Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Guiado de robots móviles mediante sistema de localización multicámara

Autor: Miguel Granero Ramos

Tutor: Carlos Vivas Venegas

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Guiado de robots móviles mediante sistema de localización multicámara

Autor:
Miguel Granero Ramos

Tutor:
Carlos Vivas Venegas
Profesor titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Grado: Guiado de robots móviles mediante sistema de localización multicámara

Autor: Miguel Granero Ramos

Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2021

Agradecimientos

Gracias a mis padres y a mi hermana por aguantarme en las épocas de estrés.

Miguel Granero Ramos

Sevilla, 2021

Resumen

Este proyecto trata el guiado de un robot móvil teledirigido cuya posición y orientación es estimada mediante un sistema multicámara externo a este.

La plataforma de cámaras está disponible en los laboratorios del Departamento de Ingeniería de Sistemas y Automática de la Universidad de Sevilla. Se trata de un cubo con volumen de aproximadamente 3x3x3 metros y cuatro cámaras infrarrojas dispuestas en las esquinas superiores en configuración multiestéreo.

Para la comunicación y control de las cámaras se emplea el SDK de programación desarrollado por el propio fabricante, Sapera, así como OpenCV para el posterior procesado de las imágenes tomadas. Este post-procesado se realiza en un equipo externo y se envía vía Bluetooth al robot móvil.

La localización del robot móvil se obtiene a partir de la proyección 2D del robot capturada en cada una de las imágenes. Para ello, se hace uso de unos focos infrarrojos cercanos a las cámaras y de unos marcadores reflectantes dispuestos encima del robot móvil. Estos deben ser aislados e identificados en cada una de las cámaras. Es decir, cada marcador tiene asignado un identificador, y debemos ser capaces de reconocer cada marcador en cada una de las imágenes. Para esto es imprescindible emplear una geometría de marcadores suficientemente distintiva, como para que estos sean inequívocamente identificados en cada cámara. Adicionalmente se realizan una serie de filtrados de los datos de marcadores obtenidos con el fin de estabilizar las medidas y obtener un comportamiento lo menos sensible a perturbaciones posible.

Posteriormente se mezcla la información obtenida de cada una de las cámaras para realizar una estimación de la pose del robot móvil en 3D. Se muestran diferentes resultados con diferentes técnicas de obtención de la pose del robot, el *Perspective-n-Points (PnP)* y la *Triangulación*. En el caso del PnP se hace uso del promediado de cuaternios para obtener la orientación media estimada entre las cámaras, ya que cada una realiza una estimación de manera aislada (monocular).

La estimación media final calculada mediante el PnP o Triangulación es enviada al robot móvil y posteriormente filtrada allí con un Filtro de Kalman. Esto nos permite obtener unas estimaciones de posición mejores y con un comportamiento menos impreciso y errático. El robot móvil emplea estos datos filtrados para cerrar el bucle de control de posición y alcanzar el punto objetivo designado por el usuario.

Finalmente, algunos datos recopilados en el robot son enviados de vuelta a la estación de tierra externa vía Bluetooth para ser guardados y posteriormente analizados con Matlab.

Abstract

This project handles the guidance of a teleoperated robot, whose position and orientation are estimated through an external multicamera system.

The camera platform is available at the laboratory of the Systems and Automation Engineering Department from the University of Seville. It is a 3x3x3 meters cube in which the cameras are disposed at the top corners with a multistereo configuration.

For the communication and control of the cameras is used the programming SDK of the own manufacturer, Sapera, in addition to OpenCV for the post-processing. This is done in external equipment, and it is sent to the mobile robot via Bluetooth.

The localization of the robot is obtained from the 2D projection of the robot captured by each camera. For that purpose, several infrared spotlights and reflective markers disposed on the robot are used. These are to be identified in each camera. That means that every marker has an identifier (id) and we should be able to assign these ids to the corresponding marker in every image. To do this, it is essential to use a distinctive geometry for the markers, so they are unequivocally identified in each camera. Additionally, some filters to the makers' data are included, to stabilize the measurements and obtain a less noise-sensitive behavior.

Later the information provided by each camera and it processing is mixed to estimate the 3D robot's pose. Different results are shown using different estimation techniques, such as Perspective-n-Points (PnP) and Triangulation. In the case of PnP, the mathematical tool "quaternion meaning" is used to obtain the mean orientation given by the cameras, since each one of them makes a monocular estimation.

The final mean estimation calculated by the PnP or Triangulation is transmitted to the robot via Bluetooth and later filtered there with a Kalman Filter. This allows the robot to get better pose estimations with a more accurate and less erratic behavior. The robot uses this data to close the position control loop and reach the objective point given by the user.

Finally, some data is gathered in the robot and sent back to the central PC to be stored and later analyzed with the aid of Matlab.

Índice

1 INTRODUCCIÓN

La estimación de pose es un eslabón necesario para la creación de todo tipo de robots autónomos. El conocimiento de su posición y orientación en el espacio es precisamente lo que les permite tomar decisiones de cómo interactuar con el entorno para alcanzar los objetivos propuestos.

En exteriores se suele hacer uso del un sistema muy familiar para todos hoy en día, el GPS[1]. Este nos permite posicionar cualquier objeto sobre la tierra con una precisión de unos pocos metros, lo cual es más que suficiente para poder localizar nuestro coche. Aunque esta medida puede llegar a aumentarse bastante con la ayuda de mediciones y técnicas adicionales.

En interiores, sin embargo, el sistema GPS brilla por su ausencia. Los edificios dificultan el uso de este y por ello es necesario pensar en nuevas soluciones. Este problema no es nuevo, pero aún sigue bajo estudio para lograr un sistema de localización eficaz. La mayoría de ellas están basadas en sistemas IPS[2] o visión artificial. En este TFG se abordará una solución con técnicas de este último grupo.

Las cámaras tienen la gran ventaja de que podemos extraer mucha información del entorno. Siendo esto también una de sus principales desventajas, ya que, dependiendo de la característica a medir, puede resultar más o menos difícil aislarla del resto de información recopilada por el sensor. Pero por la naturaleza del mismo, que realiza una proyección en 2D del espacio, perdemos información sobre la profundidad. He aquí la necesidad de implementar un sistema multicámara.

Con un sistema multicámara podemos obtener varias vistas diferentes de una misma escena y con ellas recomponer un escenario 3D, localizando así a nuestro objetivo de una forma global. Es más, si somos capaces de diferenciar diferentes objetos, también seremos capaces de localizarlos simultáneamente de una forma relativamente sencilla.

En nuestro caso en particular, haremos uso de cámaras del espectro del infrarrojo y la ayuda de unos focos emisores de esta radiación y unos marcadores esféricos recubiertos de un material reflectante. Con ayuda de técnicas de visión por computador elaboraremos una estimación de la pose (posición y orientación) de un robot móvil tipo diferencial, y realizaremos experimentos para ver la viabilidad de uso de esta plataforma para el control de este robot.

1.1 Motivación

Se trata de una primera implementación práctica de un proyecto de localización multicámara basado en una plataforma de visión multiestéreo instalada en los laboratorios de la Escuela Superior de Ingenieros de Sevilla. Esta consta de tres cámaras infrarrojas dispuestas en las esquinas superiores de un cubo de 3 metros de lado.

El trabajo constituye la continuación natural de algunos trabajos previos realizados mediante simulación en [6][17], donde se mostró el potencial de una plataforma de este tipo. La idea es implementar experimentalmente lo realizado en estos trabajos anteriores, comprobar su utilidad y validez práctica y aplicar estos métodos como base para un control por posición de un robot móvil diferencial. Por tanto, espero que sirva como fundamento para futuros proyectos y ampliaciones sobra la misma plataforma, pudiendo conocer así desde un inicio sus posibles capacidades y limitaciones.

En lo personal, he decidido realizar este TFG y abordar este problema por su alto componente práctico y su contenido relacionado con la visión por computador. Siendo este último uno de los campos más interesantes y con más avances y aplicaciones en los últimos tiempos. El componente práctico también es bastante importante a mi parecer, porque nos enseña a lidiar con los problemas físicos que se pueden presentar y a tratar con hardware real, lo cual es adicionalmente un reto en sí.

[1] Global Positioning System
[2] Indoor Positioning System

1.2 Estado del Arte

Los sistemas multicámara tampoco son nada nuevo, de hecho, es bastante común encontrarnos con sistemas de cámaras estéreo (dos cámaras) incluso en nuestras propias casas. Hace algo más de una década Microsoft popularizó un sistema estéreo para uso recreativo con sus consolas XBOX, las Kinect [1].

Figura 1.1 *Kinect [1]*

Pero para estimaciones y problemas más complejos, este sistema bicámara puede quedarse corto. En la industria del cine y animación, por ejemplo, esta tecnología es ampliamente utilizada para registrar los movimientos corporales y poder realizar una reconstrucción fiel a posteriori. Esta técnica se suele denominar por "Motion Capture".

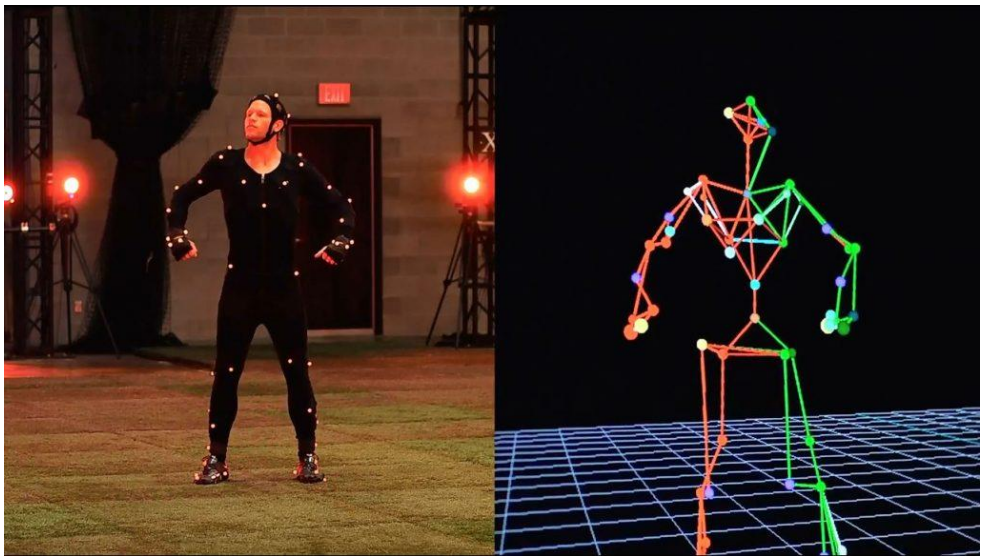

Figura 1.2 *Motion Capture. [proyectoidis.org]*

Aunque para nuestro caso la aplicación sea distinta, la tecnología aplicada detrás es parecida. La identificación y localización de marcadores con sistemas multicámara es un problema sobre el que también podemos encontrar bastantes estudios. Podemos encontrar un problema similar en [3] ("Marker Localization with a Multi-Camera System") en el que realizaban un post-procesado de un vídeo grabado a color para localizar una serie de marcadores.

También existen múltiples artículos y productos desarrollados para la localización de viandantes y vehículos a través de los diferentes planos tomados por cámaras de vigilancia. Un ejemplo de esto lo podemos ver en [4].

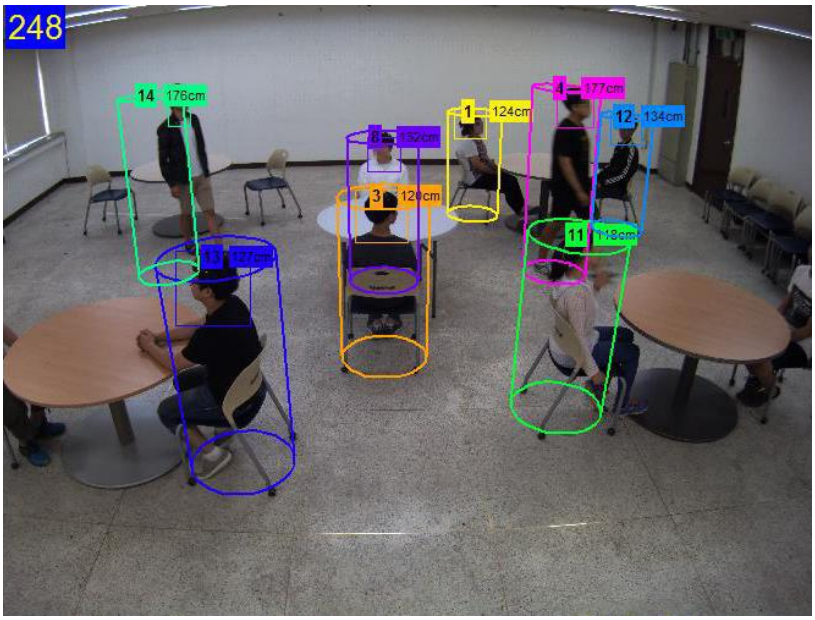

Figura 1.3 *Resultado de [4] para el tracking de personas en interiores.*

El problema que se plantea abordar en este proyecto ya ha sido atacado por otras empresas de la industria. Una de las más conocidas en aplicaciones para robótica es VICON [2]. Estos disponen de una gran gama de cámaras y soluciones para afrontar este tipo de problemas. Cuentan incluso con su propio software para garantizar una mejor precisión, pudiendo hacer uso este desde una interfaz gráfica y haciendo uso de una API[3] para poder crear aplicaciones más personalizadas.

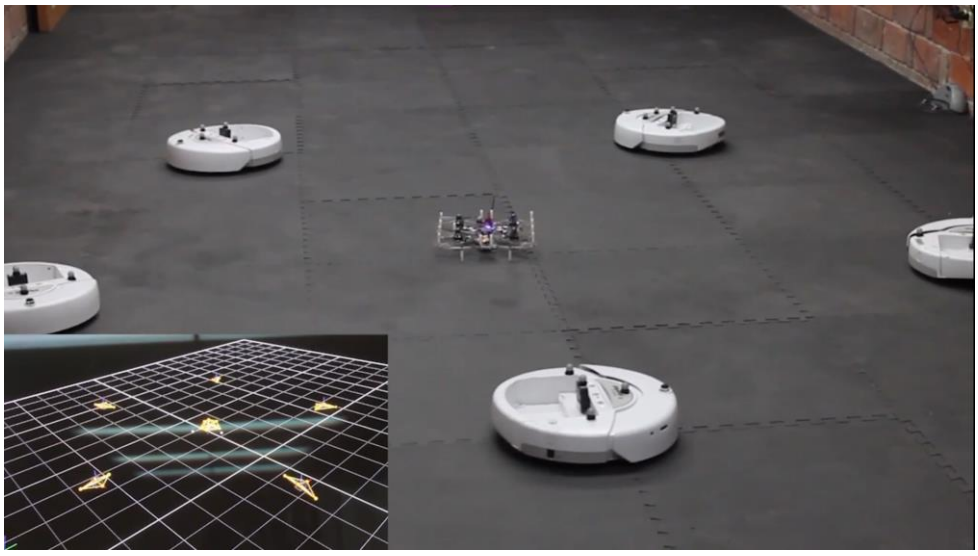

Figura 1.4 *Aplicación VICON para posicionamiento multi-robot*

En el caso de la figura 1.4 podemos ver cómo pueden localizar tridimensionalmente varios robots, incluso de diferente tipo, gracias a la ayuda de su plataforma. Esto les permite cerrar el bucle de control y realizar un control en posición de cada uno de ellos, ya que se encuentran perfectamente identificados.

[3] Application Programming Interface [5]

Si buscamos algo más de información en cuanto a los algoritmos y métodos empleados en este tipo de sistemas podemos ver que suelen repetirse dos métodos:

- Triangulación
- Perspective-N-Point (PNP)

El primero se suele realizar con puntos individuales y nos da una solución que minimiza el error para las perspectivas dadas. El PNP por su parte, intenta "cuadrar" en el espacio una geometría predefinida. Veremos más en detalle estos métodos en sus respectivos subapartados.

Después de esta breve descripción del estado del arte, cabe destacar que en el caso particular de este TFG, parte del trabajo ha tomado como base lo realizado en el TFG de Elisa Hidalgo Moreda [6] ,aunque se han tenido que realizar muchos ajustes para adaptarlo al laboratorio.

1.3 Estructura del documento

Este documento tiene una estructura similar a la que ha llevado el desarrollo del proyecto.

- Software utilizado
- Hardware empleado. Plataforma del laboratorio, cámaras, robot móvil, etc.
- Toma de imágenes y procesado (Percepción)
- Robot Móvil
- Experimentos

La idea de esto es ver como el desarrollo ha ido evolucionando y finalmente realizar una serie de experimentos para comprar los diferentes casos de uso, métodos empleados y resultados finales.

2 SOFTWARE Y ARQUITECTURA

En este apartado veremos todas las herramientas software utilizadas en el proyecto y se explicará brevemente su papel en el mismo. Asimismo, se explicará la arquitectura empleada que determina el flujo de datos a través de los diferentes elementos del sistema.

2.1 Software Empleado

2.1.1 Visual Studio Community 2019

Este entorno de programación por Microsoft [9] es bastante conocido y empleado de forma profesional. Nos permite crear soluciones añadiendo dependencias y compilarlas de forma relativamente sencilla y avanzada. En nuestro caso, dado que la programación debía ser en C++ para obtener una mayor velocidad de ejecución y para poder hacer uso de referencias anteriores, este era el entorno ideal para realizarla. Además, su versión Community es gratuita y ya venía instalada en el PC del laboratorio.

Cabe destacar el amplio uso de la librería *Eigen* [10] para cálculos con matrices, que ha facilitado bastante el desarrollo del proyecto.

2.1.2 OpenCV

OpenCV [11] es la mayor librería para visión por computador del mundo. Tiene versiones en varios idiomas, como Python o C++, y es la que empleamos en este proyecto para realizar de una forma mas sencilla todo el procesamiento de imágenes y algoritmos de alto nivel, ya que la mayoría de ellos ya tienen función propia en esta librería.

Cabe destacar su naturaleza *open-source* que facilita a la comunidad crear y modificar funcionalidades y aumentar la cantidad de información y usuarios a su alrededor.

2.1.3 Sapera SDK

Para el control de las cámaras y la toma de imágenes es necesario usar la propia API[12] proporcionada por Sapera, el fabricante de las cámaras. Aunque su manual cuenta con más de 300 páginas, hemos podido encontrar algunos ejemplos sencillos para su manejo.

Como de aquí en adelante esta parte del procesado se va a obviar, procedo a explicar brevemente su funcionamiento, ya que también ha requerido cierto tiempo de investigación y aprendizaje y espero pueda servir a futuros usuarios de la plataforma. El siguiente código es uno de los ejemplos del SDK (*GrabConsoleCPP*).

Para la obtención de imágenes es necesario crear una serie de objetos y buffers requeridos por la propia API.

Código 1. Creación objetos obtención de imágenes (GrapCPP.cpp)

```
  int main(int argc, char* argv[])
  {
     UINT32   acqDeviceNumber;
     char*    acqServerName = new char[CORSERVER_MAX_STRLEN];
     char*    configFilename = new char[MAX_PATH];

     printf("Sapera Console Grab Example (C++ version)\n");

     if (!GetOptions(argc, argv, acqServerName, &acqDeviceNumber, configFilename))
     {
        printf("\nPress any key to terminate\n");
        CorGetch();
        return 0;
     }
  SapAcquisition Acq;
     SapAcqDevice AcqDevice;
     SapBufferWithTrash Buffers;
     SapTransfer AcqToBuf = SapAcqToBuf(&Acq, &Buffers);
     SapTransfer AcqDeviceToBuf = SapAcqDeviceToBuf(&AcqDevice, &Buffers);
     SapTransfer* Xfer = NULL;
     SapView View;

     SapLocation loc(acqServerName, acqDeviceNumber);

     if (SapManager::GetResourceCount(acqServerName, SapManager::ResourceAcq) > 0)
     {
        Acq = SapAcquisition(loc, configFilename);
        Buffers = SapBufferWithTrash(2, &Acq);
        View = SapView(&Buffers, SapHwndAutomatic);
        AcqToBuf = SapAcqToBuf(&Acq, &Buffers, XferCallback, &View);
        Xfer = &AcqToBuf;

        // Create acquisition object
        if (!Acq.Create())
           goto FreeHandles;

     }

     else if(SapManager::GetResourceCount(acqServerName,SapManager::ResourceAcqDevice)
> 0)
     {
        if (strcmp(configFilename, "NoFile") == 0)
           AcqDevice = SapAcqDevice(loc, FALSE);
        else
           AcqDevice = SapAcqDevice(loc, configFilename);

        Buffers = SapBufferWithTrash(2, &AcqDevice);
        View = SapView(&Buffers, SapHwndAutomatic);
        AcqDeviceToBuf = SapAcqDeviceToBuf(&AcqDevice, &Buffers, XferCallback, &View);
        Xfer = &AcqDeviceToBuf;

        // Create acquisition object
        if (!AcqDevice.Create())
           goto FreeHandles;

     }

     // Create buffer object
     if (!Buffers.Create())
        goto FreeHandles;

     // Create transfer object
     if (Xfer && !Xfer->Create())
        goto FreeHandles;
```

```
   // Create view object
   if (!View.Create())
      goto FreeHandles;
```

En este código se crean los objetos de buffer, ventana de vista y objeto de transferencia para una cámara. Si tuviésemos más de una, como en nuestro caso, debemos realizar esto tantas veces como cámaras tengamos, cada una necesita sus propios recursos.

Una vez hecho esto podemos tomar imágenes desde las cámaras, para realizar esto hay dos formas, una empieza una toma constante de imágenes y otra solo toma tantos fotogramas como queramos:

Código 2. Toma de imágenes

```
Xfer->Grab(); // Para tomar imágenes de forma continuada.
Xfer->Snap(n); //Tomar n fotogramas.

   printf("Press any key to stop grab\n");
   CorGetch();

   // Stop grab
   Xfer->Freeze();
   if (!Xfer->Wait(5000))
      printf("Grab could not stop properly.\n");

FreeHandles:
   printf("Press any key to terminate\n");
   CorGetch();

   //unregister the acquisition callback
   Acq.UnregisterCallback();

   // Destroy view object
   if (!View.Destroy()) return FALSE;

   // Destroy transfer object
   if (Xfer && *Xfer && !Xfer->Destroy()) return FALSE;

   // Destroy buffer object
   if (!Buffers.Destroy()) return FALSE;

   // Destroy acquisition object
   if (!Acq.Destroy()) return FALSE;

   // Destroy acquisition object
   if (!AcqDevice.Destroy()) return FALSE;


   return 0;
}
```

Al principio podemos ver las dos formas principales de tomar imágenes. Cuando acabemos debemos destruir los objetos para librar el espacio en memoria.

Cabe mencionar que este SDK también nos permite transferir las imágenes de unos de los Buffers directamente a un *Mat* de OpenCV.

Código 3. Importar imagen Sapera a OpenCV

```
Mat src1;
src1 = new Mat(1024, 1280, CV_8UC1, Buffers);
```

2.1.4 Arduino IDE

Es la Plataforma de programación de Arduino por defecto, muy extendida y conocida por su simplicidad de uso. Nos permite añadir librerías, compilar y subir los programas a nuestro Arduino o plataformas similares. En este caso lo usaremos para programar el microcontrolador *Baby Orangutan*.

Para ello simplemente debemos descargar la librería correspondiente desde la página web oficial (ver [15]).

2.2 Arquitectura

En el siguiente esquema podemos ver el flujo que los datos siguen a través del sistema.

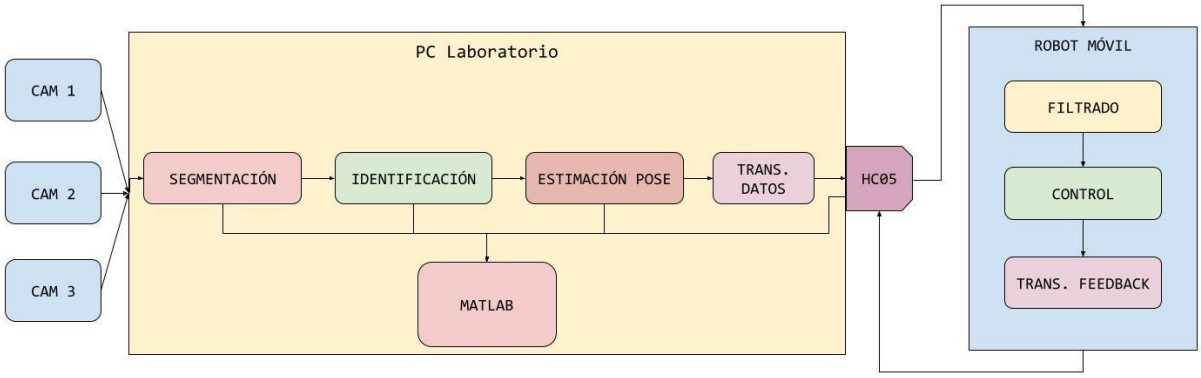

Figura 2.1 Esquema flujo de datos

Las tres cámaras envían sus imágenes al PC, donde son procesadas y segmentadas para localizar los marcadores dentro de ellas. Posteriormente se realiza una identificación de los marcadores, para saber cuál es cual y poder aplicar la estimación de posición correctamente. En este eslabón es donde mezclamos la información de las tres cámaras.

Por último, la información es enviada al robot, donde se filtran y se actúa en consecuencia con estos nuevos datos. Cabe destacar que también se envía cierta información de vuelta al PC para ser registrada y comprobada.

De cada uno de los eslabones sale información hacia Matlab. Si bien esto no se ejecuta a tiempo real y no tiene por qué hacerse en el mismo computador, esto representa que se recogen datos de todos los pasos para poder analizarlos posteriormente con Matlab.

3 HARDWARE

Esta sección tiene como objetivo presentar el hardware empleado para el desarrollo del proyecto con el fin de que el lecto sea capaz de reproducir la plataforma y experimentos si lo quisiera. Se hablará sobre las características de los diferentes componentes, pero no sobre su modo de empleo ni programación si lo requiriesen. Esto se presentará en los apartados respectivos. Espero que sirva como introducción del material que se va a emplear, así como justificación de algunos de los métodos y técnicas empleados.

