParams.TDPF(6)),40,'b');
            plot([X X],[Y Y]);
        end
        title('600'); axis(espacio(1:4)); grid on;
        subplot(233)
        aux = punto(punto(:,3)==3*(alturaMax/(nAlturas))-180,:);
        for k=1:size(aux,1)
            [X Y] = circle(aux(k,1), aux(k,2),sqrt(AlgParams.TDPF(6)),40,'b');
            plot([X X],[Y Y]);
        end
        title('1000'); axis(espacio(1:4)); grid on;
        subplot(234)
        aux = punto(punto(:,3)==4*(alturaMax/(nAlturas))-180,:);
        for k=1:size(aux,1)
            [X Y] = circle(aux(k,1), aux(k,2),sqrt(AlgParams.TDPF(6)),40,'b');
            plot([X X],[Y Y]);
        end
        title('1400'); axis(espacio(1:4)); grid on;
        subplot(235)
        aux = punto(punto(:,3)==5*(alturaMax/(nAlturas))-180,:);
        for k=1:size(aux,1)
            [X Y] = circle(aux(k,1), aux(k,2),sqrt(AlgParams.TDPF(6)),40,'b');
            plot([X X],[Y Y]);
        end
        title('1800'); axis(espacio(1:4)); grid on;
        img_=sprintf('nivel%.3d.png',AlgParams.TDPF(1));
        saveas(handler,img_,'png');
    end
    % ================================================================

    % ================================================================
    % CLUSTER-OUT. A partir de los clusters por niveles
    % ================================================================

    % Llamando al proceso de conexion de centrodies.
    clusterOut3D=conect3d(punto,Clases.TD);    % número de clases de salida

% FIN DE TDPF... sale de if de reinit pf
end
```

10.2.4. calculoPesos.m

```
 function peso=calculoPesos(y,xPartic,params)
% ========================================================================
% Función calculoPesos
% ------------------------------------------------------------------------
% Parámetros de entrada
%     y: Medidas
%     xPartic: Particulas
%     Weigh: Params of the particle weighting
%       - radio: (200mm) Radio dentro del cual se buscan medidas alrededor
%       de partic
%       - variacionAltura: (40) Tamaño del grid
%       - algoritmo: 1: Cúbico, 2: Esférico
%       - porcentajeSup: (20) Porcentaje de saturación de pesos superior
%       - porcentajeInf: (0) Porcentaje de saturación de pesos inferior
% Parámetros de salida
%     pesos: Array con los pesos calculados
% ========================================================================

% ========================================================================
% Initializing
% ========================================================================

% Recogiendo params
radio=params(1);
variacionAltura=params(2);
algoritmo=params(3);
porcentajeSup=params(4);
porcentajeInf=params(5);
nMeas=size(y,1);

% oTRAS INIT
nPartic=size(xPartic,1);    % Obtener el numero de particulas
peso=[];
if algoritmo==1             % cubico
    volumenVentana=(radio*2)^3;
elseif algoritmo==2         % esférico
    volumenVentana=(4/3)*pi*(radio*2)^3;
end

% ========================================================================
% Saturar.
% ========================================================================
% En este paso se calcula el peso máximo y mínimo asociable a una
% particula. El peso máximo será un porcentaje determinado (dado por
% porcentajeSup) de medidas que caben en la ventana de búsqueda (ésta sera
% una esfera si algoritmo==1 o un cubo si algoritmo==2), y el mínimo
% depende de porcetnajeInf

volumenMeas=variacionAltura^3;
measPorVentana=volumenVentana/volumenMeas;  % Número de meas que caben
limiteSup=measPorVentana*(porcentajeSup/100);
```

```
limiteInf=measPorVentana*(porcentajeInf/100);

% ========================================================================
% Calculo de pesos % Cubico !! OJO A SIGNIFICADO DE RADIO
% ========================================================================

if algoritmo==1
    if isempty(y)==0 && isempty(xPartic)==0
        for i=1:nPartic
            xmax=xPartic(i,1)+radio;
            xmin=xPartic(i,1)-radio;
            zmax=xPartic(i,2)+radio;
            zmin=xPartic(i,2)-radio;
            ymax=xPartic(i,3)+radio;
            ymin=xPartic(i,3)-radio;
            coincidencias=(y(:,1)>xmin&y(:,1)<xmax) &...
                (y(:,2)>zmin&y(:,2)<zmax) & (y(:,3)>ymin&y(:,3)<ymax);
            peso(i)=size(find(coincidencias),1);
            if peso(i)>limiteSup
                peso(i)=limiteSup;
            elseif peso(i)<limiteInf
                peso(i)=limiteInf;
            end
        end
    else
        disp('No hay meas o particulas, no se pueden calcular los pesos')
    end
end
peso=peso';                 % Para respetar el resto del programa
```

10.2.5. conect3D.m