3.1 Cámaras

Las cámaras empleadas son del modelo *"Genie Nano M1280-NIR"* [7]. Con un sensor infrarrojo y las siguientes características:

- Resolución: 1280x1024 pix
- Distancia Focal: 4.5mm
- Tamaño del sensor: 6.4x4.8mm
- Tasa de muestreo teórica máxima: 52fps

Figure 3.1 *Cámara Genie Nano M1280-NIR*

En total se disponen de 4 cámaras infrarrojas en el laboratorio, pero solo se han podido hacer uso de 3 simultáneamente. No hemos podido averiguar la razón exacta. Pero solo se detectan 3, puede ser porque la placa base solo acepte como máximo 4 conexiones Ethernet y una de ellas está ocupada para tener conexión con Internet.

3.2 Plataforma del laboratorio

La estructura del laboratorio está formada por una serie de vigas y tensores metálicos con el objetivo de hacerla lo más estable posible. Forman un cubo con 2.8 metros de lado.

En cada una de las 4 esquinas superiores se colocan una cámara de las antes mencionadas junto a un foco de radiación infrarroja.

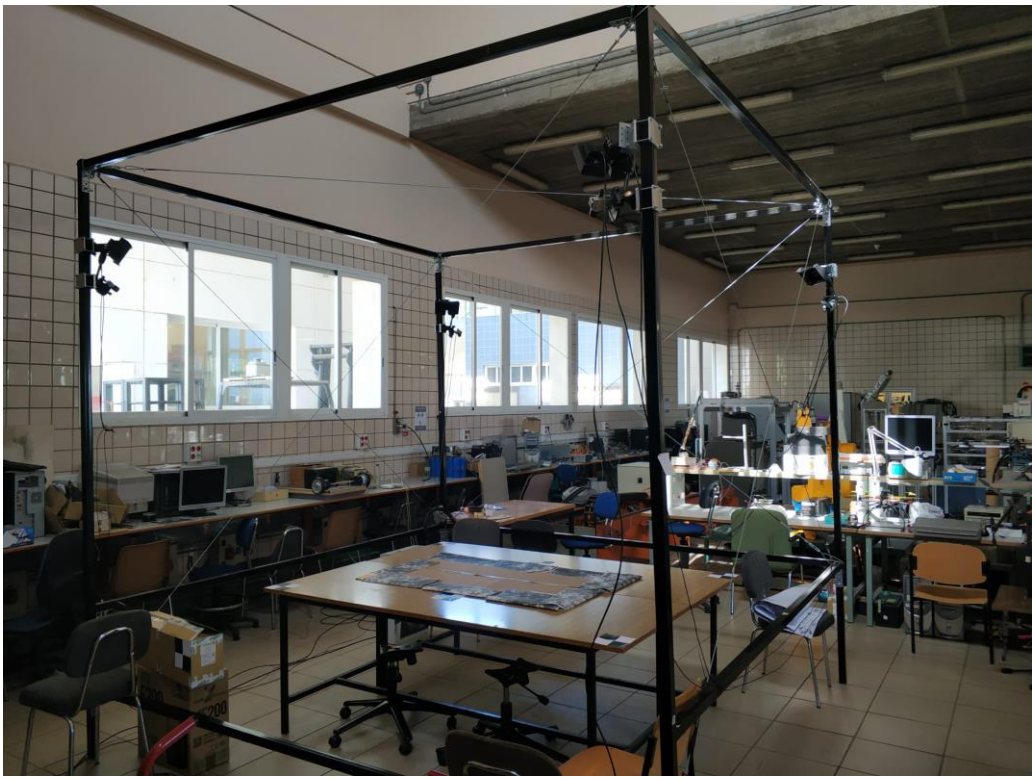

Figura 3.2 *Plataforma del laboratorio*

Figura 3.3 *Focos infrarrojos*

Estos últimos tienen el objetivo de enviar radiación a los marcadores reflectantes, para poder ser reconocidos con mayor facilidad.

3.3 Robot Móvil

Se ha reutilizado uno de los robots de proyectos antiguos del departamento. Este robot es bastante sencillo y pequeño, pero cuenta con el microcontrolador y controlador de motores todo integrado en uno. Se trata de Baby Orangutan de Pololu [8].

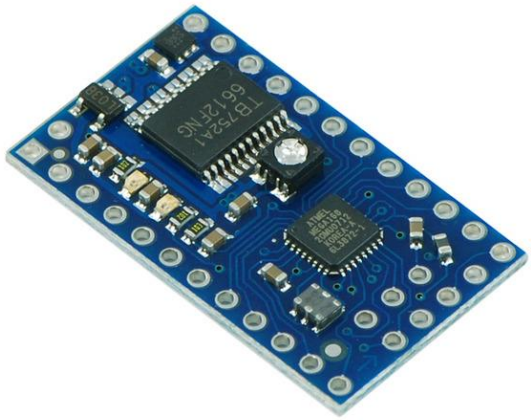

Figura 3.4 *Baby Orangutan*

Soporta una tensión de alimentación de hasta 15V y puede dar potencia hasta a dos motores de corriente continua de 1A de forma continuada. Los motores empleados son motores DC genéricos como los que podemos observar en la siguiente imagen:

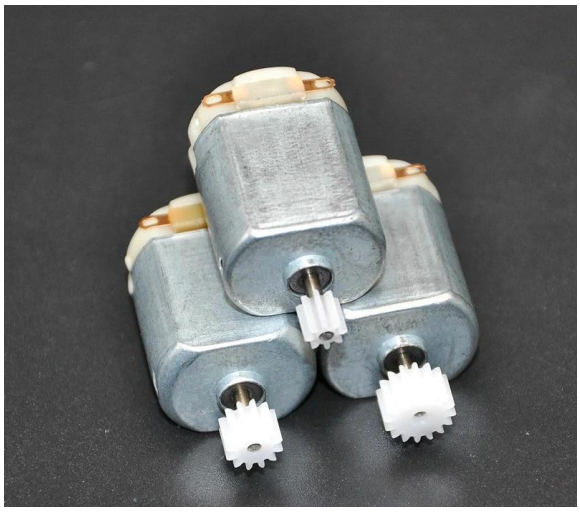

Figura 3.5 *Motores DC*

Adicionalmente se dispone de una placa prediseñada y cedida por el departamento de un antiguo proyecto. No se ha obtenido más información al respeto de ella, pero facilita el conexionado.

Para la alimentación se emplean baterías Li-Po de 7.4V y 800mAh, que otorgan suficiente potencia para hacer funcionar todo.

Figura 3.6 *Baterías*

Para la comunicación, he tenido que añadir un módulo bluetooth HC-06 conectado a los pines de RX y TX del puerto serie de la placa. Estos, desafortunadamente, no estaban accesibles ni marcados a primera vista, por lo que se han tenido que soldar dos pines directamente a su conexión, con el fin de poder seguir utilizando el resto del material sin mayores alteraciones.

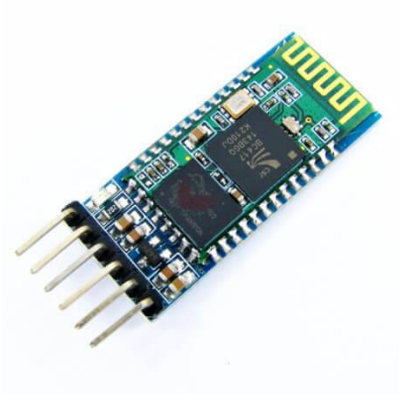

Figura 3.7 *Módulo Bluetooth HC06.*

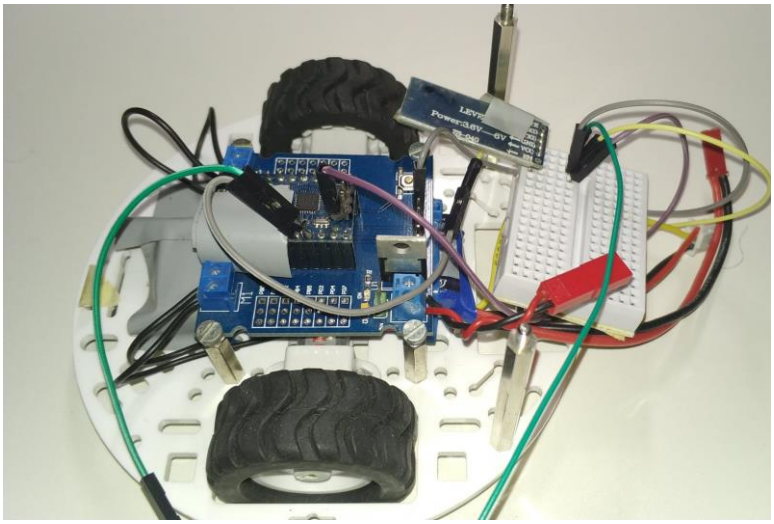

Figura 3.8 *Placa modificada*

El chasis es procedente de un kit de Pololu y también fue cedido por el departamento. Tiene forma circular y un tamaño bastante reducido, perfecto para la aplicación a realizar.

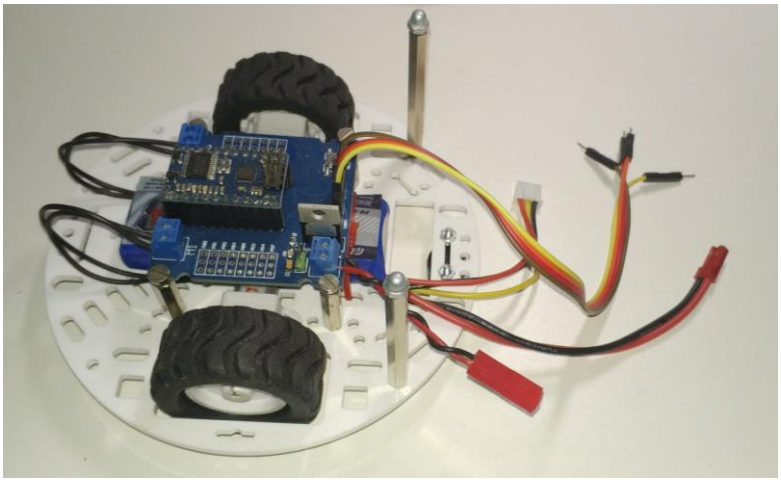

Figura 3.9 *Chásis del robot móvil sin tapa superior.*

Por último, también ha sido necesario realizar un módulo Bluetooth para el PC del laboratorio, ya que este no disponía de este tipo de conexión y tampoco de un adaptador comercial.

Para este módulo he empelado un HC05 con capacidad de conexión como maestro y un Arduino Pro Mini para poder gestionar toda la información. Este simplemente envía por el puerto serie lo que recibe por Bluetooth y viceversa.

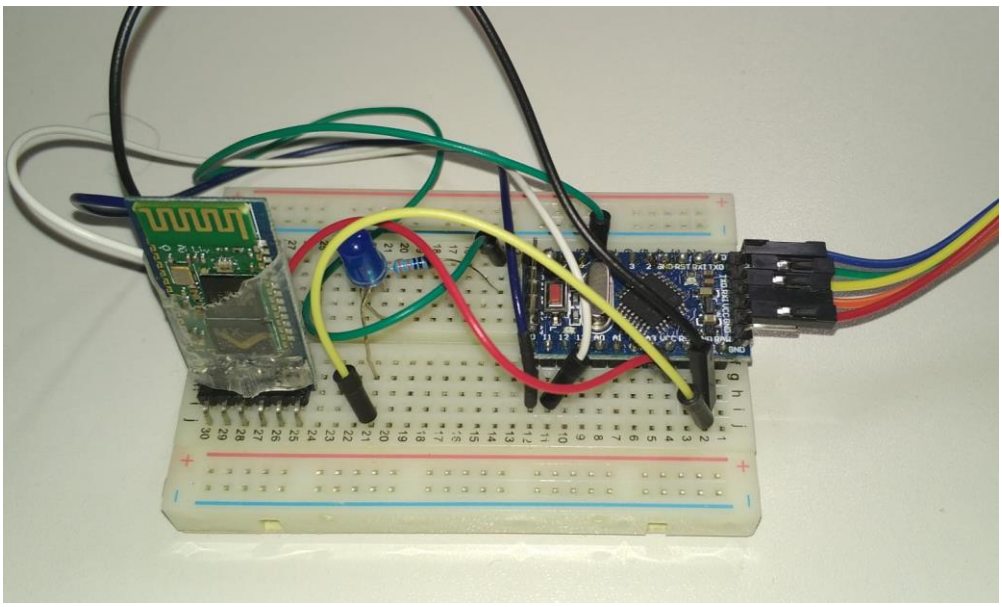

Figura 3.10 *Módulo adaptación Bluetooth para PC del laboratorio.*

4 PASOS PREVIOS Y CALIBRACIÓN

Antes de poder utilizar la plataforma para la localización de objetos, es necesario realizar un par de pasos previos de preparación y asegurarnos un funcionamiento con menos incidentes.

4.1 Adecuación de la Plataforma

Debido a la posición y construcción de la plataforma, debemos tener un par de elementos en cuenta a la hora del desarrollo del proyecto.

La luz del sol y reflejos afectan en gran medida, por ello ha sido necesario cubrir parcialmente la mesa central sobre la que se posaría el robot. Sobre todo, la zona central que es la zona dónde más reflejo encontramos a casusa de los focos y luz solar.

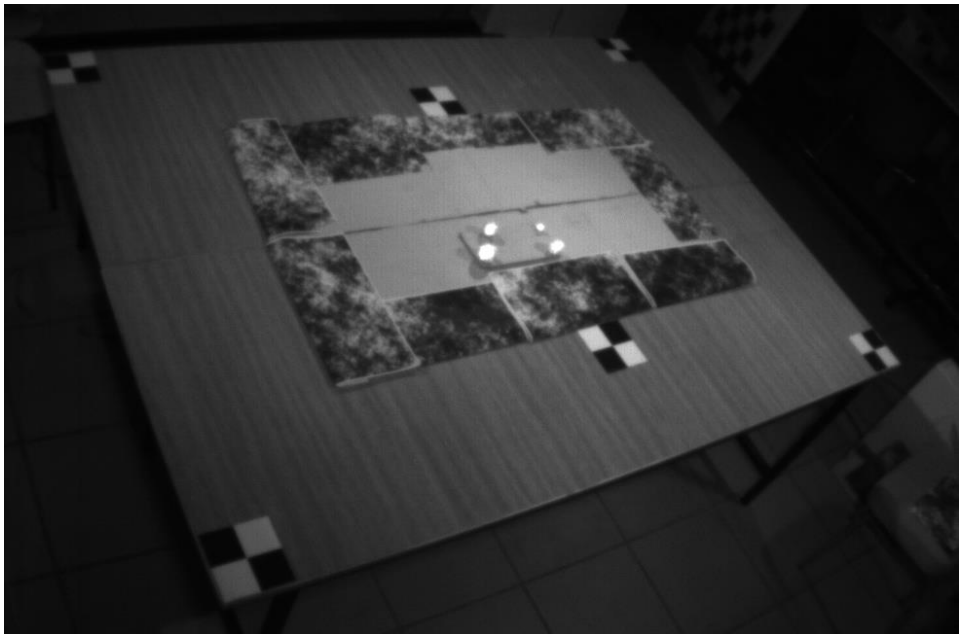

Figura 4.1 *Cartón central de la mesa para tapar reflejos*

Esto podría solucionarse de mejor manera con un mantel o similar, que además también tapase la intersección de las dos mesas para obtener una superficie más lisa. En la imagen también vemos unos marcadores con patrón de ajedrez, que explicaremos en subapartados posteriores.

Además de esto hay que tener en cuenta que hay reflejos que no podremos evitar a ciertas horas del día, ya que una de las ventanas del laboratorio permitía el acceso directo de los rayos del sol sobre la plataforma. Esto empeoraba en gran cantidad todas las medidas y, ya que, no encontramos manera de cerrar o tapar la ventana, hubo que simplemente evitar esos horarios.

Otra adecuación necesaria son los focos infrarrojos. Ya que estos pueden haberse movido por casusa ajenas, es necesario hacerlos apuntar hacia el centro de la mesa, que es donde mejores resultados se obtiene. Esto implica que la zona donde podemos obtener medidas válidas, ya que es donde los marcadores son capaces de reflejar correctamente la luz, es bastante reducida en comparación con el tamaño de la plataforma (aproximadamente el cartón central).

Al igual que los focos, la primera vez de uso de la plataforma conviene adecuar las cámaras para que tengan una buena visión de toda la zona de trabajo. Esto no suele cambiar y ya que posteriormente debemos calibrar las cámaras extrínsecamente, es conveniente realizarlo una sola vez al inicio del proyecto.

4.2 Calibración Intrínseca

4.2.1 Introducción

La calibración intrínseca nos sirve para obtener la matriz de parámetros intrínsecoss (como su propio nombre indica) y coeficientes de distorsión de cada una de las cámaras. Teóricamente disponemos de estos valores, pero no se tienen que corresponder con los valore reales, ni tampoco tienen que ser los mismos entre las cámaras, aunque si similares. Esto nos va a permitir obtener mayor precisión en el resto de procesos del proyecto.

La matriz de parámetros intrínsecos puede definirse, según las referencias de Matlab [13], como:

$$\begin{pmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{pmatrix}$$

Tabla 4.1 Parámetros Intrínsecos [13]

Parámetro	**Significado**
f_x , f_y	Distancias focales
c_x , c_y	Centro Óptico
s	Skew

El parámetro Skew caracteriza el ángulo entre los ejes de la imagen y se define como:

$$s = f_x \tan(\alpha)$$

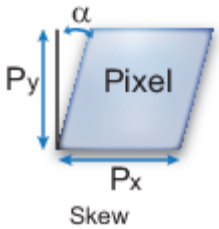

***Figura 4.2** Skew [13]*

La distorsión se caracteriza por unos coeficientes, que definen la forma y grado de esta. Existen dos tipo: radial y tangencial. Aquí solo se contemplará la radial ya que es la que más nos afecta:

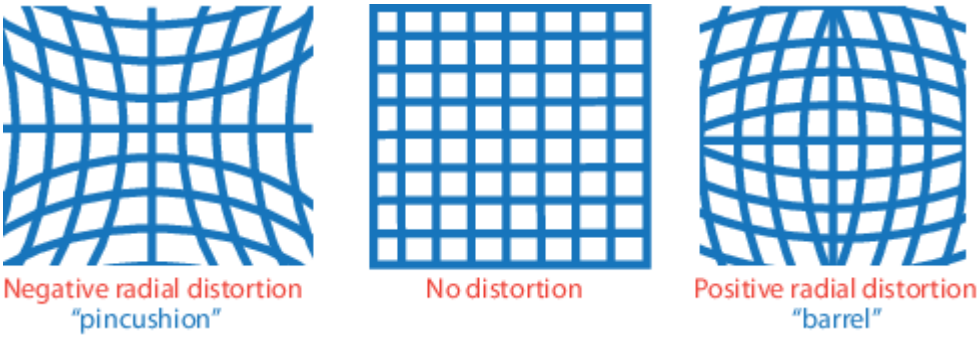

Figura 4.20 *Distorsión Radial [13]*

Esto hace que las imágenes aparezcan deformadas y por tanto haya que tenerlo en cuenta a la hora de realizar cálculos y operaciones sobre las mismas.

4.2.2 Calibración

Existen cantidad de formas de calibrar las cámaras intrínsecamente, en esta memoria solo voy a cubrir el toolbox de Matlab, ya que es la manera más sencilla y la que mejores resultados ha dado.

El toolbox *Camera Calibrator* [14] se basa en el análisis de imágenes tomadas frente a un patrón de ajedrez. Para ello fue necesario imprimir un patrón con el tamaño de cuadrados correcto dado el tamaño de la plataforma. Tras varias pruebas se llegó a la conclusión de que un tamaño de marcador de 10cm x 10cm era apropiado y arrojaba buenos resultados. El flujo de trabajo es el siguiente:

- Tomar de 10 a 20 imágenes con el tablero en diferentes posiciones y orientaciones

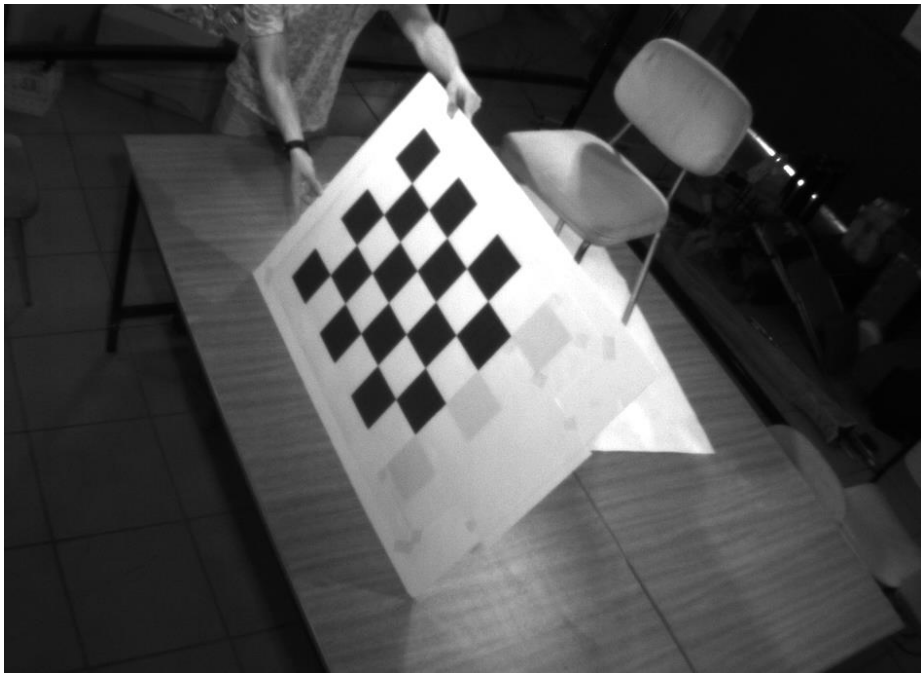

Figura 4.3 *Ejemplo imagen tomada para calibración*

- Abrir el toolBox *Camera Calibrator* e importar las imágenes haciendo clic sobre *Add Images.*
- Introducir el tamaño de los marcadores y esperar la importación

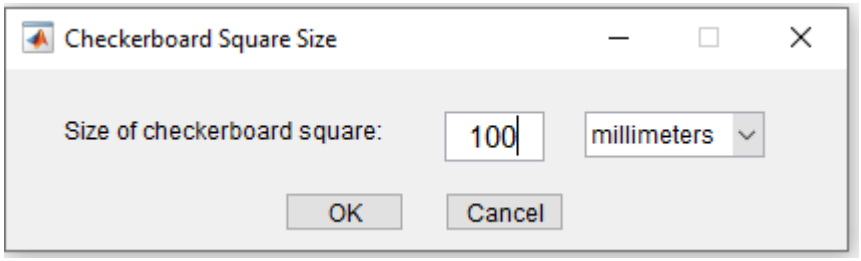

Figura 4.4 *Introducir tamaño marcadores*

- Hacer clic en *Calibrate* y ver resultados

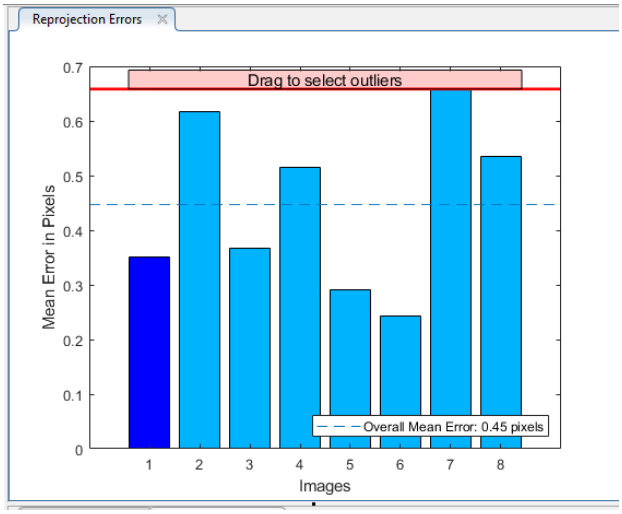

Figura 4.5 *Errores de reproyección en imágenes importadas*

Después podremos exportar los resultados y verlos en Matlab como objeto con más detalle:

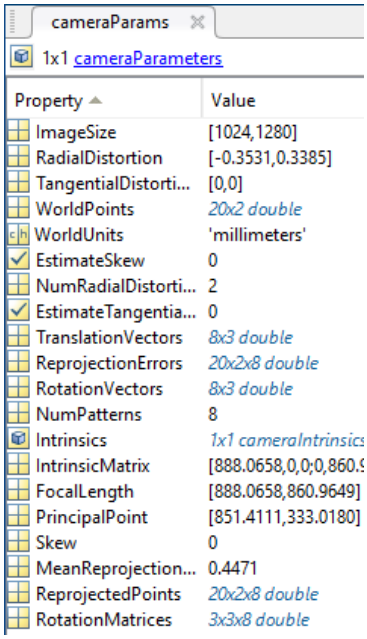

Figura 4.6 *Objeto resultante de la calibración*

Cabe mencionar que el *toolbox* permite configuraciones más avanzadas para, por ejemplo, computar más coeficientes de distorsión y *Skew* diferente de cero.

4.3 Calibración Extrínseca

La calibración extrínseca nos permite saber las transformaciones necesarias para cambiar del sistema de referencia del mundo al sistema de referencia de la cámara, con otras palabras, es la posición y orientación relativa de la cámara con respecto al mundo o a un sistema de referencia determinado. Esto suele venir dado por una translación en el espacio (vector 3D) y una rotación (vector 3D o matriz 3x3).