```
 function nClases=conect3d(punto,params)
% ========================================================================
% Función conexion clusters salida. Alvaro Marcos. 10/4/09
%    A partir de los clusters por niveles realizado anteriormente
% obtenemos las clases de salida.
% -----------------------------------
% Parámetros de entrada
%     punto: Array con los puntos a conectar
%     Clases.TD: Estructura de parámetros de clasificación de salida
% Parámetros de salida
%     nClases: Número de clases
% ========================================================================

% Llamada a variables globales
global color_
global color2
global espacio

distM = params(1);
pond_xz = params(2);
pond_y = params(3);
```

```
% Formato: coord x,coord z,coord y,id
n_puntos=size(punto,1);
punto=cat(2,punto,(1:n_puntos)');

% ========================================
% ARRAYS DE CONECTIVIDAD
% ========================================

% coger un punto
for i=1:n_puntos
    conectividad(punto(i,4)).punto=punto(i,1:3);
    k=1;

    % buscar entre todos cual tiene cerca
    for j=1:n_puntos
        if i~=j
            if norm([pond_xz*(punto(i,1)-punto(j,1)) ...
                    pond_xz*(punto(i,2)-punto(j,2)) ...
                    pond_y*(punto(i,3)-punto(j,3))])<distM
                conectividad(punto(i,4)).miembros(k,:)=punto(j,4);
                k=k+1;
            end
        end
    end
end

% ========================================
% CALCULAR LAS CLASES
% ========================================

% nClases(i).miembros -> guarda IDs de elementos pertenecientes a la nClases
id_a_analizar=1;
id_analizado=[];
miembros=[];
nClases=1;

figure(8); clf;

% while id analizado != todos
while length(id_analizado)<n_puntos

    id_analizado_ordenado=limpiar_array(id_analizado);
    for i=1:length(id_analizado_ordenado)
        if id_analizado_ordenado(i) ~= i
            id_a_analizar=i;
            break;
        end
    end

    if isempty(id_a_analizar)==1
        id_a_analizar=length(id_analizado_ordenado)+1;
    end

    miembros=id_a_analizar;
```

```
while isempty(id_a_analizar)==0

            miembros=cat(2,miembros,conectividad(id_a_analizar(1)).miembros');

            % Incluir id en id_analizado
            id_analizado=cat(2,id_analizado,id_a_analizar(1));

            % Establecer los nuevos id a analizar.
            aux=cat(2,id_a_analizar,conectividad(id_a_analizar(1)).miembros');

            % Hacer que no meta un id_a_analizar ya analizado
            for i=1:length(aux)
                ya_estaba=0;
                for j=1:length(id_analizado)
                    if aux(i)==id_analizado(j)
                        ya_estaba=1;
                    end
                end

                if ya_estaba==0
                    id_a_analizar=cat(2,id_a_analizar,aux(i));
                end
            end

            % Borrar id una vez analizado en id_a_analizar
            id_a_analizar(1)=[];
            id_a_analizar=limpiar_array(id_a_analizar);

end
miembros=limpiar_array(miembros);

% ========================================
% PLOTTING conectividad
% ========================================

figure(8); hold on;
for j=1:length(miembros)
    id=miembros(j);
    plot3(conectividad(id).punto(1),conectividad(id).punto(2),...
        conectividad(id).punto(3),color_(nClases*2:nClases*2+1),'MarkerSize',20);
    id_text=sprintf('%d',miembros(j));
    text(conectividad(id).punto(1)+30,conectividad(id).punto(2)+30,...
        conectividad(id).punto(3)+30,id_text);

    % Dibujar las lineas de conectividad
    punto_inicial_x=conectividad(id).punto(1);
    punto_inicial_z=conectividad(id).punto(2);
    punto_inicial_y=conectividad(id).punto(3);
    for k=1:size(conectividad(id).miembros,1)
        punto_final_x=conectividad(conectividad(id).miembros(k)).punto(1);
        punto_final_z=conectividad(conectividad(id).miembros(k)).punto(2);
        punto_final_y=conectividad(conectividad(id).miembros(k)).punto(3);
        plot3([punto_inicial_x punto_final_x],[punto_inicial_z punto_final_z],...
            [punto_inicial_y punto_final_y],color2(nClases*2:nClases*2+1),'LineWidth',2);
    end
```