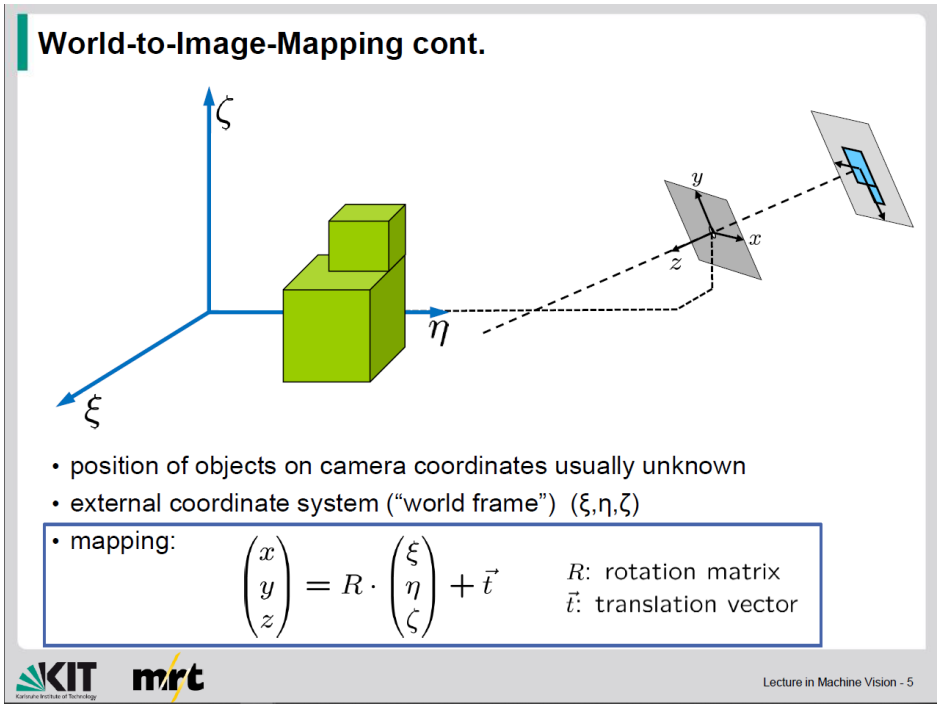

Figura 4.7 *Extrínsecos de una cámara. Machine Vision, KIT [16]*

En nuestro caso, ya comentamos que las cámaras se encuentran en las esquinas superiores de la estructura del laboratorio de 2,8 metros de lado, por ello, salvando variaciones por la montura de las cámaras, la separación entre ellas debe ser de esta misma distancia. Este ha sido uno de los criterios más sencillos para descartar o aceptar los métodos que se van a describir a continuación, ya que fue necesario probar varios de ellos hasta encontrar alguno que arrojase resultados sensatos.

Cabe mencionar que nos hemos basado en el TFG de Marcos Pérez Rus [17] para tomar ideas sobre estos métodos de calibración extrínseca. Por favor acudir a su proyecto si se requiere un análisis más detallado.

4.3.1 Patrón Aleatorio

Este método es, sobre papel, el mejor método de calibración. No solo promete calibrar las cámaras extrínsecamente sino también de forma intrínseca. Además, incluye calibración multicámara, lo que es perfecto para nuestro caso. Por tanto, este fue el primer método que se intentó de forma experimental.

Para ello, tal y como se indica en [17] debemos imprimir en un tamaño considerable (se ha probado con un formato A1) un patrón aleatorio creable con el propio toolbox de Matlab de Google [18].

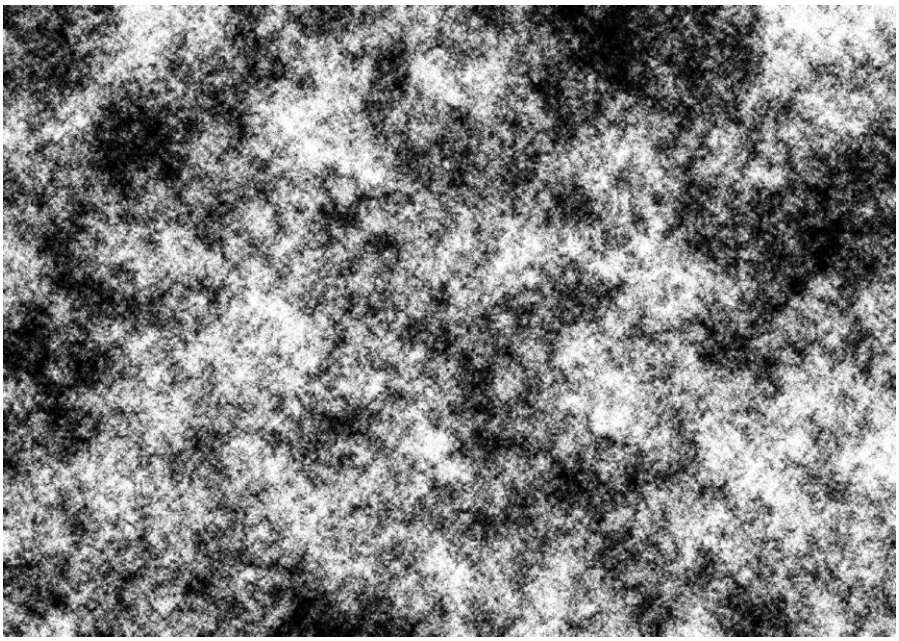

Figura 4.8 *Patrón Aleatorio*

Posteriormente se toman imágenes del patrón en diferentes posiciones de la escena y se nombran con el formato especificado en [18] (*camera_index-time_stamp.extension*) y se realiza su procesado.

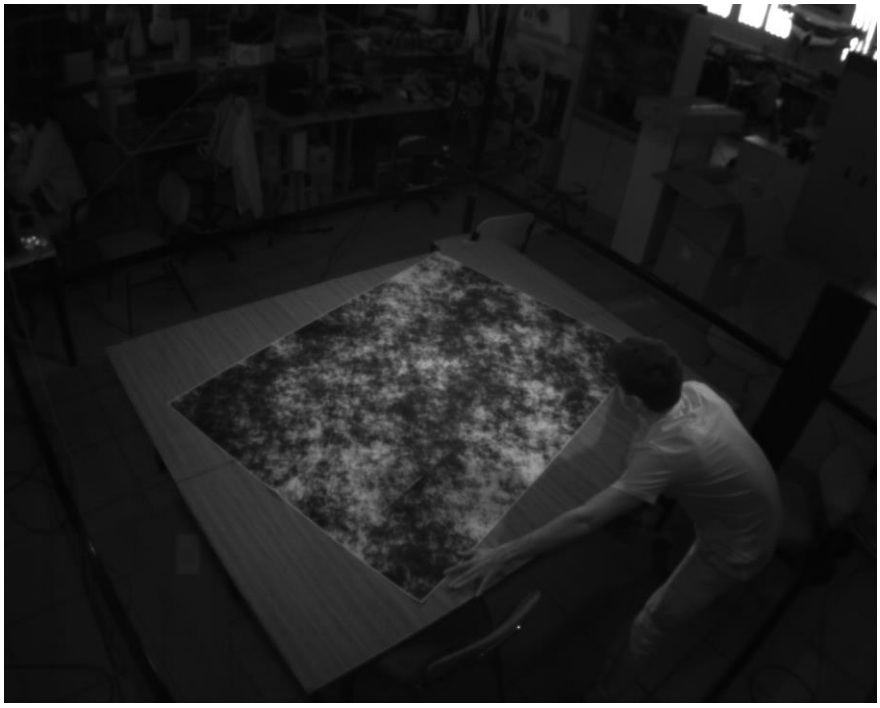

Figura 4.9 *Foto tomada con patrón aleatorio*

Se probó tanto con un patrón disponible en el laboratorio como con uno nuevo impreso, pero parece que las características de la plataforma no son las adecuadas para esta calibración, al ser muy sensible al ruido y requerir de notablemente de más detalle que el resto de métodos.

Lamentablemente la mayoría de las ocasiones este procesado fallaba y no se podía completar, lo cual imposibilitó su uso. Los resultados obtenidos además fueron bastante negativos, con distancias absurdas para la disposición de las cámaras en la plataforma. Tras muchos intentos este método fue descartado.

4.3.2 Patrón de Ajedrez – Toolbox Caltech [19]

Este es un toolbox de calibración estéreo, es decir, por par de cámaras. Necesitaremos un tablero de ajedrez que imprimiremos y tomaremos imágenes en varias posiciones, tal y como hicimos con la calibración intrínseca, pero necesitamos que el tablero sea visible en dos cámaras a la vez.

Figura 4.10 *Vista desde una de las cámaras*

Posteriormente debemos importar las imágenes en el toolbox nombrándolas con el formato *left.nImagen.jpg* y *right.nImagen.jpg* respectivamente para cada par de cámaras, en nuestro caso tenemos dos pares de cámaras, ya que vamos a referenciar siempre a una de ellas.

Figura 4.11 *Menú del toolbox [19][17]*

A diferencia del toolbox anterior, este nos pide que seleccionemos manualmente las esquinas del marcador de ajedrez en cada una de las imágenes, por lo que es un proceso bastante tedioso y para nada automatizable.

Una vez realizado esto tenemos algunas opciones avanzadas que nos podrán ayudar a mejorar la precisión de la calibración. Por ejemplo, el incluir una estimación inicial de coeficientes de distorsión, que podemos haber obtenido previamente con la calibración intrínseca. De forma general, se calculan tanto los parámetros intrínsecos como los extrínsecos.

También se permite recomputar las esquinas del marcador una vez ya se ha calibrado, para realizar una recalibración un poco más exacta.

En la práctica los resultados de este método tampoco han sido muy satisfactorios, por que se decidió desechar este toolbox también.

4.3.3 Patrón de ajedrez. *Stereo Camera Calibrator* [20]

Este es una herramienta del toolbox de Matlab de visión por computador, al igual que el *Camera Calibrator*. Con la diferencia de que este sirve para calibrar un par de cámaras, al igual que método antes explicado. La toma de imágenes es muy similar a la anterior y por tanto podemos usar las mismas imágenes siempre y cuando sean aceptadas por la propia herramienta (a la hora de importar puede rechazar algunas).

La importación se realiza de forma análoga a la de Camera Calibrator. Debemos seleccionar una carpeta para cada cámara, donde estén ordenadas cada una de las imágenes.

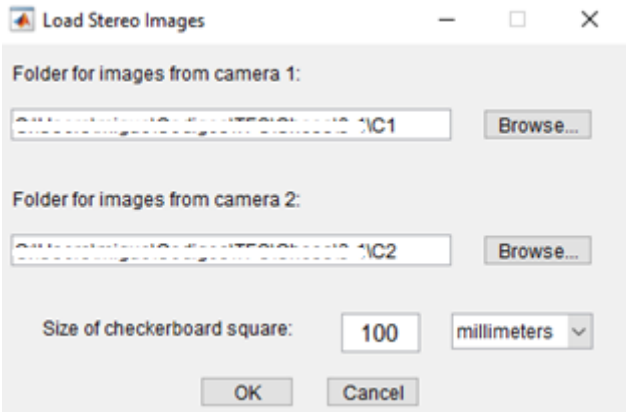

Figura 4.12 *Importación de imágenes*

Después solo debemos pulsar Calibrate y comprobar los resultados.

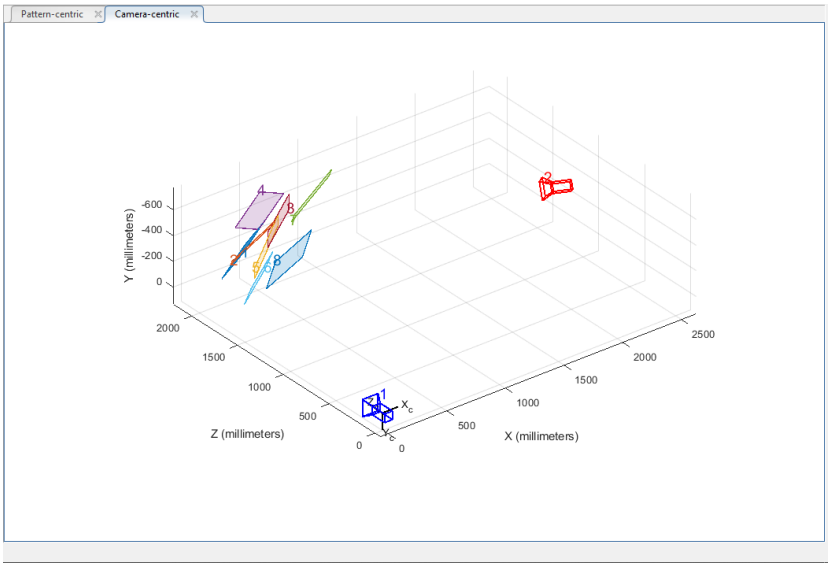

Figura 4.13 *Resultado gráfico calibración extrínseca*

Cabe destacar que es posible introducir previamente los parámetros intrínsecos de las cámaras para así mejorar la calibración extrínseca. De forma parecida a la calibración intrínseca podemos exportar los resultados y verlos con más detalle en Matlab:

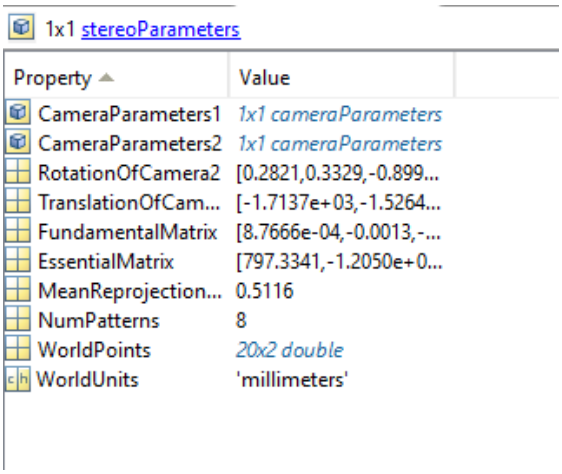

Figura 4.14 *Resultados calibración extrínseca*

Vemos que obtenemos la matriz de rotación y vector de translación entre las dos cámaras, entre otros muchos datos útiles. Para comprobar la validez de estos resultados podemos, por ejemplo, computar la norma de este vector de translación.

```
>>  norm(stereoParams.TranslationOfCamera2)

ans =

   2.8743e+03
```

Figura 4.15 *Norma vector translación*

Este método arrojó resultados sensatos, por tanto no fue descartado como los otros dos anteriores, sin embargo los frutos de las calibraciones de ambos pares de cámaras no eran muy parecidos. Esto hacía que las posiciones relativas entre ellas no fuesen correctas y estropeaba mucho la estimación al mezclar la información de las tres cámaras. Sobre todo en comparación con el método siguiente.

4.3.4 Marcadores fijos y PnP.

La idea de esta calibración extrínseca surgió más adelante en el proyecto, tras ya haber implementado gran parte de la estimación de pose con la calibración del apartado anterior.

El problema Perpective-n-Points[21] está bastante extendido y podemos encontrar una amplia gama de soluciones y algoritmos para él. Como ya se mencionó en la introducción, la cuestión a resolver es la estimación de la pose tridimensional de un objeto con geometría y proyección 2D conocidas.

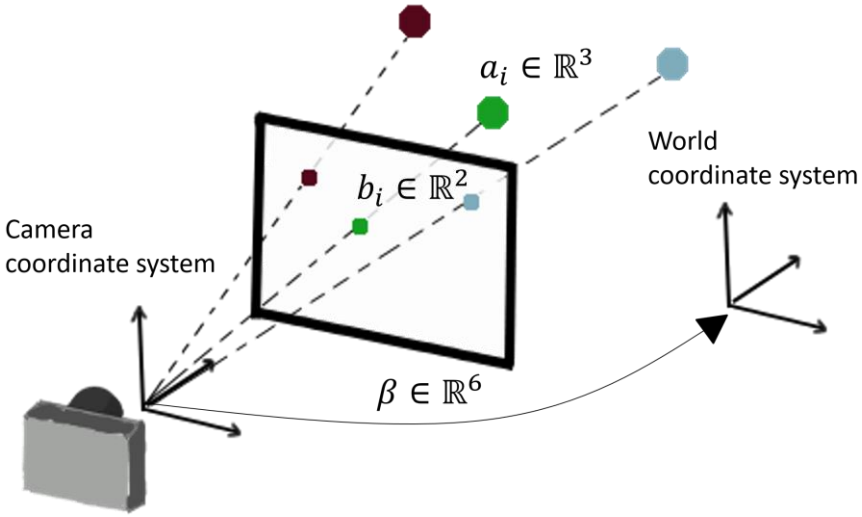

Figura 4.16 *PnP. PnP-Net: A hybrid Perspective-n-Point Network. Roy Sheffer, Ami Wiesel. (arXiv:2003.04626v1)*

Por tanto, para poder aplicar este método necesitamos conocer de antemano:

- Geometría del objeto (posición relativa marcadores)
- Proyección en 2D del objeto
- Parámetros intrínsecos y coeficientes de distorsión

En nuestro caso ya disponemos del tercer elemento de la lista, pasemos a ver qué se ha realizado para los otros dos. En la figura 4.1 podíamos ver unos marcadores sobre la mesa, estos han sido utilizados como geometría de la mesa, haciendo que esta sea el objeto sobre el que queremos calcular la pose relativa, dado que ni la mesa ni las cámaras se van a mover, podemos obtener así la posición relativa de las tres cámaras con respecto a la mesa, pudiendo utilizarla así como sistema de referencias globales.

Es importante tener en cuenta de que cuantos más marcadores y más distintiva sea la geometría, mejores resultados arrojará el PnP, por eso decidimos realizar un montaje como el que se observa en la siguiente imagen.

Figura 4.17 *Montaje marcadores mesa*

Vemos que no solo se dispusieron marcadores en el plano de la mesa, sino también en planos perpendiculares para dotar a la geometría de mayor peculiaridad. Estos marcadores después podrían ser eliminados, ya que este proceso de calibración solo es necesario realizarlo una vez.

Una vez hecho esto solo debemos aplicar un método PnP para obtener como resultado la transformación entre la mesa y la cámara. En nuestro caso hicimos que el marcador central de la mesa se correspondiese con el (0,0,0) y los ejes XYZ se dispusiesen de tal forma que la foto estuviese tomada desde el primer octante (tres ejes positivos).

OpenCV dispone de varias funciones para computar el PnP, en este proyecto usaremos en mayor parte:

◆ solvePnP()

```
bool cv::solvePnP ( InputArray  objectPoints,
                    InputArray  imagePoints,
                    InputArray  cameraMatrix,
                    InputArray  distCoeffs,
                    OutputArray rvec,
                    OutputArray tvec,
                    bool        useExtrinsicGuess = false ,
                    int         flags = SOLVEPNP_ITERATIVE
                  )
```

Python:

```
cv.solvePnP( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, flags]]]] ) -> retval, rvec, tvec
```

Figura 4.18 *Función PnP en OpenCV [22]*

Como vemos, esta función nos da como output un vector de translación y otro de rotación, que se corresponden con la transformación necesaria para pasar de coordenadas en el sistema de la mesa al de la cámara.

Sin embargo la forma más común de expresión de una rotación 3D es con una matriz 3x3, esto tiene fácil solución, solo debemos hacer uso de la función *Rodrigues*:

Figura 4.19 *Función Rodrigues [22]*

Esta función realiza la conversión entre vector y matriz de rotación. Ahora podríamos representar la transformación total (translación + rotación) como una matriz homogénea, esta tiene la siguiente estructura:

$$H = \begin{bmatrix} \boldsymbol{R}_{3x3} & \boldsymbol{t}_{3x1} \\ 0 & 1 \end{bmatrix}$$

Esta representación nos permite aplicar ambas transformaciones con una sola multiplicación matricial. Para pasar un punto de un sistema de referencia a otro:

$$\boldsymbol{P}_{r1} = \boldsymbol{H}\ x\ \boldsymbol{P}_{r2}$$

Donde P_{r1} y P_{r2} es el punto en el sistema de referencia 1 y 2 respectivamente y H es la matriz de transformación homogénea entre ambos.

Realizando esto para cada una de las cámaras podemos obtener una transformación para cada una con respecto al centro de la mesa, siendo esta el sistema de referencia "global" a partir de ahora.

5 PERCEPCIÓN

Este es uno de los apartados del proyecto que más trabajo y dedicación han requerido. En él, trato de procesar las imágenes procedentes de las cámaras para aislar e identificar inequívocamente los marcadores, pudiendo así mezclar la información de los tres sensores y realizar una estimación de la posición de estos.

Se realizaron pruebas con diferentes métodos, siendo este un proceso de prueba y error en gran medida (sobre todo los primeros pasos). Podemos dividir la percepción en tres subprocesos diferenciados:

- Segmentación y aislamiento de marcadores en la imagen
- Identificación de marcadores (*Labeling*)
- Estimación de pose

A todo esto se le suman filtrados constantes de resultados, ya que como podremos ver en esta sección, debido a las limitaciones de la plataforma, la inestabilidad de algunos de los algoritmos hacen que los resultados aportados en la fase de estimación de pose sean bastante errática en determinadas situaciones, llegando a cambiar en gran manera en situación estática si no se incorpora ningún tipo de filtrado.

Muchos de estos problemas podrían también solucionarse mediante el promediado de mediciones consecutivas, pero la baja tasa de muestreo de este procesado imposibilita esta opción. Experimentalmente se han podido llegar a obtener 2-3Hz una vez incluyendo todas las fases de procesamiento expuestas arriba. Esto choca bastante con la tasa de refresco teórica inicial de las cámaras, pero no ha sido posible encontrar manera de agilizar este proyecto, incluso tras bastante investigación en el SDK de Sapera para comprender su funcionamiento a un más bajo nivel.

5.1 Segmentación

En este subapartado se explica el proceso de tratamiento de la imagen para conseguir aislar los marcadores con el menor número de fotogramas erróneos posibles.

Como ya se comentó en la sección de *Hardware* disponemos de cámaras y focos infrarrojos, por tanto, se optó por emplear marcadores esféricos con un recubrimiento reflectante para intentar obtener una reflexión más o menos constante independientemente de la situación. Sin embargo, como ya también se comentó, el reflejo de radiación infrarroja es bastante diferente dependiendo de la posición de los marcadores, por eso se decidió reducir la zona útil de trabajo al centro de la mesa.

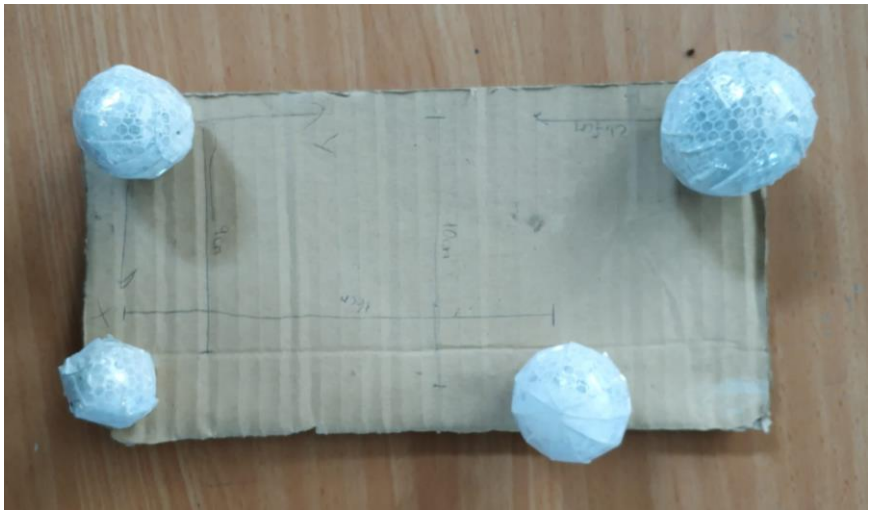

Figura 5.1 *Marcadores reflectantes(disposición 4)*

5.1.1 Proceso de aislamiento

Es importante saber cómo se ven estos marcadores en las cámaras antes de atacar el problema de aislamiento para elegir un método apropiado.

Figura 5.2 *Imagen tomada directamente de las cámaras sin post-procesado*

Vemos que, una vez configuradas las cámaras y focos de correcta, podemos observar los marcadores como "bolas" blancas que resaltan bastante del resto del entorno. Sin embargo, pueden existir reflejos u otros focos de luz, en este caso observamos que las ventanas de la esquina superior derecha también tienen un brillo considerable en comparación con el resto.

Para solucionar esto podemos realizar un recorte o *crop* de la imagen para intentar quedarnos solamente con nuestra región de interés, la mesa. De esta forma obtendremos algo parecido a:

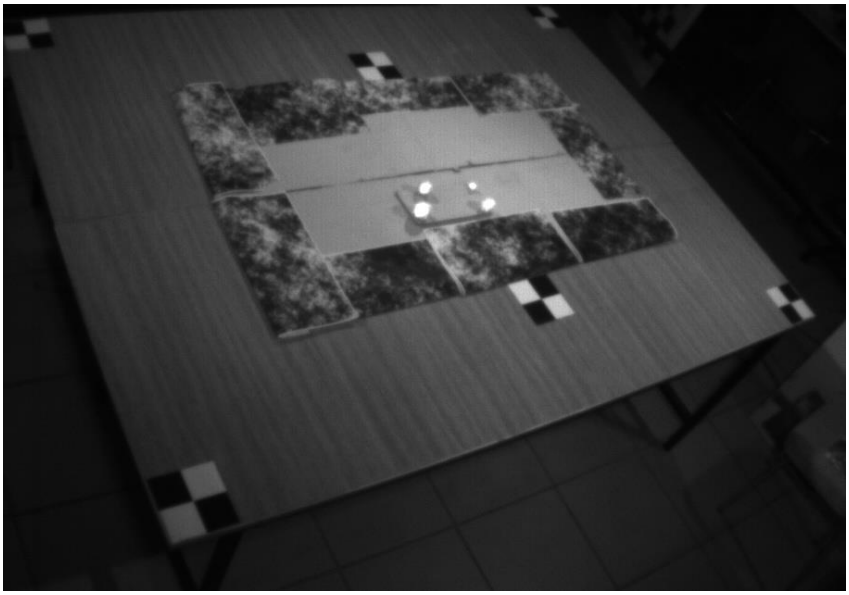

Figura 5.3 *Imagen recortada*

Esto es fácilmente realizable en OpenCV de la siguiente forma:

Código 4. Recorte Imagen

```
Mat src = imread(path);

const Rect roi(crop_left, crop_top, src.cols - crop_right - crop_left,
src.rows - crop_bot - crop_top);

src = src(roi).clone();
```

Esto también podría haberse realizado de una mejor forma y más exacta si se usase una máscara binaria para aislar la región de interés. Sin embargo, esta idea surgió más tarde y no hubo tiempo de probar si la mejora es sustancial. La mayoría de reflejos y problemas se eliminaban con este simple paso.