```
    end
    hold off; axis(espacio); grid on; view(30,60);

    % Del while
    nClases=nClases+1;
end

% Para devolver el número correcto de clases
nClases=nClases - 1;
```

10.2.6. meas_eq1

```
 function y=quitar_redundantes(y,EqParams,plotOn)
% ========================================================================
% Reducir la redundancia de medidas. Alvaro Marcos. 10/4/09
%    Se analiza el histograma de medidas para rechazar lo máximo posible
% aquellas que no aportan información útil
% ------------------------------------
% Parámetros de entrada
%     y: Medidas
%     EqParams: Parámetros de ecualización
%     plotOn: Mostrar resultados activado / desactivado
% Parámetros de salida
%     y: Medidas procesadas
% ========================================================================

% ================================================
% Inicialización de parámetros
% ================================================

regionesHist=EqParams(1);
umbral=EqParams(2);
porcentaje=EqParams(3);

% ================================================
% Obtención de los histogramas
% ================================================

[valor_x,pose_x]=hist(y(:,1),regionesHist);
[valor_z,pose_z]=hist(y(:,2),regionesHist);

% ================================================
% plotting distribucion meas y umbral
if plotOn
    figure(31); clf;
    subplot(211); hold on;
    bar(pose_x,valor_x)
    plot([pose_x(1) pose_x(regionesHist)],[umbral umbral],'r','LineWidth',2)
    subplot(212); hold on;
    bar(pose_z,valor_z)
    plot([pose_z(1) pose_z(regionesHist)],[umbral umbral],'r','LineWidth',2)
end
% ================================================
```

```
% ================================================
% Obtener regiones de redundancia.
% ================================================

paso_x=abs(pose_x(2)-pose_x(1))/2;     % anchura mitad barra x
paso_z=abs(pose_z(2)-pose_z(1))/2;     % anchura mitad barra z
k=1; l=1;
redundanteX(k).x =[];
redundanteZ(k).z =[];
for j=1:regionesHist
    if valor_x(j)>umbral
        redundanteX(k).x=[pose_x(j)-paso_x pose_x(j)+paso_x];
        k=k+1;
    end
    if valor_z(j)>umbral
        redundanteZ(l).z=[pose_z(j)-paso_z pose_z(j)+paso_z];
        l=l+1;
    end
end

% ================================================
if plotOn
    figure(32); clf;
    subplot(121); hold on;
    for j=1:(size([redundanteX.x],2)/2)
        for k=1:(size([redundanteZ.z],2)/2)
            ancho=abs(redundanteX(j).x(2)-redundanteX(j).x(1));
            largo=abs(redundanteZ(k).z(2)-redundanteZ(k).z(1));
            alto=1;
            inicial=[redundanteX(j).x(1) redundanteZ(k).z(1) 0];
            voxel(inicial,[ancho largo alto],'r',0.1);
        end
    end
end
% ================================================

% ================================================
% Quitar las medidas
% ================================================

for j=1:(size([redundanteX.x],2)/2)
    for k=1:(size([redundanteZ.z],2)/2)
        y(find((y(:,1)>redundanteX(j).x(1))&(y(:,1)<redundanteX(j).x(2))&...
            (y(:,2)>redundanteZ(k).z(1))&(y(:,2)<redundanteZ(k).z(2))&...
            rand(size(y,1),1)>porcentaje),:)=[];
    end
end

% ================================================
% Plotting
if plotOn
    figure(32); subplot(121)
    plot(y(:,1),y(:,2),'.g'); grid on;
    subplot(122)
```

```
    plot3(y(:,1),y(:,2),y(:,3),'.g');
    view(3); grid on;
end
% ================================================
```

10.2.7. meas_eq2

```
 function y_eq=meas_eq(y,ClasesParams,EqParams,ts,plotOn)
% ========================================================================
% Función ecualización de medidas. Alvaro Marcos. 10/4/09
% Nuevo algoritmo de ecualización de medidas, en tres pasos.
% 1) Se agrupan las medidas utilizando un k-medias tridimensional. Se
% analiza el número de medidas asociadas a cada grupo. Si superan un
% determinado umbral se va al paso 2)
% 2) Se analiza el histograma del grupo con exceso de medidas. En las zonas
% en las que se supere un determinado umbral se va al paso 3)
% 3) Se discretiza el volumen en celdas, en la zona con exceso de medidas
% determinada mediante 1) y 2). Si en una celda existen medidas, se
% sustituyen todas por una única medida situada en el centro de la celda.
% Así se consigue igualar la densidad de medidas.
% ------------------------------------
% Parámetros de entrada
%     y: Medidas originales
%     ClasesParams: Parámetros para la clasificación de medidas
%     EqParams: Parámetros relativos a la ecualización de medidas
%     ts: Periodo de muestreo
%     plotOn: Activa o desactiva varios plots
% Parámetros de salida
%     y_eq: Medidas ya ecualizadas
% ========================================================================

% Llamada a variables globales
global espacio

tamCelda=100;   % tamaño celda en mm


% Clases.in=[Sistema.Params(2) 100 0 0 0 0 800^2 0 distM2 300];
% - params: Array with clustering parameters (in the order indicated)
%       - nXY: Lenght of the data vector to cluster, can be nX or nY, or a mixeture
%       - iterM: Num max of iterations at the clustering algorithm NOT NEEDED IN SUBTRACT
%       - canM: NUMBER of iter. that cluster is invalid/valid before valida/invalid
%       - hist: histheresys to valid/invalid due to cluster lik | dist to pred centroide
%       - factOlv: forgeting factor in the cluster likelihood
%       - kLikM: normalizing factor to calc cluster lik according to the number clusters
%       - distM: DIFFERENT USES. Fixed here to 8*Sistema.cY; (10*) THINK AND SEE
%       - um: threshold to validate the cluster according to its likelihood
%       - distRM: (subtract) Enable area for cluster intersection. 8*Sistema.cY; (10*)? THINK
%       - umOutliers: (subtract) pdf threshold to create new cluster

% ================================================
% Realizar la clasificacion 3D
% ================================================
```

```
 [clusterIn,foo,k,foo]=clusterKMeans3SinBuff(y(:,1:3),[],[],ClasesParams,ts);

% ================================================
% Procesado de grupos.
% ================================================

y_eq=[];
umbral_hist=EqParams(2);
figure(4);clf;hold on;
for i=1:k

    % ================================================
    % Si las medidas superan el umbral de miembros, ecualizar enrejillando volumen
    % ================================================

    if clusterIn(i).nM>ClasesParams(10)

        % Posibilidad b) max mins de los hist que superan umbral
        [hist_x,val_x]=hist(clusterIn(i).M(:,1));
        [hist_z,val_z]=hist(clusterIn(i).M(:,2));
        [hist_y,val_y]=hist(clusterIn(i).M(:,3));

        esp_xmin=min(val_x(hist_x>umbral_hist));
        esp_xmax=max(val_x(hist_x>umbral_hist));
        esp_zmin=min(val_z(hist_z>umbral_hist));
        esp_zmax=max(val_z(hist_z>umbral_hist));
        esp_ymin=min(val_y(hist_y>umbral_hist));
        esp_ymax=max(val_y(hist_y>umbral_hist));
        espacio_=[esp_xmin esp_xmax esp_zmin esp_zmax esp_ymin esp_ymax];

        coincidencias=~(clusterIn(i).M(:,1)>esp_xmin&clusterIn(i).M(:,1)<esp_xmax)...
            | ~(clusterIn(i).M(:,2)>esp_zmin&clusterIn(i).M(:,2)<esp_zmax)...
            | ~(clusterIn(i).M(:,3)>esp_ymin&clusterIn(i).M(:,3)<esp_ymax);
        y_eq=cat(1,y_eq,clusterIn(i).M(coincidencias,:));

        ymin=espacio_(5);
        ymax=ymin+tamCelda;
        xmin=espacio_(1);
        xmax=xmin+tamCelda;
        zmin=espacio_(3);
        zmax=zmin+tamCelda;

        % ================================================
        % Recorrer las celdas buscando medidas interiores a ellas
        % ================================================

        while xmax<=espacio_(2)
            while zmax<=espacio_(4)
                while ymax<=espacio_(6)

                    % buscar medidas
                    numero=length(find((y(:,1)>xmin&y(:,1)<xmax) &...
                        (y(:,2)>zmin&y(:,2)<zmax)&(y(:,3)>ymin&y(:,3)<ymax)));
                    if numero>0
                        y_eq=cat(1,y_eq,[0.5*(xmax+xmin) 0.5*(zmax+zmin) 0.5*(ymax+ymin)]);
```