El siguiente paso es filtrar la imagen para suavizarla y mejorarla antes de proceder a la segmentación. Esto podemos encontrarlo en una de las guías de referencia empleadas de OpenCV [23] y también en [6], por tanto, tampoco se realizará mayor explicación aquí.

El resultado de esto debe tener un aspecto parecido a:

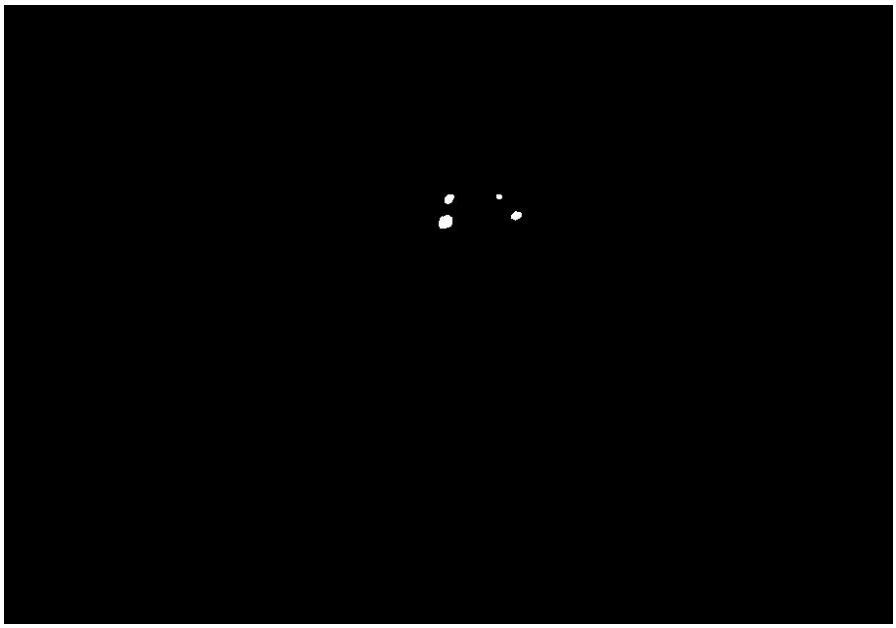

Figura 5.4 *Imagen filtrada y binarizada*

Como vemos es una imagen binaria que debe contener nuestros marcadores, aunque por reflejos o ruido pueden surgir algunos problemas como:

- Más contornos debido a reflejos de luz solar o similar (suelen no tener forma circular).
- Uno de los marcadores tiene dos reflejos muy juntos. Sucede ya que la reflexión en el material reflectante no es perfecta y pueden surgir artefactos.

Con respecto a este segundo problema se decidió aplicar un dilatado con el objetivo de eliminar estos dos contornos próximos. Se emplea un kernel de unos de 7x7 y la función *dilate* de OpenCV.

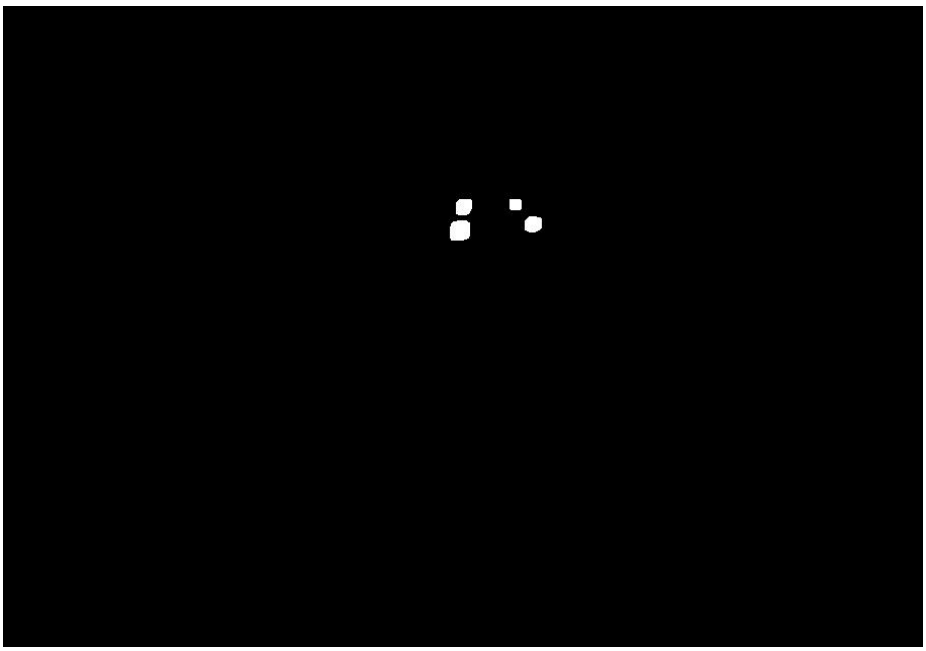

Figura 5.5 *Imagen binaria tras el dilatado*

Código 5. Dilatado

```
Mat transfMorfolog = imgResult.clone();
Mat kernel1 = Mat::ones(7,7, CV_8U);
dilate(transfMorfolog, transfMorfolog, kernel1);
```

Posteriormente, con la ayuda de *findContours* buscamos los contornos en esta imagen. Esta función nos los va a devolver por separado, por tanto, a partir de aquí es muy fácil calcular sus centros y otras características interesantes para la aplicación.

Código 6. Contornos

```
 vector<vector<Point>> contours_circ;
 findContours(transfMorfolog_8u, contours_circ, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);

 // Create the marker image
 Mat markers = Mat::zeros(transfMorfolog.size(), CV_32S);
 // Draw the foreground markers
 for (size_t j = 0; j < contours.size(); j++)
 {
 drawContours(markers, contours, static_cast<int>(j),
 Scalar(static_cast<int>(j) + 1), -1);
 }
 // Draw the background marker
 circle(markers, Point(5, 5), 3, Scalar(255), -1);
 Mat markers8u;
 markers.convertTo(markers8u, CV_8U, 10);
 int ncomp = contours.size();
 Mat drawing = original.clone();
 if (ncomp > 0) {
  vector<Moments> mu(ncomp);
  vector<Point2f> mc(ncomp);
  vector<float> area(ncomp);
  vector<float> pos_X(ncomp);
  vector<float> pos_Y(ncomp);
  vector<vector<Point>> finalContours;

  for (int seed = 1; seed <= contours.size(); ++seed)
  {
   cv::Mat1b mask = (markers == seed);
   findContours(mask, finalContours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);

   mu[seed - 1] = moments(finalContours[0], false);
   mc[seed - 1] = Point2f(mu[seed - 1].m10 / mu[seed - 1].m00,
   mu[seed - 1].m01 / mu[seed - 1].m00); //Centro de masas

   pos_X[seed - 1] = mc[seed - 1].x;
   pos_Y[seed - 1] = mc[seed - 1].y;
   area[seed - 1] = mu[seed - 1].m00;

   circle(drawing, centro, radio, Scalar(255, 0, 255));
   drawContours(drawing, finalContours, -1, Scalar(120, 120, 120), 1, 8,
   noArray(), 0, Point());
  }

}
```

No solo pintamos el borde del contorno para poder identificarlo visualmente, sino que también calculamos la posición de su centro en (X, Y) y su área. Esto no servirá posteriormente para poder utilizar su tamaño o posición relativa como parámetro en la función de identificación.

Sin embargo, estos contorno pueden cambiar de forma de un fotograma a otro aunque aparentemente todo esté estático, esto se debe a ruidos y errores que se producen en la cámara. Para reducir este efecto, en vez de calcular el centro del contorno directamente, se decidió hacer uso de la función *minEnclosingCircle*. Esta calcula el círculo de menor área que inscribe todo el contorno dentro de él.

Código 7. Contornos con *minEnclosingCircle*

```
Point2f centro; float radio;
minEnclosingCircle(finalContours[0], centro, radio);
pos_X[seed - 1] = centro.x;
pos_Y[seed - 1] = centro.y;
area[seed - 1] = M_PI * pow(radio,2) ;
circle(drawing, centro, radio, Scalar(255, 0, 255));
drawContours(drawing, finalContours, -1, Scalar(120, 120, 120),
1, 8, noArray(), 0, Point());
```

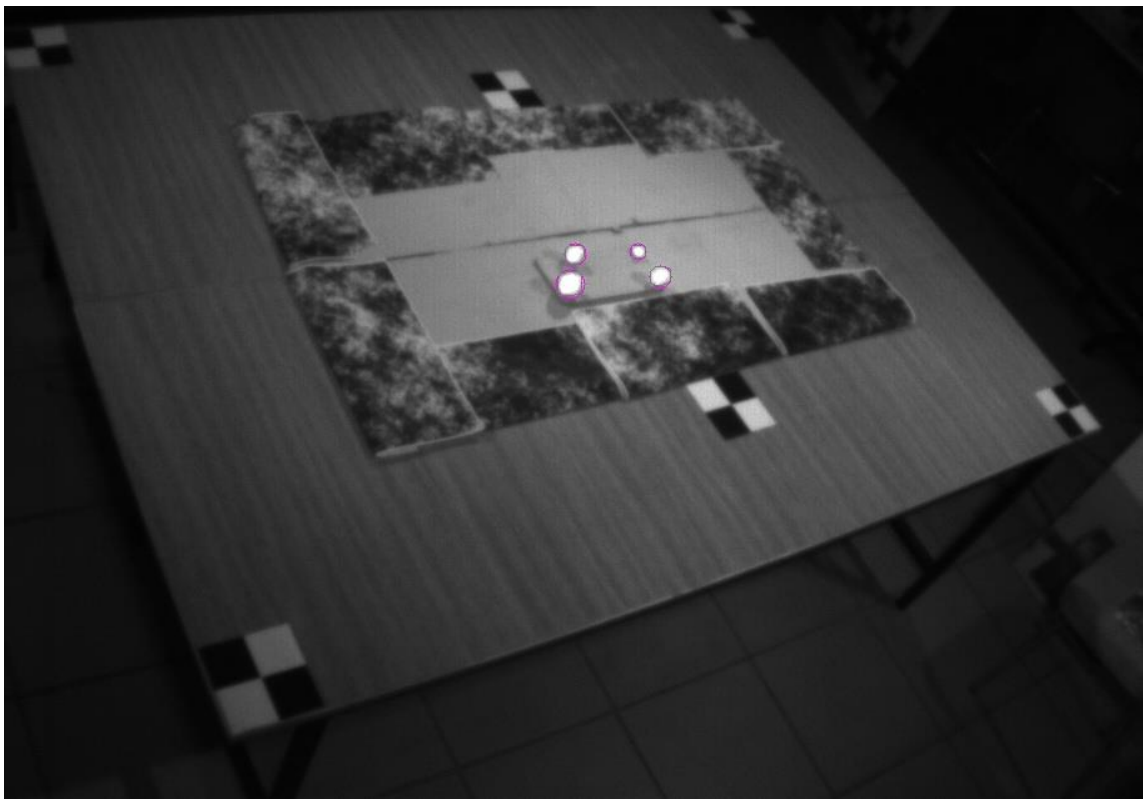

Figura 5.6 *Contornos dibujados*

5.1.2 Filtrado de circularidad

Para solucionar otro de los problemas más comunes, los reflejos indeseados, podemos aplicar un filtrado por circularidad de los contornos encontrado. Supongamos que tenemos la siguiente imagen sobre la que he pintado un artefacto manualmente de forma alargada:

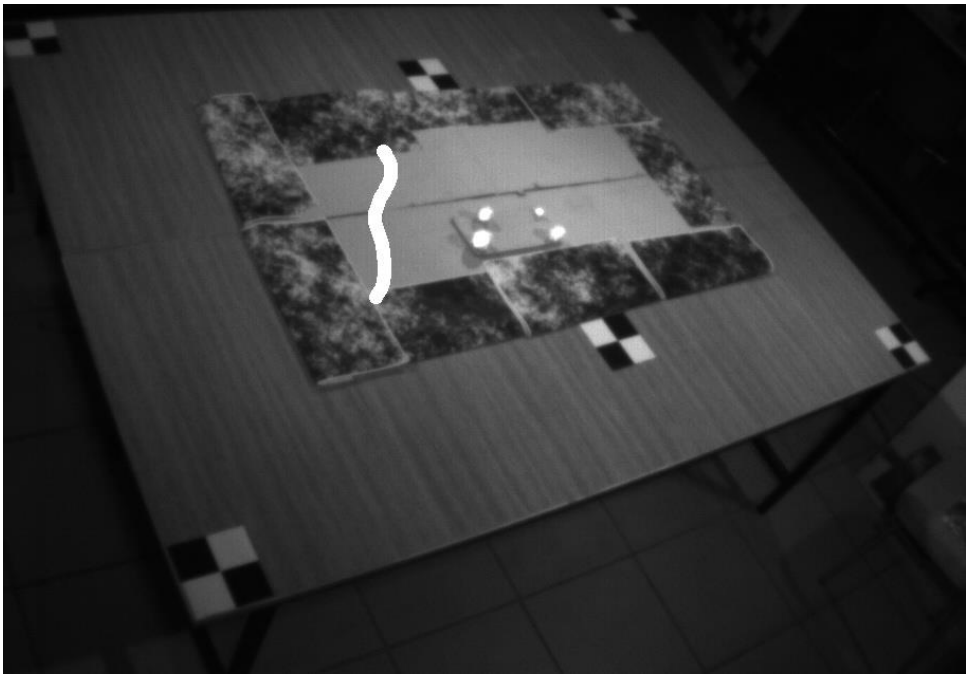

Figura5.7 *Imagen con artefacto*

Tras realizar el procesado obtenemos la siguiente imagen binaria.

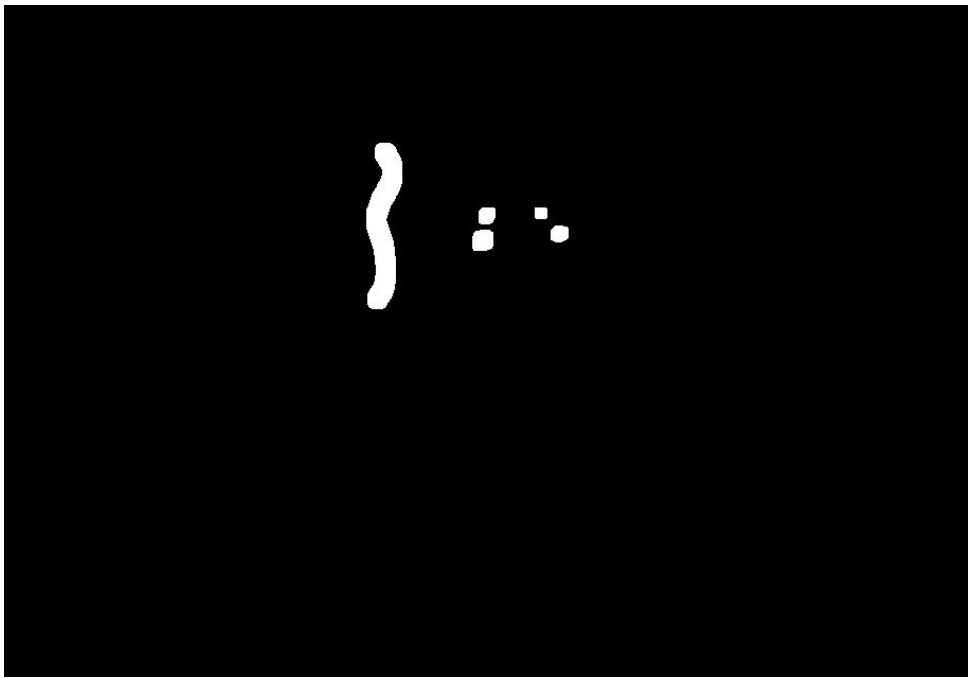

Figura 5.8 *Imagen binaria con artefacto*

Lo cual lleva a que se trate el artefacto como uno de los marcadores de forma errónea, para librarnos de él proponemos un uso de filtrado por circularidad. Tras pruebas de implementación, la que mejores resultados ha dado ha sido filtrar si se encuentran más de N contornos en la imagen. Siendo N el número de marcadores que estamos empleando.

No se filtra siempre ya que los contornos de los marcadores pueden ser más o menos circulares debido a la reflexión, por tanto, imponer unas cotas absolutas arrojaba bastantes errores. Este filtro se implementa inmediatamente después de calcular los contornos de la imagen binaria y antes de hacer los cálculos de posición y pintar su perímetro.

Código 8. Filtrado por circularidad

```
//Filtrado por circularidad
if (contours_circ.size() > NBOLAS) {
	vector<double> circularity(contours_circ.size());
	vector<double> circularity_sorted(contours_circ.size());

	for (int j = 0; j < contours_circ.size(); j++) {
		double perimeter = arcLength(contours_circ[j], true);
		double area = contourArea(contours_circ[j]);

		if (perimeter > 0) {
			circularity[j] = 4 * M_PI * (area / (perimeter * perimeter));
			circularity_sorted[j] = circularity[j];
		}

	}

	sort(circularity_sorted.begin(), circularity_sorted.end(), greater<double>());

	for (int j = 0; j < NBOLAS; j++) {
		int s = find(circularity.begin(), circularity.end(),
		circularity_sorted[j]) - circularity.begin();
		contours.push_back(contours_circ[s]);
	}

}
else
{
	contours = contours_circ;
}
```

Aquí podemos ver el resultado tras aplicar este filtrado, el contorno introducido artificialmente se ignora y no se trata como marcador.

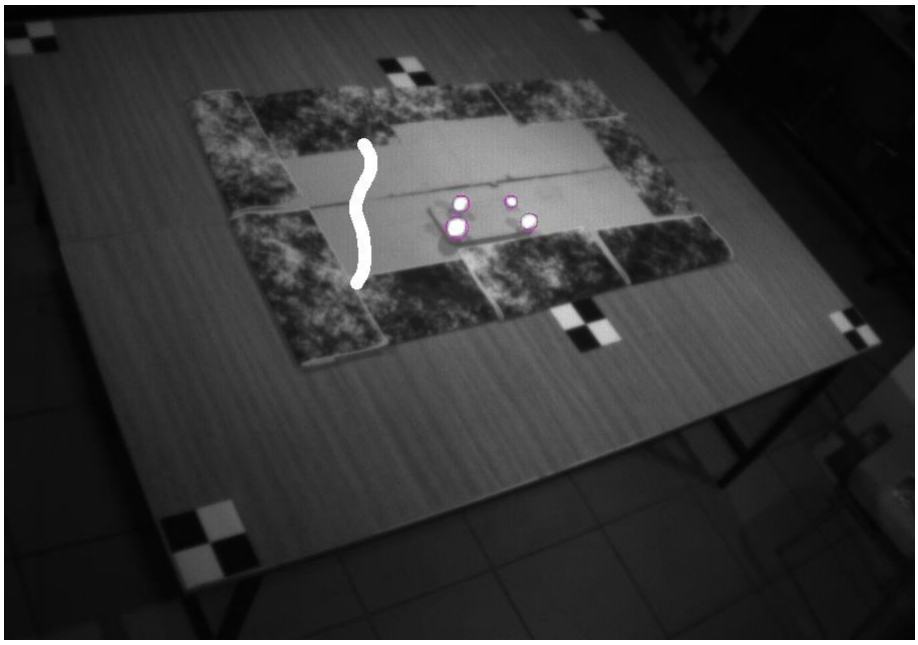

Figura 5.9 *Resultado con artefacto y filtro*

Mientras que, si no lo hubiésemos aplicado, obtendríamos:

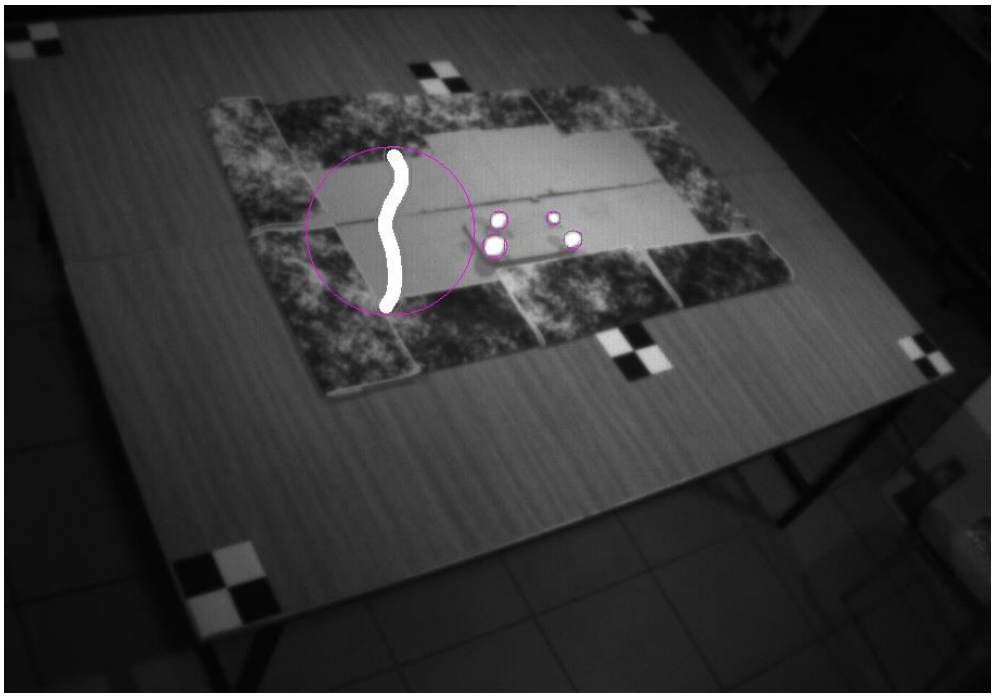

Figura 5.10 *Resultado con artefacto sin filtro*

5.2 Identificación de marcadores (*Labeling*)

En este apartado veremos el proceso que nos permite poner "nombre" a cada uno de los marcadores y en cada una de las cámaras. Esto es necesario para que la estimación de posición sea siempre referida al mismo punto, que es un requisito indispensable para los dos algoritmos que se usarán.

En la figura 5.1 vimos una disposición con cuatro marcadores, sin embargo, con el fin de comparar diferentes métodos, también se realizó una con tres marcadores.

Figura 5.11 *Disposición 3 marcadores*

Vemos en ambas geometrías que no puede haber dos marcadores en disposiciones relativas iguales o parecidas, es decir, debemos evitar, entre otros, geometrías simétricas ya que esto hace que la identificación de puntos sea bastante más laboriosa.

5.2.1 Cuatro Marcadores

Se ha seguido el siguiente criterio para la identificación:

1. Marcador con menos área.
2. Marcador más alejado de 1.
3. Marcador más cercano a 1.
4. Marcador restante.

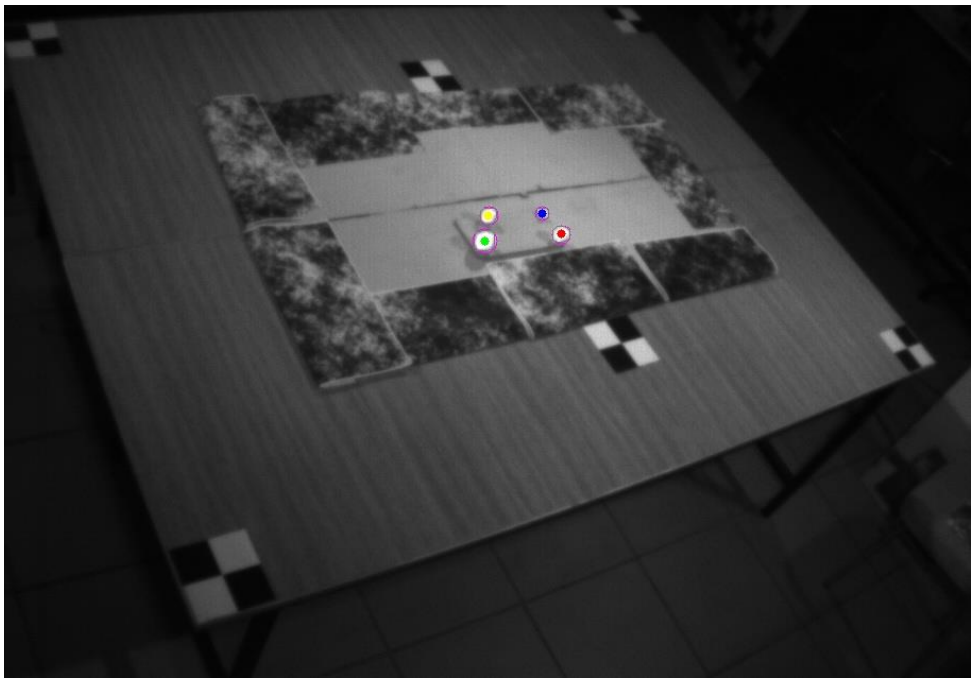

Figura 5.12 *Resultado Labeling 4 Marcadores*

Este ha sido el algoritmo que mejores resultados ha arrojado sin tener que incrementar en demasía la complejidad del mismo.