```
                end

                % ================================================
                % Plotting input classes
                if plotOn
                    voxel([xmin zmin ymin],[tamCelda tamCelda tamCelda],[numero numero numero numero],0.1)
                end
                % ================================================

                %COMMENT
                ymax=ymax+tamCelda;
                ymin=ymin+tamCelda;
            end
            ymin=espacio_(5);
            ymax=ymin+tamCelda;
            zmin=zmin+tamCelda;
            zmax=zmax+tamCelda;
        end
        xmin=xmin+tamCelda;
        xmax=xmax+tamCelda;
        zmin=espacio_(3);
        zmax=zmin+tamCelda;
    end

    % ================================================
    % Plotting input classes
    if plotOn
        plot3(clusterIn(i).M(:,1),clusterIn(i).M(:,2),clusterIn(i).M(:,3),'.r');
    end
    % ================================================

% ================================================
% Si no se supera min de meas entrada no quitamos meas
% ================================================

else

    % ================================================
    % Plotting input classes
    if plotOn
        plot3(clusterIn(i).M(:,1),clusterIn(i).M(:,2),clusterIn(i).M(:,3),'.b');
    end
    % ================================================

    y_eq=cat(1,y_eq,clusterIn(i).M);
end
end
```

III. BIBLIOGRAFÍA

[Arulampalam02] M.S. Arulampalam, S. Maskell, N. Gordon, T. Clapp. "A tutorial on particle filters for online nonlinear non-gaussian bayesian tracking", IEEE Transactions on Signal Processing, Vol. 50, nº 2, pp. 174-188, February 2002.

[Atyabi06] M. Atyabi, M. S. Kharjeh Hosseini and M. Mokhtari. "The webcam mouse: visual 3D tracking of body features to provide computer access for people with severe disabilities", Islamic Azad University science & research Branch, 2006.

[Douc05] R. Douc, O. Cappé, E. Moulines. "Comparison of resampling schemes for particle filtering", Proceedings of the Fourth International Symposium on Image and and Signal Processing and Analysis (ISPA05), pp. 64-69, Zagreb, September 2005.

[Doucet97] A. Doucet. "Monte-Carlo methods for bayesian estimation of Hidden-Markov models. Application to Radiation Signal", PhD Thesis, University of Paris-Sud, France, 1997.

[Doucet01] A. Doucet, N. de Freitas, N. Gordon. "Sequential Monte-Carlo methods in practice", Springer-Verlag, ISBN: 0-387-95146-6, New York, 2001.

[Espejo08] V. Espejo, "Sistema de detección de obstáculos y robots mediante múltiples cámaras en espacios inteligentes", Grade Final Proyect, University of Alcalá, Spain, 2008.

[Finlayson02] G.D. Finlayson, S. D. Hordley, M.S. Drewe. "Removing Shadows from Images", European Conference on Computer Vision (ECCV), 2002.

[Focken03] D. Focken and R. Stiefelhagen, “Towards visión-based 3-D people tracking in a smart room”, Interactive System Laboratories, Universität Karlsruhe (TH), Germany, 2003.

[Gordon93] N.J. Gordon, D.J. Salmond, A.F.M. Smith. “Novel approach to nonlinear/non-gaussian bayesian state estimation", IEE Proceedings-F in Radar and Signal Processing, Vol. 140, nº 2, pp. 107-113, April 1993.

[Hol06] J.D. Hol, T.B. Schön, F. Gustafsson. “On resampling algorithms for particle filters”, Proceedings of the Nonlinear Statistical Signal Processing Workshop (NSSPW06), Cambridge, September 2006.

[Jalvo09] R. Jalvo, “Técnicas de reconstrucción volumétrica en tiempo real a partir de múltiples cámaras y su aplicación a la detección de personas”, TFC pendiente de publicación, Universidad de Alcalá, 2009.

[Jeni06] L. Jeni and Z. Istenes, “Mobile agent control in intelligent space using reinforcement learning”, 7th International Symposium of Hungarian Researchers on Computational Intelligence, 2006.