Código 9. *Labeling* 4 Marcadores

```
int ord_vec[] = { min_element(area.begin(),area.end()) - area.begin(), -1, -1, -1 };
vector<int> indexes = { 0,1,2,3 };
remove_copy(indexes.begin(), indexes.end(), indexes.begin(), ord_vec[0]);
float distancias[3] = { pow(pos_X[indexes[0]] - pos_X[ord_vec[0]], 2) +
pow(pos_Y[indexes[0]] - pos_Y[ord_vec[0]], 2),

pow(pos_X[indexes[1]] - pos_X[ord_vec[0]], 2) + pow(pos_Y[indexes[1]] -
pos_Y[ord_vec[0]], 2),

pow(pos_X[indexes[2]] - pos_X[ord_vec[0]], 2) + pow(pos_Y[indexes[2]] -
pos_Y[ord_vec[0]], 2) };


if (distancias[0] > distancias[1] && distancias[0] > distancias[2]) {
      ord_vec[1] = indexes[0];
}
else if (distancias[1] > distancias[0] && distancias[1] > distancias[2]) {
      ord_vec[1] = indexes[1];
else {
      ord_vec[1] = indexes[2];
}

if (distancias[0] < distancias[1] && distancias[0] < distancias[2]) {
      ord_vec[2] = indexes[0];
}
else if (distancias[1] < distancias[0] && distancias[1] < distancias[2]) {
      ord_vec[2] = indexes[1];
}
else {
      ord_vec[2] = indexes[2];
}
remove_copy(indexes.begin(), indexes.end(), indexes.begin(), ord_vec[1]);
remove_copy(indexes.begin(), indexes.end(), indexes.begin(), ord_vec[2]);
ord_vec[3] = indexes[0];
```

Como vemos se hace uso de vectores y de funciones sobre los mismos. El vector auxiliar *indexes* es necesario ya que no solo queremos, por ejemplo, ordenar de mayor a menor el vector de áreas de los marcadores, sino que también queremos saber a qué índice se correspondían inicialmente cada una de las posiciones, ya que es necesario para después emplear el criterio de distancia. Debido a esto queda un algoritmo un poco más engorroso en cuanto a programación, pero bastante sencillo a nivel conceptual.

5.2.2 Tres Marcadores

La identificación en este caso es bastante más sencilla:

1. Marcador más alejado al de menor área.
2. Marcador más cercano al de menor área.
3. Marcador de menor área.

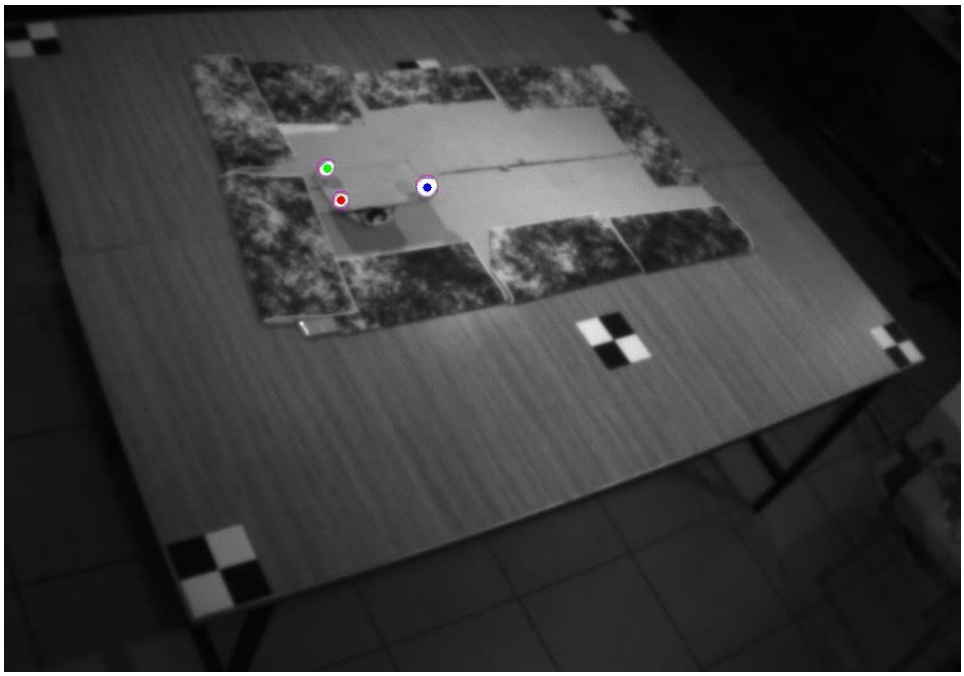

Figura 5.13 *Resultado Labeling 3 Marcadores*

Código 10. ***Labeling 3 Marcadores***

```
int ord_vec[NBOLAS] = { -1, -1 ,min_element(area.begin(),area.end()) - area.begin() };
vector<int> indexes = { 0,1,2 };
remove_copy(indexes.begin(), indexes.end(), indexes.begin(), ord_vec[2]);

if (pow(pos_X[indexes[0]] - pos_X[ord_vec[2]], 2) + pow(pos_Y[indexes[0]] -
pos_Y[ord_vec[2]], 2) > pow(pos_X[indexes[1]] - pos_X[ord_vec[2]], 2) +
pow(pos_Y[indexes[1]] - pos_Y[ord_vec[2]], 2)) {
	ord_vec[0] = indexes[0];
	ord_vec[1] = indexes[1];
}

else {
	ord_vec[1] = indexes[0];
	ord_vec[0] = indexes[1];
}
```

```
for (int j = 0; j < contours.size(); j++) {
	Scalar color = colores[j];
	circle(drawing, mc[ord_vec[j]], 4, color, -1, 8, 0);
	imPoints[i].push_back(Point2f(pos_X[ord_vec[j]] + crop_left[i],
	pos_Y[ord_vec[j]] + crop_top[i]));

}
```

Si realizamos pruebas de ambas geometrías de marcadores a tiempo real en el laboratorio, podremos ver que, dependiendo de la zona de la mesa, en general la disposición de tres marcadores parece un poco más estable, arrojando con menos frecuencia un *Labeling* en alguna de las tres cámaras, ya que para que el set de imágenes sea válido, es necesario que en las tres imágenes los marcadores estén identificados de forma correcta.

Este resultado es esperable, ya que, al disponer de más elementos, es más probable que alguno de ellos falle y su detección sea errónea.

5.2.3 Filtro Proximidad

Tras la identificación de marcadores y antes de la estimación de posición se decidió instalar un filtro con la intención de mejorar el rendimiento del algoritmo frente a escenas estáticas. Debido a la sensibilidad de los algoritmos de estimación de pose, cualquier mínimo cambio en el centro de los marcadores o detección errónea de los mismos se magnificaba y resultaba en una estimación bastante errática y con una inestabilidad considerable.

Por ello, una vez identificados los marcadores, si la posición de estos varía por debajo de un rango impuesto (en píxeles), nos quedamos con la última posición calculada. Esto no elimina las pequeñas variaciones por ruido de las cámaras. Si imponemos también un límite superior, podemos filtrar también *outliers* resultado de perturbaciones mayores, pero esto también corre en nuestra contra, ya que, si en algún momento se acepta una medición errónea como válida, ya no cambia y se quedará atrapada ahí.

Naturalmente esto podría complicarse más, haciendo que se comparasen más de dos mediciones, realizando promediados, etc.

Código 12. Filtro proximidad

```
for (int i = 0; i < 3; i++) {
	for (int j = 0; j < NBOLAS; j++) {
		float distancia = twoDPointsDistance(imPoints[i][j],
		 imPoints_previous[i][j]);

		if (distancia < MIN_DIST || distancia > MAX_DIST) {
			imPoints[i][j] = imPoints_previous[i][j];
		}
	}
}
imPoints_previous = imPoints;
```

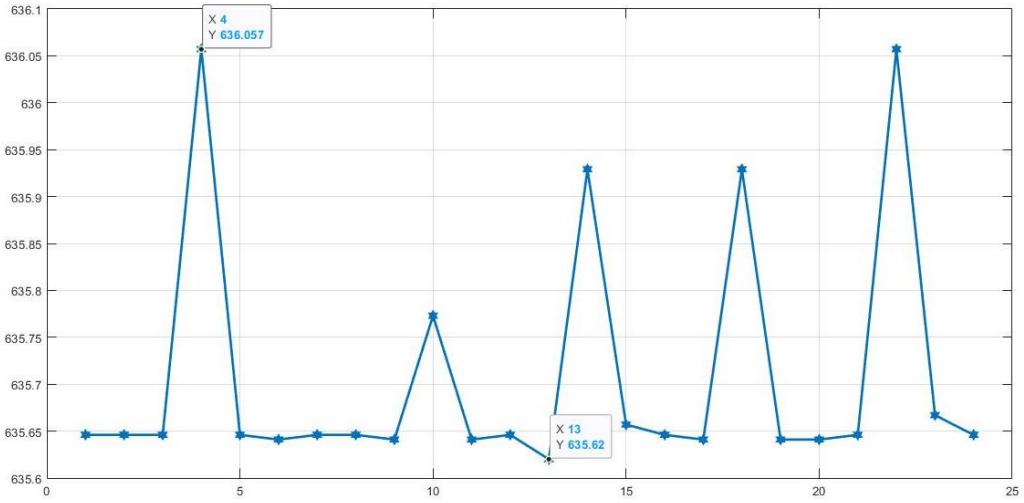

Figura 5.14 *Vibraciones estáticas en posición X de un marcador. Nº Frame – Posición [pix].*

En la gráfica de arriba podemos ver las vibraciones a nivel estático de la posición en X del centro de uno de los marcadores. En este caso es de alrededor de 0,5 pix, pero se han llegado a observar de hasta poco más de un píxel en el laboratorio.

Esto sumado a la variación en Y, provocaban la inestabilidad magnificada comentada en el inicio de este subapartado. Con el filtro podemos conseguir aplanar esta curva, a coste teórico de detectar de peor forma movimientos pequeños, pero debido a la baja tasa de refresco del algoritmo, lo movimientos tan sutiles son prácticamente inexistentes en esta aplicación.

5.3 Estimación de pose

Una vez ya obtenidos los centros de marcadores y haberlos ordenado e identificado, solo queda aplicar algún algoritmo para mezclar la información obtenida de las tres cámaras en una sola estimación.

En este TFG se van a cubrir dos casos:

- *Perspective-n-Points* (PnP), problema ya mencionado y explicado en el apartado de calibración extrínseca.
- Triangulación de puntos por par de cámaras.

5.3.1 PnP

5.3.1.1 Estimación

Este es el caso de uso más común del PnP. Se realiza una aplicación muy similar a la vista en [6], primero se ejecuta el algoritmo PnP con el método correspondiente para cada una de las tres cámaras. Esto nos da como resultado la transformación en el espacio entre el objeto de los marcadores y cada una de las cámaras. Debemos tener en cuenta que existen dos escenarios posibles, una geometría con tres marcadores y otra con cuatro. La forma más sencilla de solucionar esto es mediante software. Debemos indicar antes de ejecutar cuál es el número de marcadores que vamos a emplear (modificar *N_BOLAS*).

Código 13. Ejecución algoritmo PnP

```
void PNP(int camara, vector<Point2f> imagePoints, Mat& rmat, Mat& tvec, Mat& rvec)
{
	Mat cameraMatrix, distCoeffs;
	Mat cameraMatrix1 = (Mat_<float>(3, 3) <<
		943.3056, 0, 681.2714,
		0, 947.5338, 489.8241,
		0, 0, 1.0000);
	Mat distCoeffs1 = (Mat_<float>(4, 1) << -0.2713, 0.0183, 0, 0);

	Mat cameraMatrix2 = (Mat_<float>(3, 3) <<
		965.5391, 0, 613.7441,
		0, 965.8018, 519.5804,
		0, 0, 1.0000);
	Mat distCoeffs2 = (Mat_<float>(4, 1) << -0.3045, 0.1392, 0, 0);

	Mat cameraMatrix3 = (Mat_<float>(3, 3) <<
		959.0305, 0, 678.4766,
		0, 956.9934, 518.3325,
		0, 0, 1.0000);
	Mat distCoeffs3 = (Mat_<float>(4, 1) << -0.3149, 0.1334, 0, 0);

	switch (camara) {
	case 1:
		cameraMatrix = cameraMatrix1;
		distCoeffs = distCoeffs1;
		break;
	case 2:
		cameraMatrix = cameraMatrix2;
		distCoeffs = distCoeffs2;
		break;
	case 3:
		cameraMatrix = cameraMatrix3;
		distCoeffs = distCoeffs3;
		break;
	}
	vector<Point3f> objectPoints;
	vector<Point2f> imagePoints_repro;
	vector<Point3f> objectPoints_repro;
```

```
	if (NBOLAS == 3) {
		objectPoints = {Point3f(0.1,0.215,0), Point3f(0,0,0), Point3f(0.1,0,0)};
		vector<Mat> rvec2, tvec2;
		solveP3P(objectPoints, imagePoints, cameraMatrix, distCoeffs,
		rvec2, tvec2, SOLVEPNP_P3P);
		rvec = rvec2[0];
		tvec = tvec2[0];
	}

	else if (NBOLAS == 4) {
		objectPoints = { Point3f(0.09,0,0), Point3f(0,0.215,0),
		Point3f(0,0,0), Point3f(0.1,0.16,0) };

		solvePnP(objectPoints, imagePoints, cameraMatrix, distCoeffs,
		rvec, tvec, false, SOLVEPNP_ITERATIVE);
	}

	Rodrigues(rvec, rmat);
```

Como vemos se han incluido los parámetros de la calibración intrínseca de cada una de las cámaras (*cameraMatrix* y *distCoeffs*) y se define la geometría del objeto de marcadores una vez se sepa con cuantos se trabaja (*objectPoints*). Finalmente se pasa de vector a matriz de rotación con la función *Rodrigues*.

Posteriormente pasamos las transformaciones estimadas al sistema de referencia de la cámara 1 con la ayuda de las matrices de transformación obtenidas con la calibración extrínseca y *changeRefSystem*. Esta es una función personalizada que realiza la transformación homogénea necesaria, pasando las matrices y vector a matrices homogéneas en *Eigen*, computando las operaciones necesarias y volviendo a separar rotación de translación. Esta transformación a Eigen es un intento posterior de optimización, ya que las operaciones matriciales en Eigen pueden llegar a ser hasta 10 veces más rápidas que en OpenCV.

```
 changeRefSystem(mRotcamera[1], mTranscamera[1],
 mRotcameraC1[1], mTranscameraC1[1], c1Tc2);

 changeRefSystem(mRotcamera[2], mTranscamera[2],
 mRotcameraC1[2], mTranscameraC1[2], c1Tc3);
```

Código 14. Función *changeRefSystem*

```
void changeRefSystem(Mat srcRMat, Mat srcTVec, Mat& dstRmat, Mat& dstTVec,
Eigen::Matrix4d transf) {

	// Las operaciones con Eigen son mucho mas rapidas que con OpenCV
	// convertirmos a eigen
	Eigen::Matrix3d srcR;
	cv2eigen(srcRMat, srcR);
	Eigen::Vector3d srcT;
	cv2eigen(srcTVec, srcT);

	//Pasamos la rotacion y translacion a matriz homogenea
	Eigen::Matrix4d srcHom = makeHomogMatrix(srcR, srcT);

	//Operamos para multiplicar
	Eigen::Matrix4d dstHom = transf * srcHom;

	Eigen::Matrix3d dstR;
	Eigen::Vector3d dstT;
	dstR = dstHom.block(0, 0, 3, 3);
	dstT = dstHom.block(0, 3, 3, 1);

	// Pasamos a Mat
	eigen2cv(dstR, dstRmat);
	eigen2cv(dstT, dstTVec);
}
```

5.3.1.2 Promediado

Ahora, ya que los resultados no son exactamente los mismos debido a errores de calibración e imperfecciones experimentales, debemos realizar un promediado tanto de la posición como de la orientación del objeto de marcadores. Para la posición se aplica una simple media de las coordenadas, pero para la orientación haremos uso de una herramienta matemática, el promediado de cuaternios.

Un *cuaternión* (o *cuaternio*) es un formato de expresión de giros y orientaciones bastante extendido en la ingeniería. Son una expresión más sencilla de componer que los *ángulos de Euler* y permiten operaciones de una forma más sencilla a parte de tener otras ventajas como evitar el famoso *Gimbal Lock*.

Una rotación alrededor de un eje $\boldsymbol{v}$ (vector 3D unitario) con un ángulo Φ puede expresarse como un cuaternio (q) de la siguiente forma:

$$\boldsymbol{v} = [a\ b\ c]^T \ \ donde\ \left|\left|\boldsymbol{v}\right|\right| = 1$$

$$q = \cos\left(\frac{\Phi}{2}\right) + \boldsymbol{v}\ \cdot sen\left(\frac{\Phi}{2}\right) = \ \cos\left(\frac{\Phi}{2}\right) + \boldsymbol{i}\ a\ sen\left(\frac{\Phi}{2}\right) + \ \boldsymbol{j}\ b\ sen\left(\frac{\Phi}{2}\right) + \ \boldsymbol{k}\ c\ sen\left(\frac{\Phi}{2}\right)$$

En muchas herramientas se realiza su representación como vectores de la siguiente forma:

$$q = \left[a\ sen\left(\frac{\Phi}{2}\right), b\ sen\left(\frac{\Phi}{2}\right), c\ sen\left(\frac{\Phi}{2}\right), \cos\left(\frac{\Phi}{2}\right)\right]^T$$

Eigen tiene un objeto propio para definir los cuaternios [25], se basa en esta última definición vectorial:

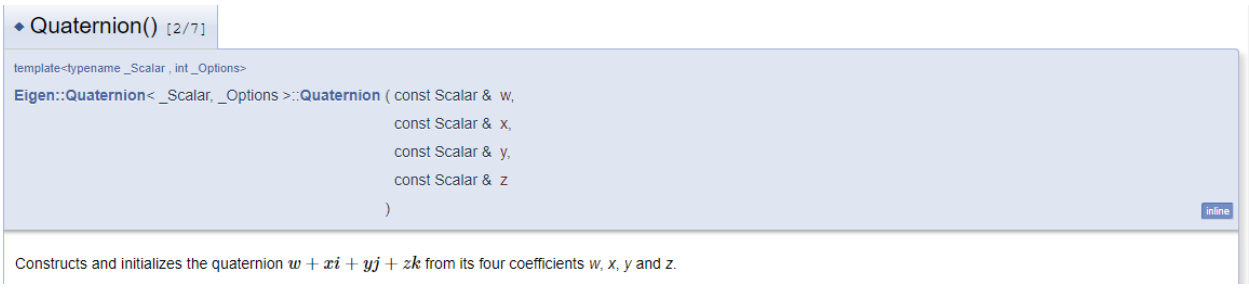

Figura 5.15 *Cuaternios en Eigen [25]*

El promediado de cuaternios aplicado en el proyecto se basa en lo explicado en *Averaging Quaternions*, de F.Landis Markley, Yang Cheng, John L.Crassidi y Yaakov Oshman [24] y utilizaremos una implementación realizada por Elisa Hidalgo en [6] con mínimas modificaciones. Para una explicación en más profundidad y de forma más matemática, refiero a estas dos últimas referencias.

A modo de resumen, si definimos *M* como:

$$M = \sum_{i=1}^{n} w_i q_i q_i^T \tag{5.1}$$

Podemos obtener el cuaternio promedio del autovector correspondiente al mayor autovalor de M. Ya que como se indica en [24] la solución a:

$$\bar{q} = argmax(q^T M q) \tag{5.2}$$

Es bien conocida e igual al autovector del mayor autovalor.

5.3.1.3 Cálculo de peso de ponderaciones

Se hace necesario todavía otorgar unos pesos a cada uno de los cuaternios, es decir imponer una importancia relativa a las estimaciones de cada cámara. Para esto se ha decidido dar un peso a cada medida de manera inversamente proporcional al error de reproyección cuya cámara comete en el set de imágenes que están siendo procesadas.

El error de reproyección se calcula tras la estimación de posición del PnP con la ayuda de la función *projectPoints.* Esto es un añadido a la función PnP del código 13.

Código 15. Cálculo error de reproyección

```
projectPoints(objectPoints, rvec, tvec, cameraMatrix, distCoeffs, imagePoints_repro);
*errorProm = 0;
for (int i = 0; i < NBOLAS; i++) {
      *errorProm += float(sqrt(pow(imagePoints_repro[i].x - imagePoints[i].x, 2) +
      pow(imagePoints_repro[i].y - imagePoints[i].y, 2)));
}
*errorProm /= (float)NBOLAS;
```

La función projectPoints realiza una proyección de unos puntos en el espacio, para ello necesita la transformación recién calculada. Posteriormente se compara con los puntos obtenidos en el procesado de imágenes gracias a la segmentación e identificación de marcadores y se realiza una media de todos los marcadores.

Esto se realiza en cada iteración para cada una de las cámaras, pudiendo así después, otorgar unos pesos correspondientes.

Hemos seguido una distribución parecida a la de este gráfico para la asignación de pesos:

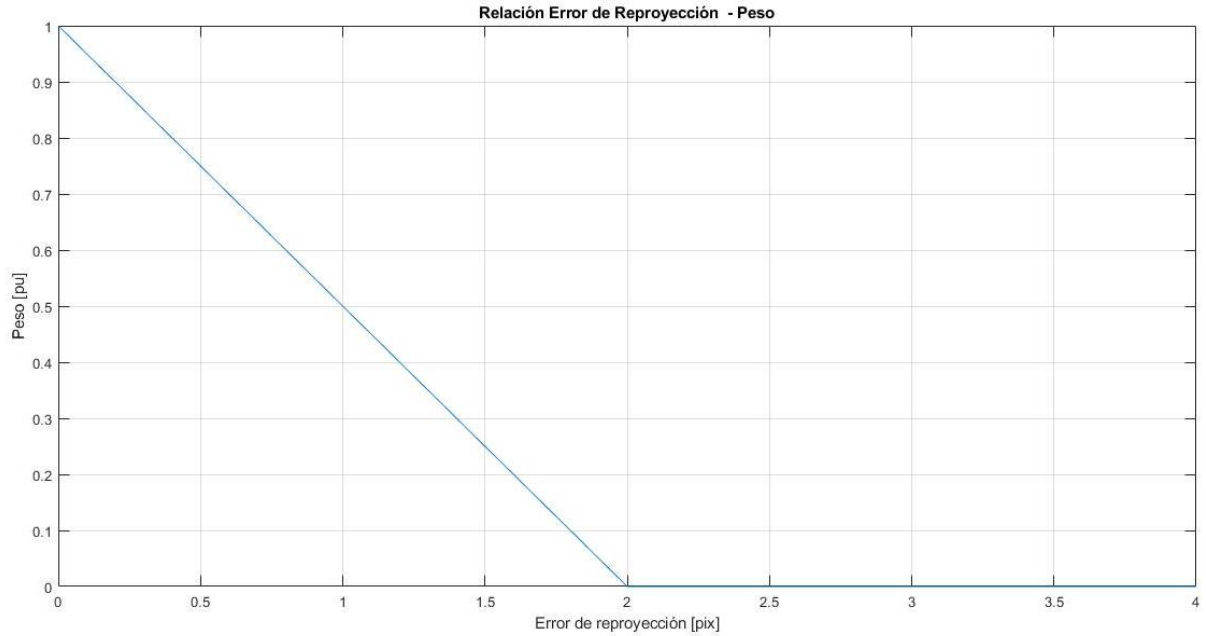

Figura 5.16 *Peso en función del error de reproyección*

Realizamos una distribución lineal desde los 0 a los 2 píxeles de error de reproyección, ya que experimentalmente no hemos obtenido mayor error que este. Si fuese mayor, damos la medida como inválida imponiendo un peso de 0.

Posteriormente debemos normalizar los pesos para que puedan ser aplicables a la ponderación:

Código 16. Cálculo de ponderaciones

```
float w[3];

for (int j = 0; j < 3; j++) {
	w[j] = 1.0 - errorRepro[j] / 2.0;
	if (w[j] < 0) w[j] = 0;
}

//Normalización
float wtot = w[0] + w[1] + w[2];

for (int j = 0; j < 3; j++) {
	w[j] /= wtot;
}
```

Una vez hecho esto ya estamos en disposición de aplicar el promediado tanto de orientación, como de pose. Ambos empleando estas ponderaciones.

Por último, debemos pasar la pose media a coordenadas globales (de la mesa) y extraer la posición y orientación de la matriz homogénea.

Código 17. Posición y orientación media PnP

```
poseMedia = promed_quat(mRotcameraC1, mTranscameraC1, mRotVecCamera, w);

poseGlobal = wTc1 * poseMedia;

//Obtenemos posición
poseGlobal = wTc1 * poseMedia;
int posX = round(poseGlobal(0, 3) * 100);
int posY = round(poseGlobal(1, 3) * 100);

//Orientacion
Eigen::MatrixXd ejeY_local(4, 1);
ejeY_local << 0.0, 1.0, 0.0, 1.0;
Eigen::MatrixXd ejeY(4, 1);
ejeY = poseGlobal * ejeY_local;

float orientacion = atan2(ejeY(1), ejeY(0));
```

5.3.2 Triangulación

La triangulación intenta buscar un punto en el espacio dadas dos proyecciones 2D del mismo y las transformaciones entre las dos proyecciones. Es decir, intentar encontrar el punto de intersección entre los rayos de proyección de dos cámaras.

En teoría este problema es trivial, pero en práctica entran en juego imperfecciones, que hacen que no exista un punto 3D exacto en el que se corten ambos haces de proyección. Por tanto, el problema ahora pasar a ser el de buscar el punto que más cercano a ambos rayos de proyección esté.

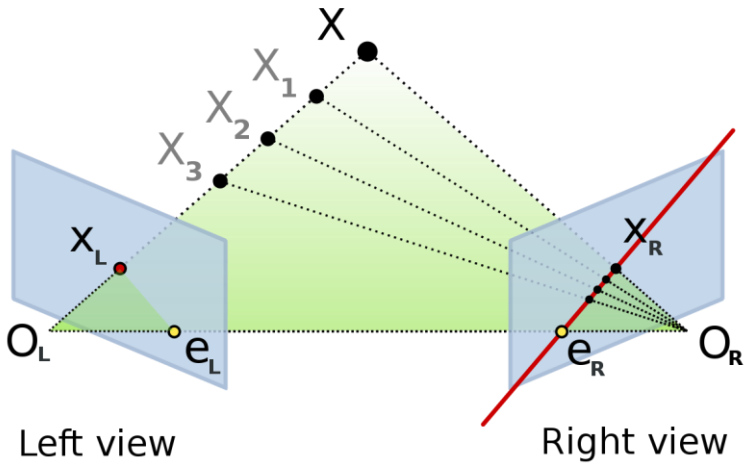

Figura 5.17 *Geometría Epipolar. Visión Estéreo.. Freepng.es*

En nuestro caso, tenemos tres cámaras, por tanto, tenemos que realizar este proceso dos veces con la ayuda de la función *triangulatePoints* de OpenCV. Esta nos permite dar un conjunto de puntos a triangular juntos con las matrices de proyección de ambas cámaras y nos da como resultado la posición 3D de los puntos.

La matriz de proyección es igual al producto de la matriz extrínseca homóloga (sin la última fila) por la matriz de parámetros intrínsecos. Si queremos referenciar siempre a la cámara 1, la matriz de proyección debe tener la siguiente forma, equivalente a no tener ni translación ni rotación:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} x \boldsymbol{K}$$

Donde $\boldsymbol{K}$ es la matriz de parámetros intrínsecos.

La función triangulatePoints nos devuelve una matriz donde cada columna son las componentes de un punto 3D triangulado. Posteriormente se realiza una media de la posición de todos estos, como forma de mezclar la información obtenida de las dos triangulaciones.

Código 18. Triangulación

```
triangulatePoints(cameraMatrix1* projMatrC1, cameraMatrix2* projMatrC2, imPoints[0],
imPoints[1], points4D_c1c2);
triangulatePoints(cameraMatrix1* projMatrC1, cameraMatrix3* projMatrC3, imPoints[0],
imPoints[2], points4D_c1c3);

cv2eigen(points4D_c1c2, puntosC1C2);
cv2eigen(points4D_c1c3, puntosC1C3);

mediaPuntos = puntosC1C2;

for (int i = 0; i < NBOLAS; i++) {
      for (int j = 0; j < 3; j++) {
            mediaPuntos(j, i) = (puntosC1C2(j, i)/ puntosC1C2(3, i) +
            puntosC1C3(j, i) / puntosC1C3(3, i))/2.0;
      }
}
```

Después debemos pasar todos estos puntos a coordenadas globales, ya que actualmente se encuentran referenciados a la cámara 1.

```
Eigen::MatrixXf puntosGlobal = wTc1 * mediaPuntos;
```

Por último, solo queda extraer la posición y orientación de este conjunto de puntos:

Código 19. Posición y orientación media Triangulación

```
//coordenadyas x,y del segundo punto (verde)
int posX = round(puntosGlobal(0,1) * 100);
int posY = round(puntosGlobal(1,1) * 100);

//Orientacion
//coordenadas y,x de los puntos primero y tercero
float orientacion = atan2(puntosGlobal(1, 0) - puntosGlobal(1, 2),
                          puntosGlobal(0, 0) - puntosGlobal(0, 2));
```

6 ROBOT MÓVIL

Como ya se comentó en la sección de Hardware, el robot empleado es uno cedido por el departamento de un proyecto anterior. Tampoco se tiene más información sobre él, pero se ha podido identificar la electrónica y así como usarla.

En esta sección se explicará todo lo relacionado con la programación y el software embebido en el sistema móvil, así como la comunicación con la estación en tierra.

6.1 Programación de la placa

El microcontrolador Baby Orangutan carece de cualquier tipo de conexión USB como es común tener en las conocidas placas Arduino. Su programación es algo diferente pero también es posible hacerlo con la ayuda del Arduino IDE. Sin embargo, necesitaremos otro elemento hardware que realice la conexión necesaria entre este microcontrolador y el puerto serie de nuestro ordenador.

Gran parte de la información ha sido obtenida gracias al blog de Julio Echeverri [26].

Una vez instaladas las librerías de Pololu en el Arduino IDE [15], como se indicó en la sección de Software, debemos disponer de una placa Arduino para emplearla como programador ISP. En mi caso, disponía de una placa Arduino MEGA, que puede ser perfectamente empleada para esto. Debemos programar esta placa con el ejemplo 11 de la librería del IDE de Arduino, *ArduinoISP*.

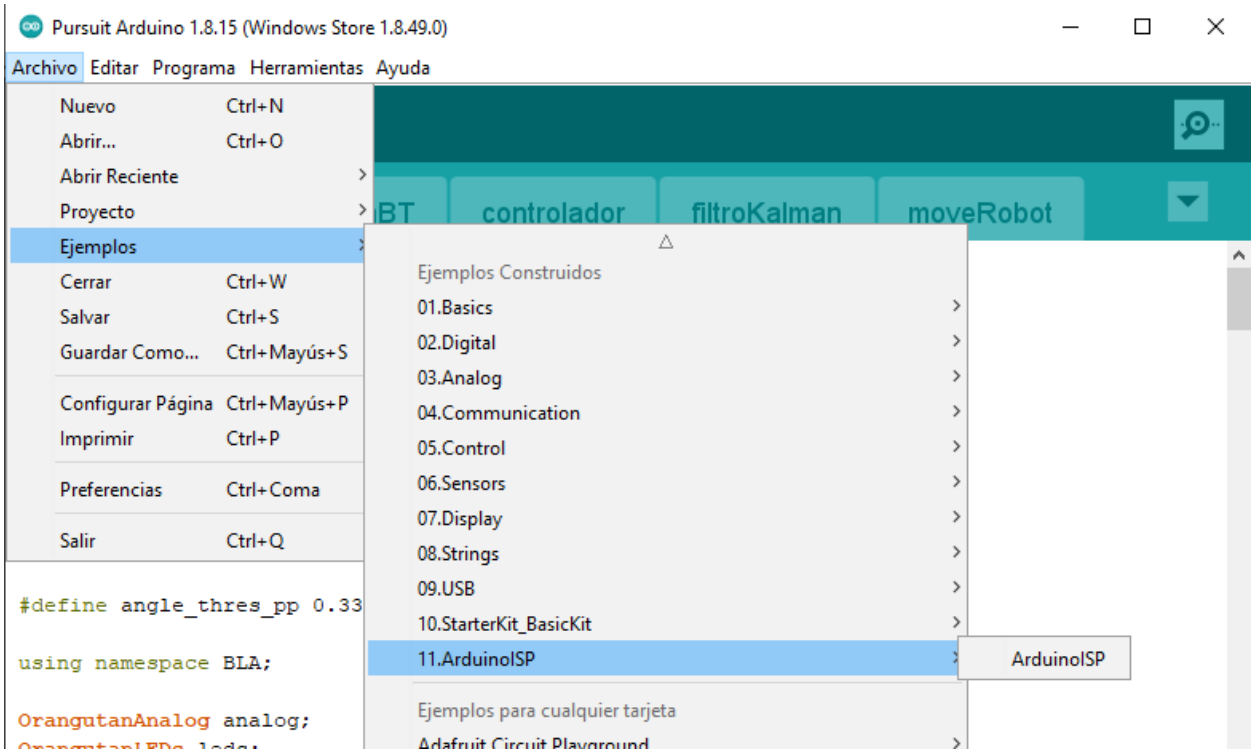

Figura 6.1 *Arduino ISP en la librería de ejemplos*

Una vez cargado este ejemplo en nuestra placa, debemos cambiar en la sección herramientas:

- Placa: Pololu Orangutan or 3pi Robot w/ ATmega328P
- Programador: Arduino as ISP

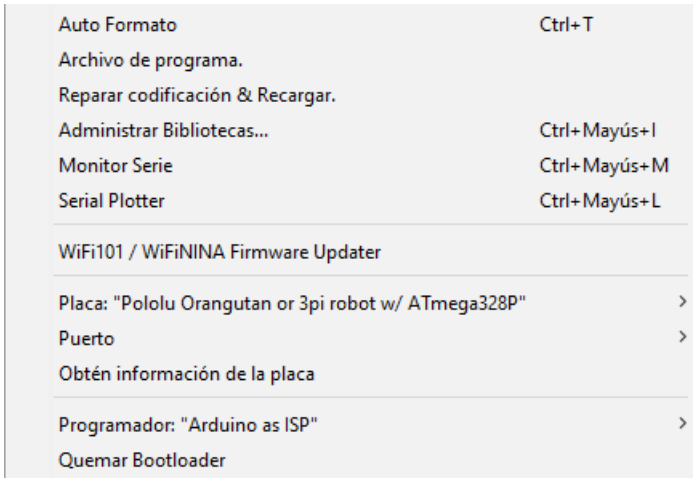

Figura 6.2 *Modificaciones en Herramientas*

Una vez hecho esto ya podemos programar nuestro Baby Orangutan tal y como lo haríamos con un Arduino, tan solo nos falta conectar los pines correspondientes entre la placa de Pololu y la de Arduino. Para ello hacemos uso de la interfaz ISP de ambos micros.

Tabla 6.1 Pinout Arduino MEGA

Nombre PIN ISP	Número Pin
MISO	50
MOSI	51
SCK	52
RST	10

En el Baby Orangutan, podemos guiarnos por este esquema, siendo la fila 1 la más externa a la placa:

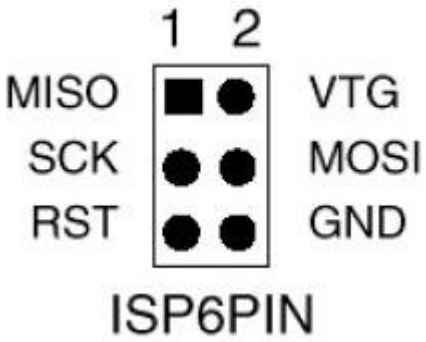

Figura 6.3 *ISP Pinout en BabyOrangutan*

Ahora simplemente debemos conectar el Arduino Mega a nuestro ordenador por cable, tal y como estamos acostumbrados a hacerlo y apretar el botón de programar en el Arduino IDE.

6.2 Comunicación

6.2.1 Base de tierra

Hemos de añadir alguna forma de enviar los datos de posición y orientación calculados en la sección anterior al Arduino vía Bluetooth. Como ya se comentó en la sección de Hardware, también se ha realizado con la ayuda de un Arduino Pro Mini un adaptador Bluetooth a USB para poder enviar datos por este canal desde el PC del laboratorio.

A efectos prácticos una conexión Bluetooth y una conexión USB se interpretan igual desde el sistema operativo, empleando puertos COM. Por tanto, no es necesario hacer distinción para el caso de un PC que si disponga de Bluetooth integrado.

Se ha decidido emplear la librería *ArduSerial* [27] disponible en GitHub para su libre uso. Esta tiene la ventaja de que su programación es prácticamente idéntica al del puerto serie en Arduino. Solo debemos abrirlo y esperar recibir bytes y leerlos o enviar. Para ello debemos descargarnos la librería y añadirla a nuestro proyecto de Visual Studio como cualquier otra librería.

De la comunicación continua se encarga una función de comunicación (*testBT*). Esta crea un hilo para atender las órdenes por teclado (función *teclado*), en caso de querer un control manual, y otro para atender a la información que llega por Bluetooth, ya que la dirección es bidireccional, y también recopilamos información de vuelta del robot.

Código 20. Comunicación de base a robot. Función principal.

```
void teclado(string* texto) {
	Sleep(2000);
	while (1) {
		cin.clear();
		cin >> *texto;
		Sleep(1000);
	}
}

void testBT(string* dataArd) {
	ofstream myfile;
	myfile.open("output/arduino.txt", ios::out | ios::trunc);
	Serial3.begin(9600);
	std::string data = "";
	std::thread hiloTeclado(teclado, &data);
	std::cout << "Starting..." << std::endl;
	while (!Serial3) Sleep(1000);
	while (Serial3.available())
		Serial3.read();
	std::cout << "Connected" << std::endl;
	while (1) {

		if (Serial3.available()) {
			while (Serial3.available()) {
				char c = Serial3.read();
				printf("%c", c);
				myfile.open("output/arduino.txt", ios::out | ios::app);
				myfile << c;
				myfile.close();
```

```
                }
            }
            else {
                if (*dataArd != "") {
                    Serial3.println(*dataArd);
                    *dataArd = "";
                    Sleep(5);
                }
                if (data != "") {
                    Serial3.println(data);
                    data = "";
                    Sleep(5);
                }
            }
        }
}
```

El hilo se crea con *std::thread* [28] y tiene un uso bastante parecido al visto en asignaturas de la carrera como *Informática Industrial*. Al crearlo solo debemos pasarle una función y los parámetros de la misma, en este caso llamamos a la función *teclado* y le pasamos una cadena de caracteres sobre la que queremos que escriba lo introducido por el usuario.

Solo debemos tener en cuenta, que hay que cambiar el puerto COM que se use, esto depende el ordenador y del puerto empleado. Solo hay que cambiar el número después de cada comando *SerialX*. En nuestro caso tenemos Serial3 ya que es el puerto COM utilizado. Para más información ir a [27].

Algo equivalente se hace en el bucle principal, esta función *testBT* también se carga a un hilo para poder tener paralelismo entre la comunicación Bluetooth y la ejecución de los algoritmos de post procesado.

```
string data_arduino;
data_arduino = "";
std::thread hiloComunicacion(testBT, &data_arduino);
```

Luego, una vez ya calculadas la posición y orientación estimadas solo debemos modificar una cadena de texto y que el hilo principal se encargue de enviarlo.

```
sprintf_s(aux, "x%03d", posX);
data_arduino = aux;

Sleep(60);
sprintf_s(aux, "y%03d", posY);
data_arduino = aux;

Sleep(60);
sprintf_s(aux, "z%03d", int(orientacion * 100));
data_arduino = aux;
```

Aquí vemos que se envía la posición y orientación con la siguiente codificación (3 cifras de precisión):

- Posición X: x###
- Posición Y: y###
- Orientación: z###

6.2.2 Robot móvil

La recepción de información procedente de la base en tierra se realiza en una sola función de comunicación, una vez por iteración de bucle principal del microcontrolador (*loop*). Se conforma una estructura de comandos en la que la primera letra indica un comando y posteriormente pueden venir un número indeterminado de bytes acompañando al comando que sirven como parámetro para el mismo.

Código 21. Recepción de datos en microcontrolador

```
void BTread(func_modes* modoFuncionamiento, int* velocidad, float* rx, float* ry,
float* x, float* y, int* ls, int* as, float* angle)
{
  int SerialIn = 0;
  int dataIn;
  int mod;
  mod = Serial.read();

  switch (mod)  {
    case 'q':
      *modoFuncionamiento = 0;
      break;
    case 'w':
      *modoFuncionamiento = 1;
      break;
    case 's':
      *modoFuncionamiento = 2;
      break;
    case 'a':
      *modoFuncionamiento = 3;
      break;
    case 'd':
      *modoFuncionamiento = 4;
      break;
    case 'o':
      *modoFuncionamiento = 5;
      break;
    case 'p':
      *modoFuncionamiento = 6;
      break;
    // Cambio de velocidad
    case 'v':
      *velocidad = formarNumero(3);
      break;
    // Cambio de posicion X
    case 'x':
      *x = formarNumero(3);
      posXupd = true;
      break;
    // Cambio de posicion Y
    case 'y':
      *y = formarNumero(3);
      posYupd = true;
      break;
    case 'z':
      *angle = float(formarNumero(3) / 100.0);
      thetaupd = true;
      break;
    // Cambio de referencia X
    case 'r':
      *rx = formarNumero(3);
      break;
    // Cambio de referencia Y
    case 't':
      *ry = formarNumero(3);
      break;
```

```
      case 'k':
        *as = float(formarNumero(3) / 10.0);
        break;
      case 'l':
        *ls = float(formarNumero(3) / 10.0);
        break;

      case 'j':
        resetKalman();
        break;
  }
}
```

El modo de funcionamiento indica los diferentes modos manuales y automáticos. Estos se configuran mediante la siguiente enumeración:

```
typedef enum {halt, forw, backw, tleft, tright, controlv1, pursuit} func_modes;
```

Los cuatro primero son modos manuales y simplemente indican el tipo de movimiento que queremos que el robot haga (avanzar, retroceder, girar, etc.). Los dos último son modos automáticos que hacen uso de la información de posición para intentar alcanzar un objetivo.

Para conformar los bytes correspondientes al parámetro en un número, se hace uso de la función *formarNumero.*

Código 22. *formarNumero*

```
int formarNumero(int N) {
  int SerialIn = 0;
  int signo = 1;
  int dataIn;
  while (Serial.available() < N) delay(50);
  while (Serial.available() > 0 && dataIn != 'b') {
    dataIn = int(Serial.read());
    if (dataIn >= 48 && dataIn <= 57) SerialIn = 10 * SerialIn + dataIn - 48;
    else if (dataIn == '-') signo = -1;
  }
  SerialIn = SerialIn * signo;
  return SerialIn;
}
```

En esta, teniendo en cuenta que los números vienen codificados como caracteres uno a uno, debemos de pasar su valor de codificación ASCII a decimal. Simplemente se resta 48, ya que este es el código del número 0 y el resto van en orden. También se ha tenido en cuenta el símbolo "-" para cambiar de signo el parámetro.

El envío de información se realiza sencillamente cada 500ms en el bucle principal:

Código 23. Envío desde robot

```
if((tcAct - tcAnt) >= 500) {
    //Serial.println(String(refx) + "\t" + String(refy)+ "\t" + String(posx) + "\t" +
      String(posy) + "\t" + String(velL)+ "\t" + String(velR) + "\t" + String(phi));

    Serial.println(String(velL)+ " " + String(velR));
    tcAnt = millis();
}
```

Se debe tener en cuenta que, en la parte de tierra, la conexión puede fallar, por ello conviene limitar los datos enviados desde el robot móvil. Esto sucede así con varias librerías Serial probadas, y la que al final se ha empleado es la que mejores resultados ha dado.

6.3 Control

En esta subsección trato ambos modos de control automático desarrollados, siendo el segundo la evolución lógica del primero, y sus limitaciones.

Ambos modos de control son relativamente sencillos, esto se debe a que el rango de velocidades al que podemos someter el robot es muy reducido, ya que opera casi a velocidad mínima constantemente. La baja tasa de refresco de las cámaras hace que tengamos que mover el robot lo más lentamente posible. Si pusiésemos los motores, no a velocidad máxima, sino a una velocidad intermedia, este va demasiado rápido como para que la estimación de posición tenga algún valor. Ya que rápidamente se sale de la zona de trabajo y además las variaciones entre estimaciones son demasiado grandes.

Estas velocidades limitan mucho el tipo de control que se pueda realizar, siendo un *todo-nada* prácticamente la única solución viable.

El bucle de control se ejecuta cada 50ms y, en los modos automáticos hay cierta parte del proceso que se repite. Lo primero es el cálculo de errores de posición y orientación que tenemos.

Código 24. Cálculo errores de posición y orientación

```
//Calculamos el ángulo que debe tomar el robot para ir al punto destino
angRef = atan2(refy - posy, refx - posx);

//El angulo resultante deberá estar entre 0 y 2*pi
while (angRef < 0) angRef += 2.0 * pi;

//Error de ángulo (entre -pi y pi)
eAng = angRef - phi;
if (eAng > pi) eAng -= 2.0 * pi;
else if (eAng < -pi) eAng += 2.0 * pi;

//Error lineal
eLineal = sqrt(pow(refy - posy, 2) + pow(refx - posx, 2)); //error lineal
```

Ambos controles tienen dos fases:

1. Fase de orientación: Si el robot está fuera de unos límites en su orientación con respecto al objetivo (no está mirando hacia él), gira sobre sí mismo para orientarse.
2. Fase de avance: El robot se desplaza para llegar al objetivo. Aquí es donde se distinguen los dos métodos.

El primer método, más sencillo, emplea una velocidad constante de avance hasta el objetivo. Esto se debe al problema ya mencionado de rango de velocidades disponible.

El segundo método, sin embargo, se intenta asemejar a un control *pure-pursuit,* para ello se define la velocidad de avance de forma proporcional al error en distancia, pero imponiendo un mínimo de 5 cm, ya que si no la velocidad resultante haría que los motores no tuviesen par suficiente. A

Código 25. Control estilo *pure-pusuit*.

```
//Control de velocidad linear
lookAhead = -eLineal;
if(lookAhead < -5.0) lookAhead = -5.0;
vCont = controlador1(0, lookAhead, tm) + 10;
```

Adicionalmente se sobrepone a este movimiento lineal una rotación. Esto se hace para que el robot pueda realizar pequeñas curvaturas a la vez que avanza, y poder ir corrigiendo así la deriva en orientación que va a ir sufriendo.

```
      wCont = controlador2(0, -eAng, tm);
      wCont = wCont * radioRueda;

      //Calculo de la velocidad de referencia de cada rueda
      vLCont = vCont - baseline * wCont / 2.0;
      vRCont = vCont + baseline * wCont / 2.0;

      colR = round(vRCont);
      colL = round(vLCont);
```

Para ello también se emplea un controlador de tipo proporcional (P). Posteriormente se traspasan ambas velocidades las velocidades correspondientes de cada rueda. Para ello necesitamos los parámetros radioRueda y baseline. Que debemos medir directamente sobre el robot:

```
#define radioRueda  2.0   //cm
#define baseline    9.0 //cm
```

Estos son los dos únicos modos de control que se han podido desarrollar debido a las limitaciones ya explicadas. Una solución posible podría ser la inclusión de reductoras, esto haría que se pudiese manejar el robot de forma más precisa y controlada a menores velocidades, obteniendo así un mayor rango de movimiento y permitiendo desarrollar controles más complejos.

6.4 Filtro de Kalman

Para poder emplear los datos de posición y orientación enviados desde la estación en tierra, conviene realizar primero un filtrado de estos, ya que la precisión y estabilidad de estas estimaciones no es ideal. Para ello se ha decidido hacer uso de un filtro de Kalman.

Estos filtros son ampliamente conocidos y empleados en el campo de la robótica, ya que no solo te permiten filtrar la información, sino que también permiten la fusión de datos de diferentes sensores para mejorar la estimación. Es muy típico el uso de estos sensores en automóviles, donde las señales de GPS tienen relativa poca exactitud y una tasa de refresco también muy baja, entorno a 2 Hz en muchos casos. La localización GPS en compañía de otros sensores como pueden ser *encoders* o sensores inerciales, permiten obtener una posición mucho más precisa.

El filtro de Kalman basa su estimación en una dinámica dada, es decir, compara los nuevos datos llegados de los sensores con una estimación interna. Posteriormente compensa y actualiza el nuevo estado en función de las incertidumbres de todas las medidas y estimaciones.

En este caso, el filtro de Kalman a implementar se basa en el descrito en [29] por Mohammad Zahaby y Pravesh Gaonjur, pero con alguna modificación para incluir también la orientación. Basándonos en [29] definimos la matriz de estados y la que define el modelo dinámico:

$$X_p = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \end{bmatrix} \tag{6.1}$$

$$A = \begin{bmatrix} 1 & 0 & 0 & dt & 0 \\ 0 & 1 & 0 & 0 & dt \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.2}$$

Matriz de input de control:

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ dt & 0 & 0 \\ 0 & dt & 0 \end{bmatrix} \tag{6.3}$$

Matriz de medidas, para acoplar las medidas de los sensores (*Z*):

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{6.4}$$

Matriz de covarianza del modelo dinámico:

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \left(\frac{\pi}{3}\right)^2 \end{bmatrix} \tag{6.5}$$

Matriz de covarianza del sistema de localización:

$$R = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 100 & 0 \\ 0 & 0 & \left(\frac{\pi}{3}\right)^2 \end{bmatrix} \tag{6.6}$$

Los parámetros de varianza han sido seleccionados de forma experimental, a prueba y error y comprobando que diesen unos resultados parecidos a los obtenidos en el laboratorio.

Las matriz de covarianza del filtro viene dado por S_p , donde S_p se inicializa con el valor de *R*.

El bucle del algoritmo del filtro de Kalman sería:

$$X_{kp+1} = A\ X_p \tag{6.7}$$

$$S_{kp+1} = A\, S_p A^t + BQB^t \tag{6.8}$$

$$K = \frac{S_{kp+1} H^t}{H S_{kp+1} H^t + R} \tag{6.9}$$

$$X_{p+1} = X_{kp+1} + K(Z - HX_{kp+1}) \tag{6.10}$$

$$S_{p+1} = (I - KH) S_{kp+1} \tag{6.11}$$

Hay un paso extra que no es estándar en los filtros de Kalman, la sustitución de velocidad. En [29] realizan esta sustitución como forma de incrementar la precisión del sistema. En nuestro caso estimaremos la velocidad gracias a la señal de control mandada a los motores. Para ello simplemente debemos modificar las dos últimas componentes del vector de estado *X*.

Si aplicamos este algoritmo a Matlab (en este caso estimamos velocidad constante), podemos simular el comportamiento del sistema en cuanto al problema del tracking.