[Kang03] D. B. Yang, H. H. González-Baños and L.J. Guibas, “Counting people in crowds with a real-time network of simple image sensors”, Proceedings of the Ninth IEEE International Conference on Computer Vision, 2003.

[Khan05] Z. Khan, T. Balch, F. Dellaert, “MCMC-Based Particle Filter for Tracking a Variable Number of Interacting Targets” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2005.

[Khan06] Z. Khan, T. Balch, M. Shah, “MCMC-Based Particle Filter for Tracking a Variable Number of Interacting Targets” European Conference on Computer Vision (ECCV), 2006.

[Koller-Meier01] E.B. Koller-Meier, F. Ade. “A Multiview Approach to Tracking People in Crowded Scenes using a Planar Homography Constraint”, Journal of Robotics and Autonomous Systems, Vol. 34, pp. 93-105, February 2001.

[Krumm01] J. Krumm, S. Harris, B. Meyers, B. Brumitt, M. Hale and Steve Shafer, “Multi-camera multi-person tracking for EasyLiving”, Third IEEE International Workshop on Visual Surveillance, 2001.

[Lee99] J.H. Lee, N. Ando, H. Hashimoto. “Intelligent Space for Mobile Robots”, Proceedings of the 1999 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM99), ISBN: 0-7803-5040-5, pp.85-90, Atlanta, September 1999.

[Liu98] J.S. Liu, R. Chen. “Sequential Monte-Carlo methods for dynamic systems”, Journal of the American Statistical Association, Vol. 93, pp. 1032-1044, 1998.

[López07] A. López, C. Canton-Ferrer and J. R. Casas, “Multi-person 3D tracking with particle filters on voxels”, Image Processing Group, Technical University of Catalonia, 2007.

[Marrón05] M. Marrón, J.C. García, M.A. Sotelo, D. Fernández, D. Pizarro. "XPFCP: An extended particle filter for tracking multiple and dynamic objects in complex environments", Proceedings of the IEEE International Symposium on Industrial Electronics 2005 (ISIE05), ISBN: 0-7803-8738-4, Vol. I-IV, pp. 1587-1593, Dubrovnik, June 2005.

[Marrón08] M. Marrón. "Seguimiento de múltiples objetos en entornos interiores muy poblados basado en la combinación de métodos probabilísticos y determinísticos", PhD Thesis, University of Alcala, Spain, 2008.

[Marrón09] M. Marrón, D. Pizarro, A. Marcos, R. Jalvo, J. C. García, M. Mazo. "Multi-agent 3D Tracking in Intelligent Spaces with a Single Extended Particle Filter", Vision and Audio Sensors in Intelligent Spaces, Special Session, WISP. 2009.

[Merwe01] R. Van der Merwe, N. de Freitas, A. Doucet, E. Wan. "The unscented particle filter", Advances in Neural Information Processing Systems, Vol. 13, pp. 584-590, November 2001.

[Pizarro08] D. Pizarro, M. Marrón, D. Peón, M. Mazo, J.C. García, M.A. Sotelo, E. Santiso. "Robot and obstacles localization and tracking with an external camera ring". Proceedings of the 2008 IEEE International Conference on Robotics and Automation (ICRA08), ISBN: 978-1-4244-1647-9, pp. 516-521, Pasadena, May 2008.

[Rodríguez08] J. Rodríguez, "Técnicas de reconstrucción volumétrica a partir de multiples cámaras", TFC, Universidad de Alcalá, 2008.

[Ristic04] B. Ristic, S. Arulampalam, N. Gordon. "Beyond the Kalman filter: Particle filters for tracking applications", Artech House, ISBN: 978-1-58053-631-8, Boston, 2004.

[Shafer98] S. Shafer, J. Krumm, B. Brumitt, B. Meyers, M. Czerwinski, D. Robbins. "The New EasyLiving Proyect at Microsoft Research", DARPA/NIST Smart Spaces Workshop, 1998.

[Viola-Jones02] P. Viola, M. Jones. "Robust Real-Time Object Detection", International Journal on Computer Vision, 2002.

[Yang03] D.B. Yang, H.H. González-Baños, L.J. Guibas. "Counting People in Crowds with a Real-Time Network of Simple Image Sensors", Proceedings of the Ninth IEEE International Conference on Computer Vision (ICCV03), ISBN:0-7695-1950-4, Vol. 1, pp. 122-129, Nice, October 2003.

[Zhou93] B. Zhou, N. K. Bose. "Multitarget Tracking in Clutter: Fast Algorithms for Data Association", IEEE Transactions on Aerospace and Electronic Systems, 1993.