Código 26. Filtro de Kalman en Matlab

```
%- Inicializamos las variables
%r y q son son parametros a tocar
q = 1; %Varianza del ruido blanco del sistema
qd = pi/3;
r = 10; %Varianza del ruido blanco del gps
rd = pi/3; %Varianza medida angulo
% Le pongo mas ruido al GPS que a la dinamica

%1)Trayectoria coche
step = 1;
alfa = [60:step:120-step];
alfa = alfa/180*pi;
N=size(alfa,2);

R = 120;
xr = R*cos(alfa) - R*cos(alfa(1));
yr = R*sin(alfa) - R*sin(alfa(1));
pos = [xr;yr];
dir = alfa+pi/2;

t=[0:0.5:N/2-0.5];

%2)Modelo lineal del problema
d = 0.5 % segundos
A = eye(5,5);
A(1,4) = d;
A(2,5) = d;

B = zeros(5,3);
B(4,1) = d;
B(5,2) = d;
H = eye(3,3);
H = [H zeros(3,2)];

Q = q^2*eye(3,3); %Matriz de covarianza de la incertidumbre del modelo dinamico
Q(3,3) = qd;

R = r^2*eye(3,3); %Matriz de covarianza del gps
R(3,3) = rd;
%3)
sigma = r^2*eye(5,5);
sigma(3,3) = rd^2;

nu = zeros(5,N);
L = zeros(2,N);
or_gps = dir;
%4) Implementacion del filtro
for i = 1:(N-1)
    z_(:,i) = pos(:,i) + r*randn(2,1);
    gps(:,i) = z_(:,i);
    or_gps(i) = dir(i) + rd*randn;
    z(:,i) = [z_(:,i);or_gps(i)];
    %Prediccion
    nu_ = A*nu(:,i);
    sigma_ = A*sigma*A' + B*Q*B';
    K = sigma_*H'*(H*sigma_*H' + R)^(-1);
```

```
    %Actualizacion
    nun = nu_ + K*(z(:,i) - H*nu_);
    sigman = (eye(5,5) - K*H)*sigma_;
    nu(:,i+1) = nun;
    sigma = sigman;

    %Velocity replacement
    % Estimación de velocidad -> Señal control robot
    % Ángulo filtrado por Kalman
    vel = 1.7;
    nu(4,i+1) = vel*cos(nu(3,i+1));
    nu(5,i+1) = vel*sin(nu(3,i+1));
end

figure(7); hold on; grid;
plot(pos(1,:),pos(2,:),'*r',gps(1,:),gps(2,:),'^b')
plot(nu(1,:),nu(2,:),'og')
legend('Real','GPS','Est');
hold off;
grid on;
axis equal;
title('Posición')

figure(8);
plot(t,nu(4,:),'-o',t,nu(5,:),'-o',t,vecnorm([nu(4,:);nu(5,:)]),'-o');
legend('vx','vy','v')
grid on;
title('Velocidades')

figure(9);
plot(t,dir(:),'*r',t,or_gps(:),'^b',t,nu(3,:),'og');
legend('thetaReal','thetaGPS','thetaEst');
grid on;
title('Orientación [rad]')
```

Así podemos simular de antemano el comportamiento en trayectorias predefinidas, por ejemplo, un arco de circunferencia (GPS hace referencia al método de localización):

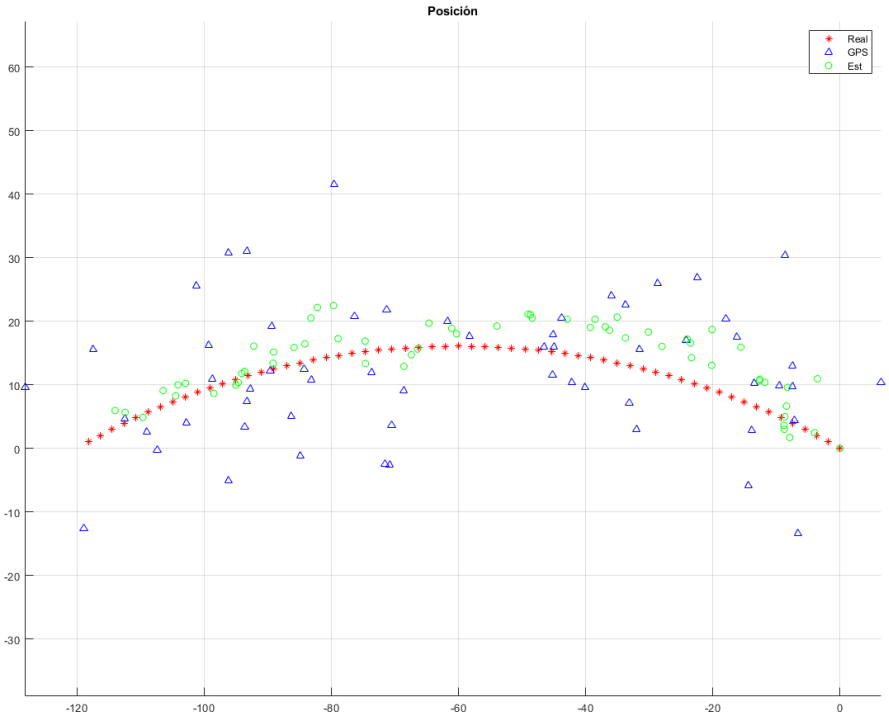

Figura 6.4 *Simulación KF en Matlab. Posición*

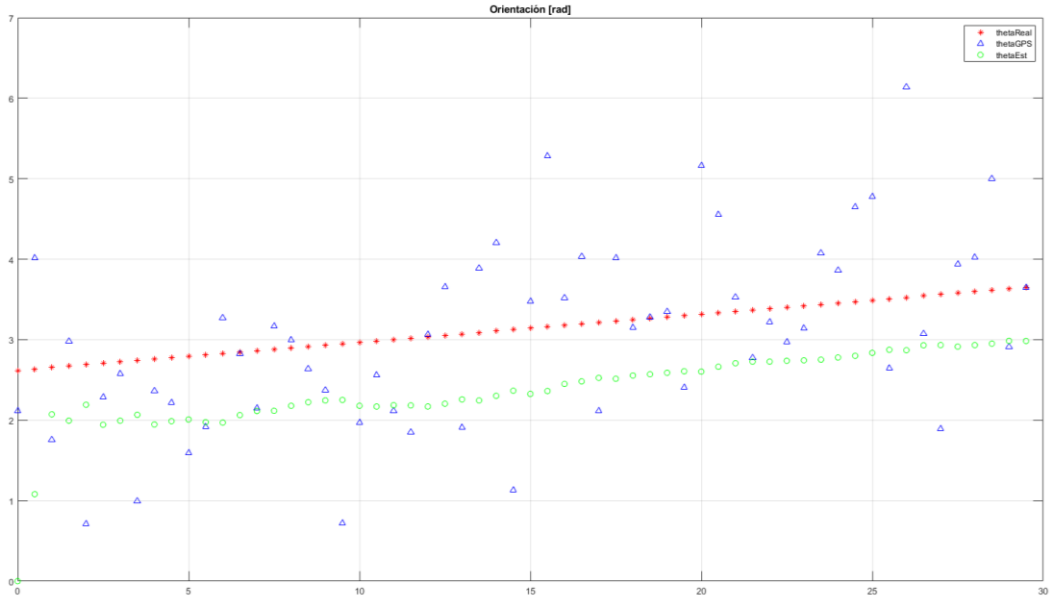

Figura 6.5 *Simulación KF en Matlab. Orientación*

Vemos que, sobre todo en el caso de la posición, el filtro de Kalman consigue mejorar considerablemente la precisión y estabilidad de las medidas. Su implementación real en Arduino hace uso de la librería *Basic Linear Algebra* [30], que nos facilita el uso de matrices. El bucle principal se ejecuta cada vez que se reciben nuevas medidas de posición:

Código 27. Bucle principal filtro de Kalman. Arduino.

```
void filtrarKalman() {
  Matrix<5, 5>predSigma;
  Matrix<5, 1>predX;
  Matrix<5, 5>A_t = ~A;
  Matrix<3, 5>B_t = ~B;
  Matrix<5, 3>H_t = ~H;
  Matrix<5, 3>K;
  Matrix<3, 3> inversa;

  //Prediccion
  predX     =   A * X;
  predSigma =   A * sigma * A_t + B * Q * B_t;

  //Calculo ganancia Kalman
  inversa   =   H * predSigma * H_t + R;
  Invert(inversa);
  K         =   predSigma * H_t * inversa;

  //Actualizacion
  sigma     =    (eye5 - K * H) * predSigma;
  X         =     predX + K * (Z - H * predX);

  //Sustitucion velocidad
  //estVel = estimateVel();
  X(3) = estVel * cos(X(2));
  X(4) = estVel * sin(X(2));
}
float estimateVel(){
  return float((velL + velR)/2.0/10.0);
 }
```

También ha sido necesario el uso de un par de funciones auxiliares para facilitar la programación, una de actualización de matrices con *dt* transcurrido entre iteraciones y otra de reset del filtro, para poder comenzar desde condiciones iniciales.

Código 28. updateMatrices y resetKalman

```
void updateMatrices() {
  A <<  1.0, 0.0,  0.0,  dt,  0.0,
  0.0, 1.0,  0.0,  0.0,  dt,
  0.0, 0.0,  1.0,  0.0,  0.0,
  0.0, 0.0,  0.0,  1.0,  0.0,
  0.0, 0.0,  0.0,  0.0,  1.0;

  B.Fill(0.0);
  B(3, 0) = dt;
  B(4, 1) = dt;

  H.Fill(0.0);
  H(0, 0) = 1.0;
  H(1, 1) = 1.0;
  H(2, 2) = 1.0;

  Q.Fill(0.0);
  Q(0, 0) = q;
  Q(1, 1) = q;
  Q(2, 2) = qd;

  R.Fill(0.0);
  R(0, 0) = r;
  R(1, 1) = r;
  R(2, 2) = rd;

}

void resetKalman() {
  X.Fill(0.0);
  sigma.Fill(0.0);
  sigma(0, 0) = 1.0;
  sigma(1, 1) = 1.0;
  sigma(2, 2) = 1.0;
}
```

Una vez realizado esto, ya estamos en disposición de probar experimentalmente el robot en el laboratorio.

7 EXPERIMENTOS Y CONCLUSIONES

En esta sección se revisan diferentes experimentos realizados en el laboratorio con el fin de evaluar las capacidades del sistema y comparar los diferentes métodos empleados. La idea es extraer unas conclusiones a partir de los datos experimentales.

7.1 Filtro de Kalman

Si extraemos la posición recibida por el robot, podemos comparar la trayectoria realizada en caso de aplicación del filtrado de Kalman y en caso de no aplicarlo. La idea es obtener una gráfica parecida a la figura 6.4 pero con datos reales.

Comparando una trayectoria medianamente compleja, en la que el robot gira y avanza podemos obtener los siguientes resultados:

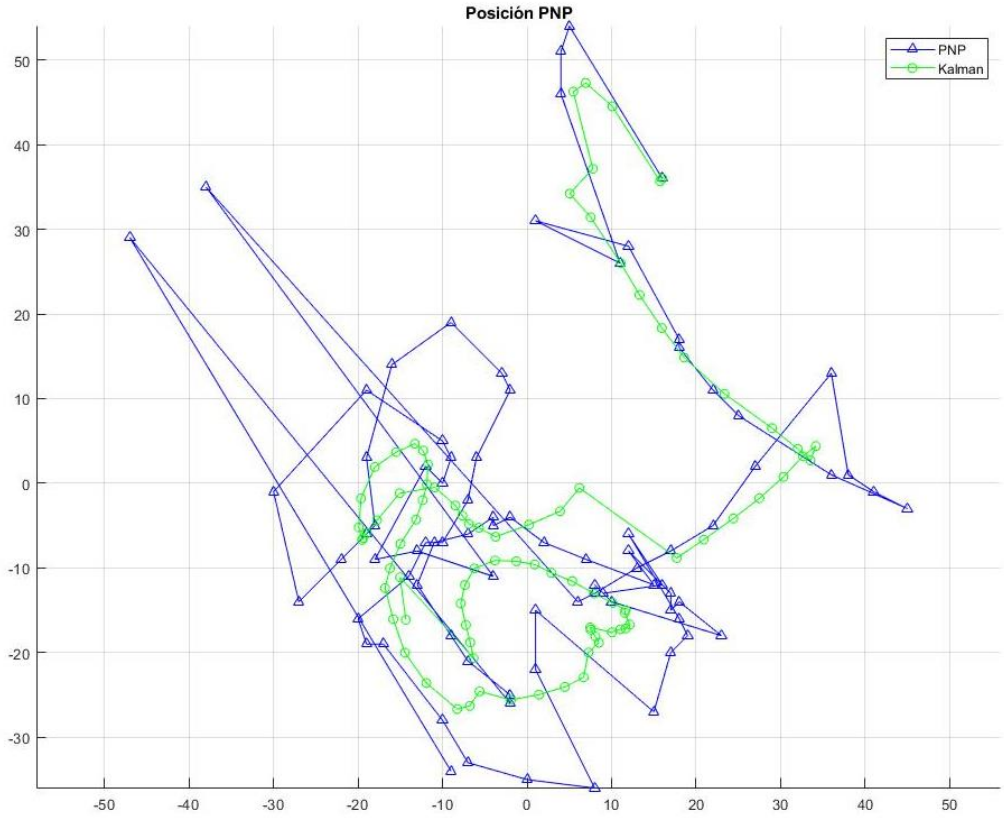

Figura 7.1 *Filtrado vs no Filtrado*

En este ejemplo en concreto se compara la posición dada por el PnP antes y después del filtrado. Vemos que la posición filtrada es mucho menos errática y se aproxima bastante más a la trayectoria real realizada por el robot. De hecho, el uso de la posición sin filtrar para el control del vehículo en modo automático es casi imposible, ya que el robot no alcanza nunca una estabilidad debido a estos errores y la baja tasa de refresco. Por esta razón, el uso del filtro de Kalman es imperativo en esta aplicación.

En esta gráfica no se ha realizado control automático, simplemente seguimiento para analizar los resultados.

7.2 Número de marcadores

Siguiendo el orden de desarrollo del proyecto, la siguiente comparación a tratar es la geometría de los marcadores. Como ya hemos comentado anteriormente en esta memoria de proyecto, disponemos de dos geometrías, una de cuatro marcadores y otra de tres marcadores. Cuya única diferencia a nivel de algorítmica es el uso del método P3P en esta última en lugar del PnP, ya que el problema clásico PnP necesita un mínimo de 4 puntos [21].

Aunque ya se mencionó que el de tres marcadores parecía tener un comportamiento menos errático, ya que es menos probable que falle al tener menos componentes, basado en pequeños experimentos elaborados a medida que se desarrollaba el proyecto, conviene realizar experimentos en condiciones similares para ver cual de las dos arroja mejores resultados.

Primeros veamos la precisión en la tarea de determinación de posición para un experimento en línea recta y bastante controlado para obtener unas condiciones idóneas:

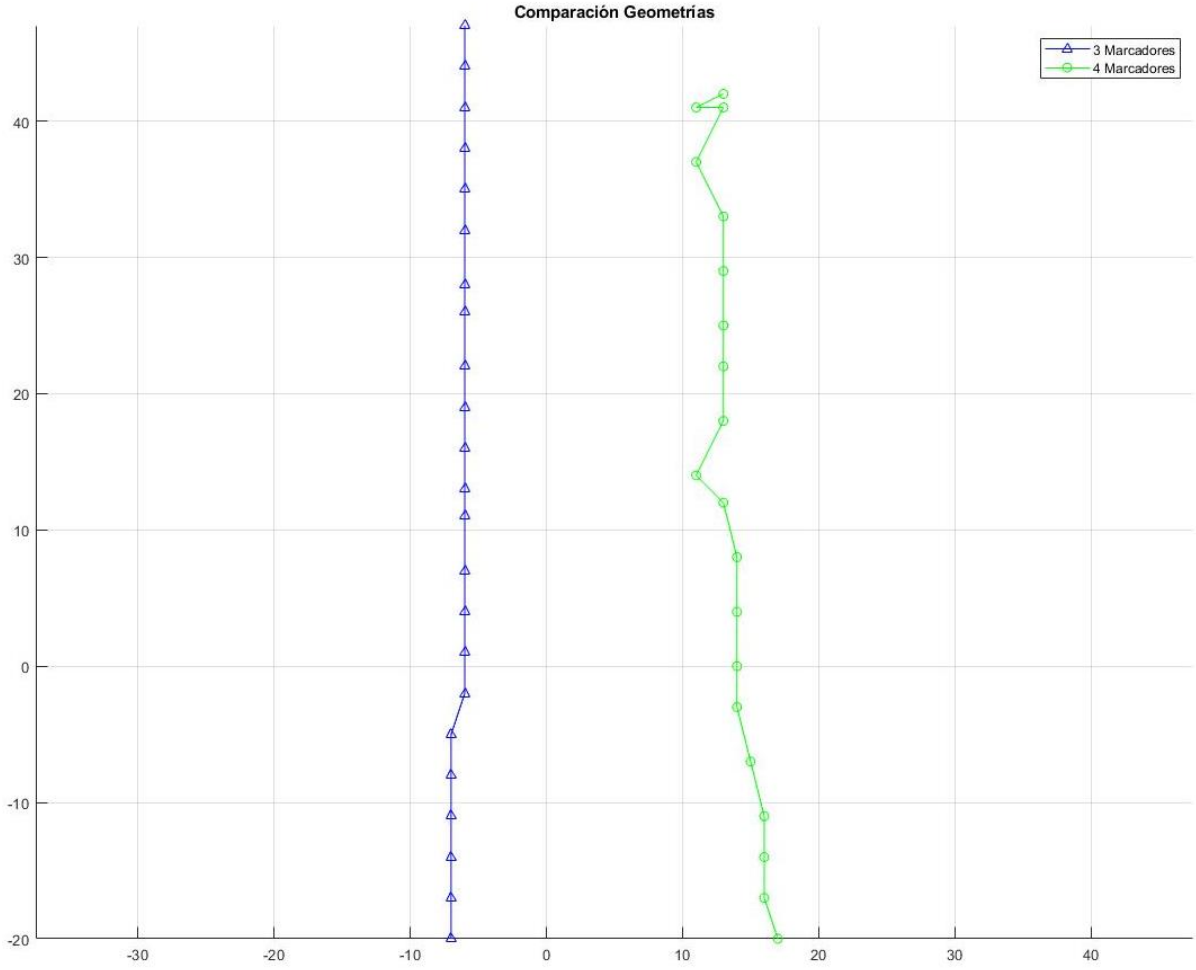

Figura 7.2 *Seguimiento en línea recta. Comparación de geometrías*

En principio obtenemos resultados bastante parecidos con ambas geometrías, siendo la de tres marcadores algo más estable. Hay que tener en cuenta que todo esto es después de todos los filtros implementados para mejorar la estabilidad del sistema y que se ha empleado la triangulación como método para la estimación.

Aunque el de tres marcadores arroja mejores resultados, el de cuatro tampoco tiene un funcionamiento del todo erróneo. Otro aspecto por comparar sería la orientación que obtenemos.

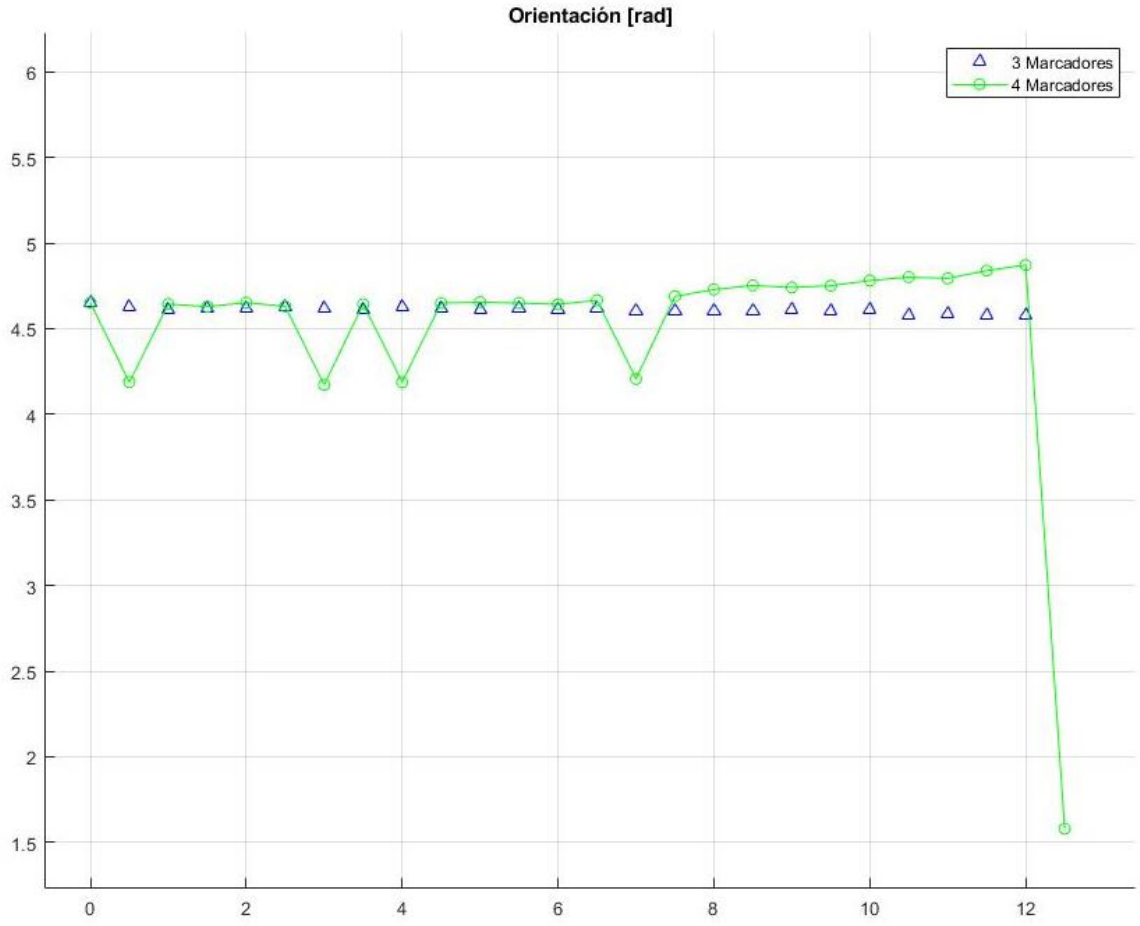

Figura 7.3 *Orientaciones. Comparación de geometrías*

Las oscilaciones en orientación si que son mucho más dañinas para el control, sobre todo a medida que el robot se acerca al objetivo. En este caso vemos que la geometría de cuatro marcadores tiene bastante más inestabilidad que la de tres, alrededor de medio radián de diferencia entre una medida y otra es algo común en el experimento.

Cabe destacar que en el laboratorio y probando con posiciones más desfavorable para las cámaras y la reflexión de infrarrojos de los marcadores, la geometría de cuatro bolas tenía un rendimiento considerablemente más pobre, obteniendo así muchos *frames* en los que la orientación era errónea o directamente no se podía realizar una estimación debido a que no se detectaban todos los marcadores correctamente.

Por estas causas se llega a la conclusión de que, con el fin de dotar a la plataforma de mayor estabilidad, es recomendable emplear la geometría de tres marcadores propuesta, o idea otra en la que todos los marcadores sean absolutamente inequívocos en la identificación.

7.3 Control PnP vs Triangulación

En estos experimentos trato de identificar cuál de los dos métodos de estimación de posición es mejor para emplear en el control del robot móvil. Para ello se emplearán en ambos casos la geometría de tres marcadores y se le ordenará al robot que vaya a una posición determinada, partiendo también de la misma posición (aproximadamente).

Primero de todo veamos qué datos arrojan ambos métodos para un mismo movimiento aleatorio en bucle abierto:

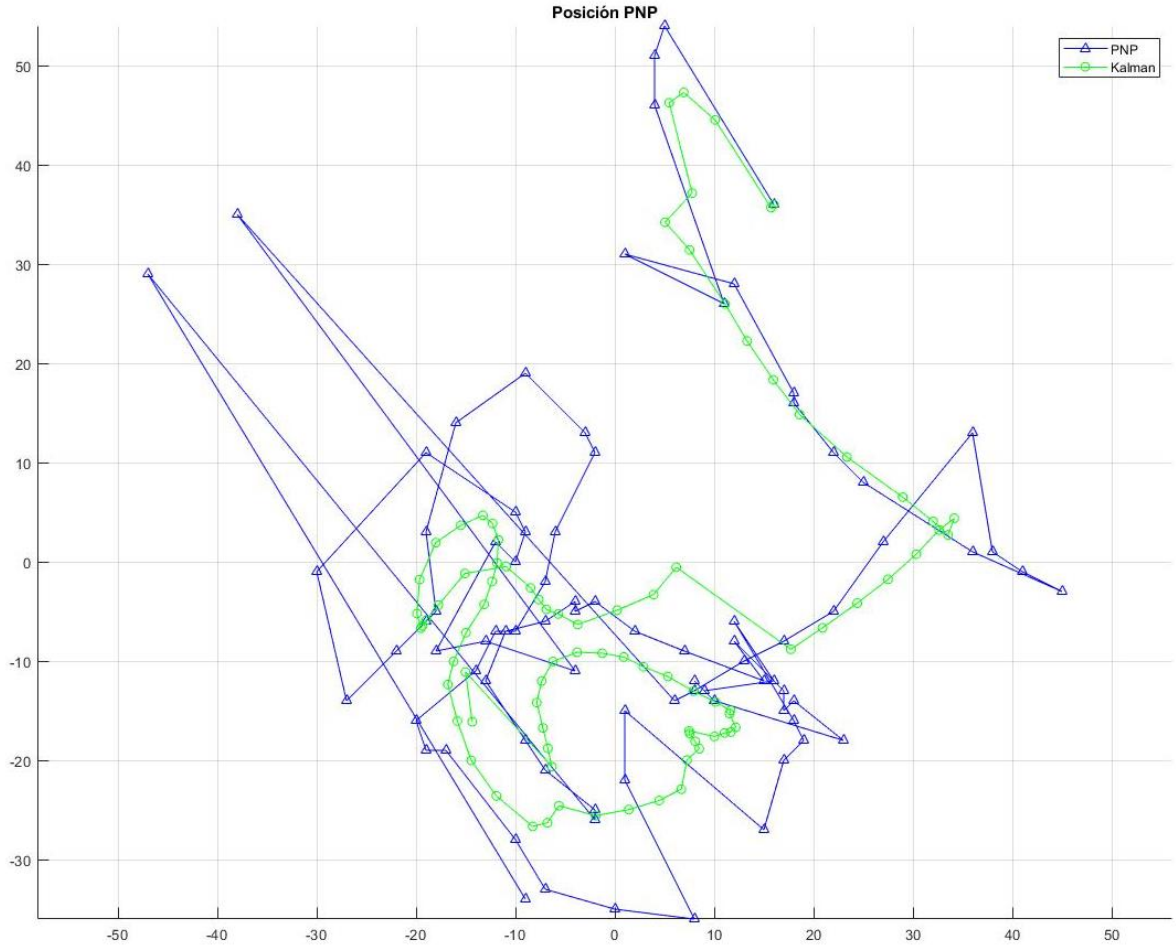

Figura 7.4 *Tracking con PnP (misma que 7.1)*

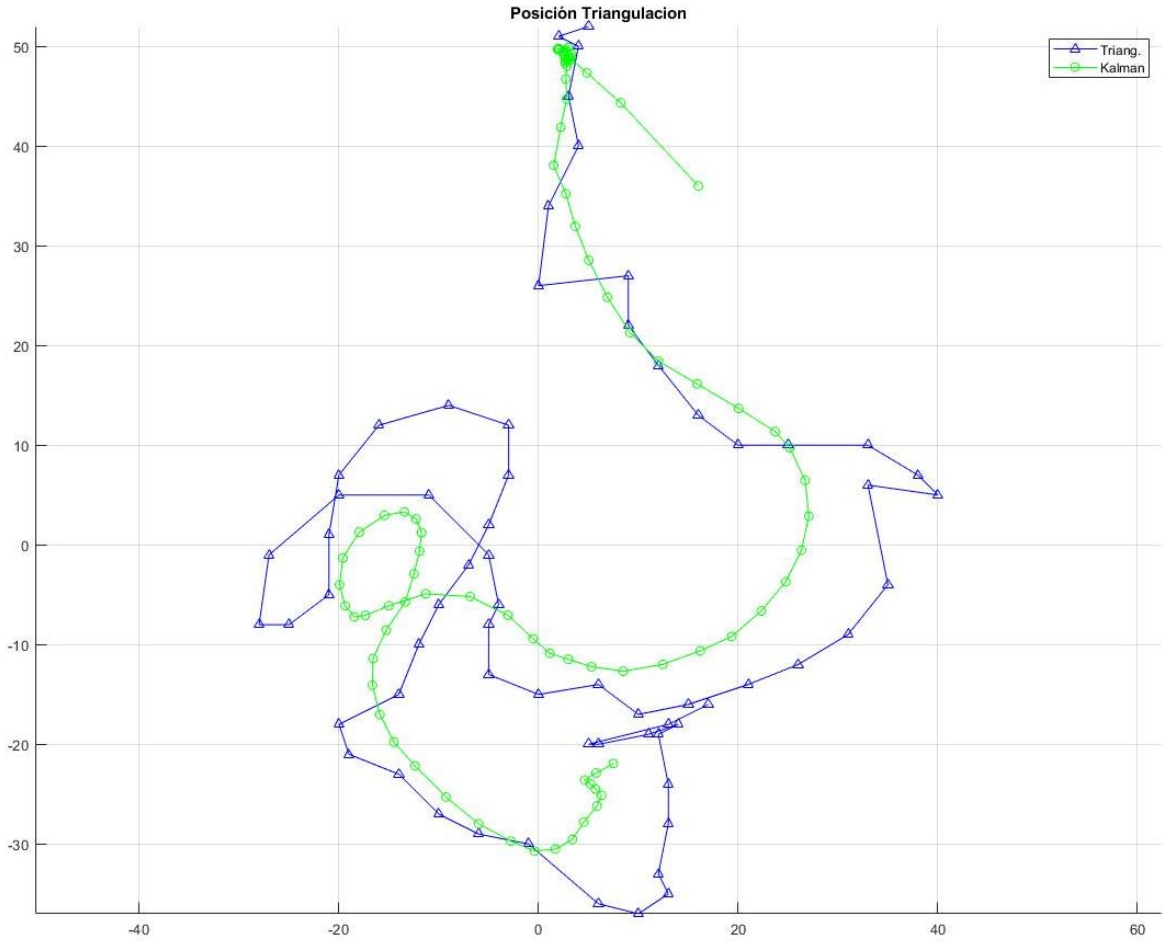

Figura 7.5 *Tracking con Triangulación*

A vista de las gráficas anteriores, podemos esperar que la triangulación sea bastante más estable que el PnP y por tanto resulte en un mejor control del robot móvil. Las posiciones obtenidas por este método siguen una trayectoria más "suave" y por tanto el control no tiene que hacer frente a tantas perturbaciones. Vemos sin embargo que, en ambos casos, el filtro de Kalman realiza una buena labor suavizando aún más estas medidas.

Pasemos ahora a ver el funcionamiento en bucle cerrado. En el primer experimento de control, el punto objetivo se encuentra prácticamente en frente del robot (0,-15), por tanto, este solo debería avanzar y girar levemente, no debería realizar giros muy abruptos.

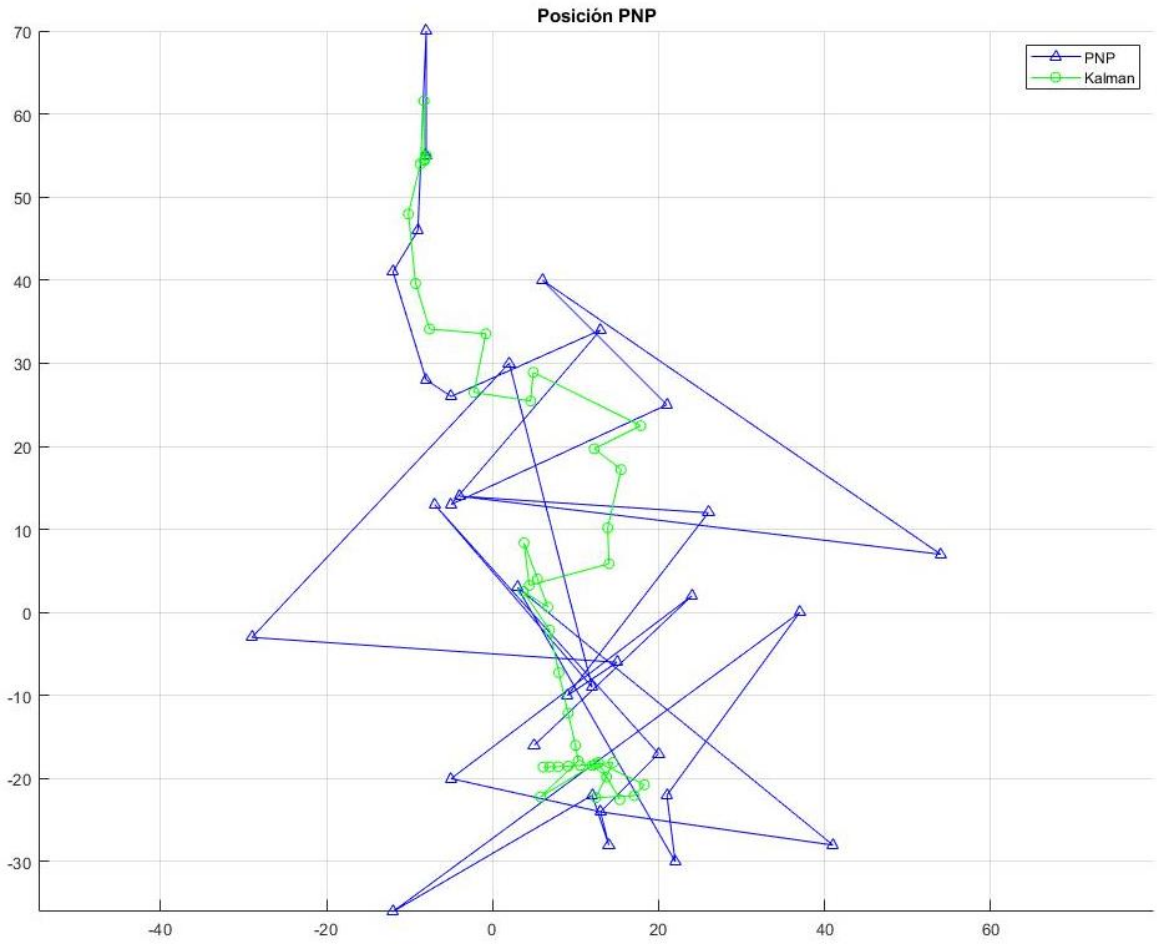

Figura 7.6 *Experimento 1. PnP*

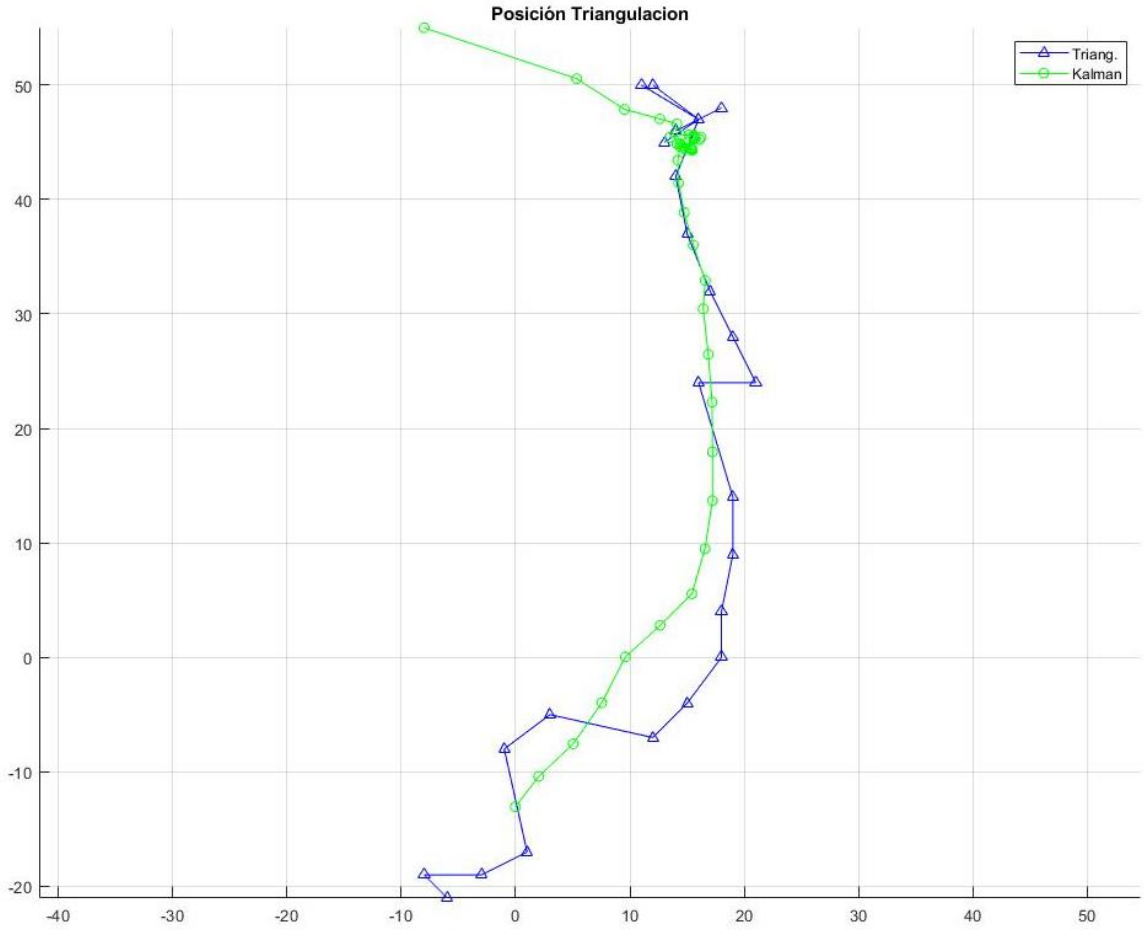

***Figura 7.7** Experimento 1. Triangulación*

Los primeros puntos estimados (arriba) son los del inicio del algoritmo, este debe primero converger para obtener una posición más fiable, por tanto, en esos puntos el robot se encuentra quieto. Posteriormente se produce el movimiento. Podemos ver una gran diferencia entre ambos movimientos realizados. La estimación de PnP es mucho más caótica que la de triangulación, eso se traduce en que el movimiento es más errático, realizando giros en direcciones erróneas.

La orientación es uno de los principales problemas del trabajo en general. Como ya se comentó, debido a la baja tasa de refresco, es necesario que el robot gire muy despacio, siendo prácticamente imposible de conseguir con los motores disponibles. El efecto de esto lo vemos claramente en el PnP, que empieza a realizar eses en cuanto se encuentra con el primer error en orientación. La triangulación, sin embargo, parece solventarlo con un poco más de precisión y estabilidad.

Aun así, conviene comprobar el comportamiento en trayectorias más abruptas, es decir, con giros más destacados. En el siguiente experimento, el punto objetivo es el mismo, pero el robot empieza con una orientación que dista unos 90º de la de avance. Por tanto, el robot primero debe orientarse y luego avanzar:

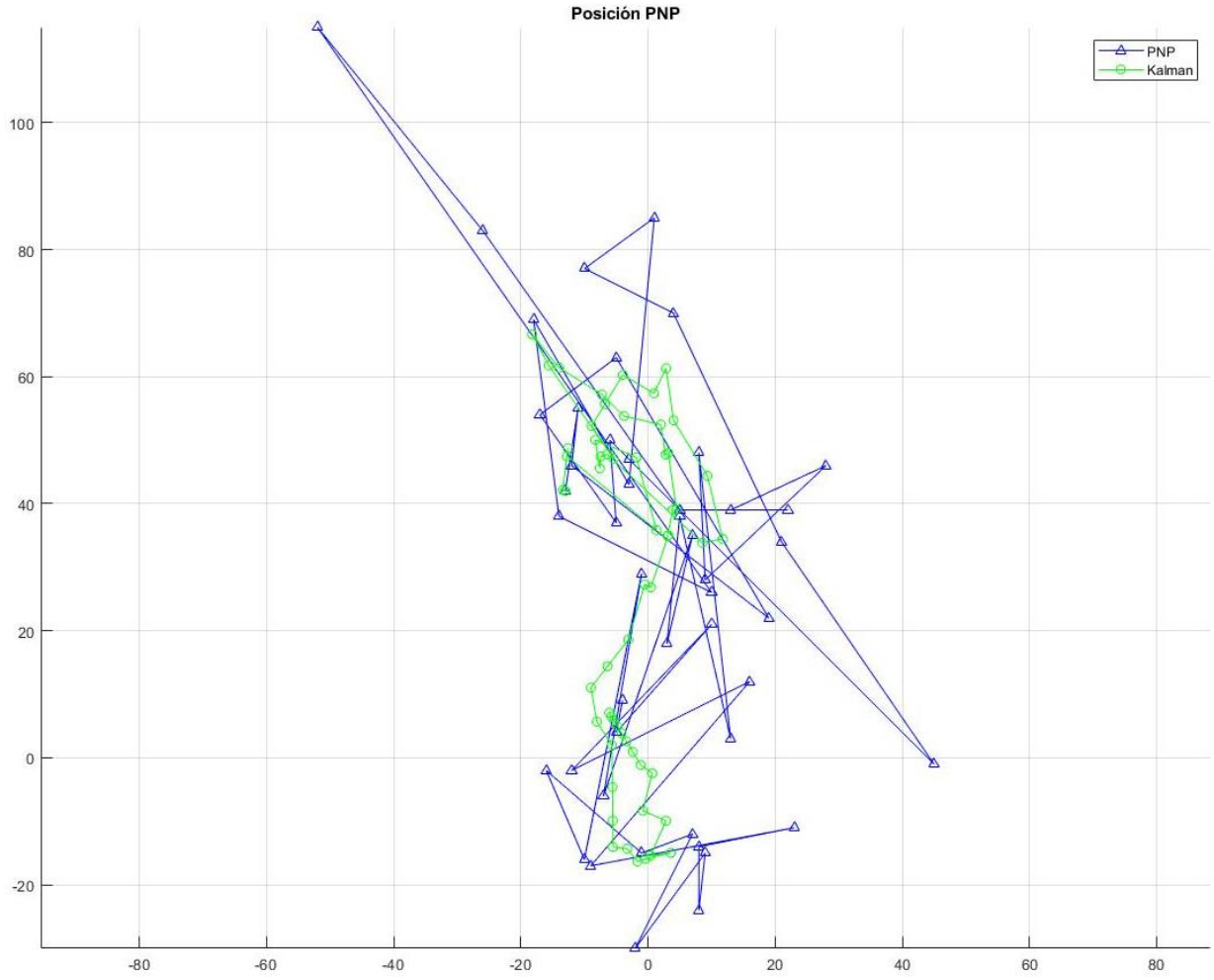

Figura 7.8 *Experimento 2. PnP*

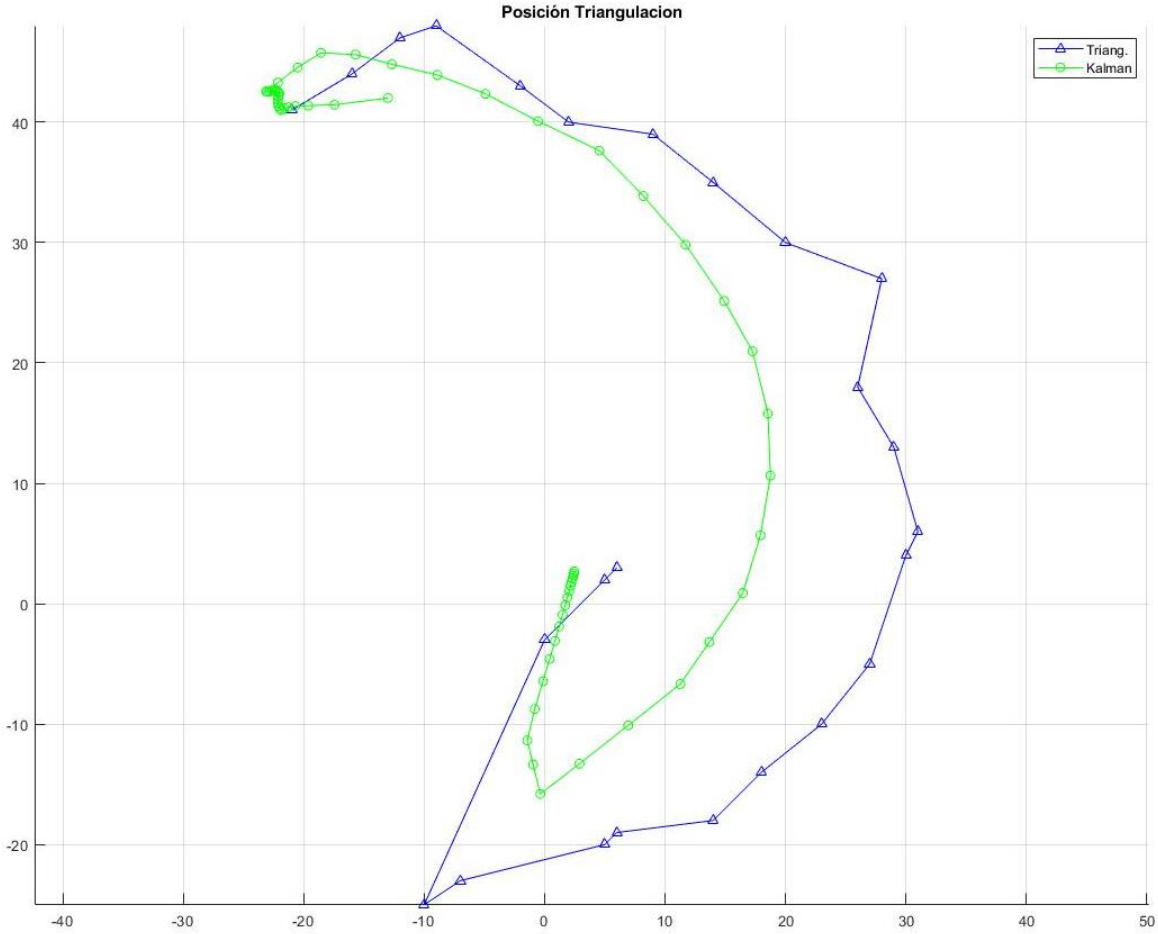

Figura 7.9 *Experimento 2. Triangulación*

De nuevo se comprueba el funcionamiento errático del PnP. El nuevo objetivo, al precisar de una rotación del robot, hace que las estimaciones varíen mucho, ya que el PnP es muy sensible a las variaciones de los centros de los marcadores y al girar, la reflexión sobre los mismo es inestable. En este experimento el robot se acerca mucho al objetivo, pero no consigue estabilizarse.

Por el contrario, la triangulación sí que lo consigue. Aunque dista mucho de realizar una trayectoria óptima (el control tampoco lo es debido a las limitaciones), sí que realiza algunas curvas lógicas y con una estabilidad mucho mayor que con el otro método.

7.4 Conclusiones

Una vez revisados los experimentos, se puede concluir que, a vista de los resultados obtenidos en esta serie de experimentos y el uso que se le ha dado durante el desarrollo, la combinación para obtener los mejores resultados en cuanto a precisión y, sobre todo, estabilidad es emplear la geometría de tres marcadores y la triangulación. Este método es además mucho más sencillo de implementar, pudiendo prescindir del promediado de cuaternios y postprocesado de información del PnP. Al ser más liviano, también puede que permita utilizarse de forma más rápida y optimizada, ideal para la plataforma ya que se deben intentar alcanzar el mayor número de imágenes por segundo posibles.

Hay problemas que se han obviado o pasado por alto en el trabajo. Por ejemplo, la sincronización entre las cámaras, a simple vista no parece haber mucho retardo entre le toma de imágenes de una cámara y de otra, pero es algo a tener en cuenta para posibles ampliaciones y mejores de la plataforma. El uso de unos marcadores más homogéneos quizás también sea conveniente, pero los focos a disposición en el laboratorio no iluminan de igual forma toda la zona de trabajo, por tanto, también aconsejo el uso de LEDs infrarrojos o similar. Esto es algo que se probó en el laboratorio y eran captados por las cámaras casi como circunferencias perfectas, lo cual ayuda a su segmentación, pero debido a la falta de medios era demasiado complicado identificar las geometrías y no se pudo continuar el desarrollo.

El hardware del robot móvil ha respondido bastante bien, el tamaño y durabilidad del mismo es bastante bueno, sin embargo, debido a la baja tasa de refresco, se hace necesario emplear algún tipo de reductora que permita un movimiento más lento y controlado del robot móvil.

Creo que podemos llegar fácilmente a la conclusión de que la plataforma aún necesita trabajo para poder ser empleada en proyectos de mayor envergadura. Espero que este proyecto sirva como base para futuras implementaciones prácticas de la plataforma, con el objetivo de que se puedan conocer de antemano las limitaciones y por tanto las posibilidades de esta.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Referencias

[1] Microsoft, Kinect. 2010.

[2] Vicon. [En línea] https://www.vicon.com/software/tracker/

[3] Szalóki, Dávid & Koszó, Norbert & Csorba, Kristóf & Tevesz, Gábor. (2013). Marker Localization with a Multi-Camera System. ICSSE 2013 - IEEE International Conference on System Science and Engineering, Proceedings. 10.1109/ICSSE.2013.6614647.

[4] *Moonsub Byeon, Songhwai Oh, Kikyung Kim, Haan-Ju Yoo and Jin Young Choi. Efficient Spatio-Temporal Data Association Using Multidimensional Assignment in Multi-Camera Multi-Target Tracking. In Xianghua Xie, Mark W. Jones, and Gary K. L. Tam, editors, Proceedings of the British Machine Vision Conference (BMVC), pages 68.1-68.12. BMVA Press, September 2015.*

[5] Wikipedia, *Interfaz de programación de aplicaciones*.

[6] Elisa Hidalgo Moreda, *Seguimiento de objetos móviles con sistema de posicionamiento multicámara. TFG Universidad de Sevilla, 2020.*

[7] Teledyne Dalsa, *Genie Nano.*

[8] Pololu, *Baby Orangutan B-328 Robot Controller.*

[9] Microsoft*, Visual studio* 2019.

[10] Benoît Jacob, Gaël Guennebaud, *Eigen.*

[11] Intel Corporation*, OpenCV.*

[12] *Sapera LT SDK*. [En línea] *https://www.teledynedalsa.com/en/products/imaging/vision-software/sapera-lt/*

[13] MathWorks*, What is Camera Calibration?*

[14] MathWorks*, Camera Calibrator.* [En línea] https://es.mathworks.com/help/vision/ref/cameracalibrator-app.html

[15] Pololu, *Programming Orangutans and the 3pi Robot from the Arduino Environment*.

[16] Dr. Martin Lauer*, Machine Vision Chapter 7: Camera Optics. Transparencias asignatura Machine Vision (Karlsruhe Institute of Technology) 2019.*

***[17]** Marcos Pérez Rus, Experiencias en calibración de sistema multicámara para seguimiento tridimensional de objetos, TFG Universidad de Sevilla 2020.*

[18] *A Multiple-Camera System Calibration Toolbox Using A Feature descriptor-based calibration pattern.Li, Bo, y otros. Tokyo : IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),2013.*

[19] *Bouguet, Jean-Yves. Camera Calibration Toolbox for Matlab. 1999. [En línea] http://www.vision.caltech.edu/bouguetj/calib_doc/.*

[20] MathWorks*, Stereo Camera Calibrator.* [En línea] https://es.mathworks.com/help/vision/ref/stereocameracalibrator-app.html

[21] *Perspective-n-point pose, 2019.*

[22] OpenCV*, Camera Calibration and 3D Reconstruction.*

[23] *OpenCV, Image Segmentation with Distance Transform and Watershed Algorithm.*

[24] *Landis Markley, Yang Cheng, John Crassidis, and Yaakov Oshman,* Averaging quaternions*, Journal of Guidance, Control, and Dynamics* ***30*** *(2007), 1193–1196..*

[25] *Eigen, Eigen::Quaternion< _Scalar, _Options > Class Template Reference .*

[26] Julio Echeverri (https://julioecheverri.wordpress.com/), *Trabajando con Baby Orangutan y Arduino IDE, 2017..*

[27] GitHub, PowerBroker2, *ArduSerial*. [En línea] https://github.com/PowerBroker2/ArduSerial

[28] *cppreference.com, std::thread.*

[29] *Zahaby, Mohammad & Gaonjur, Pravesh & Farajian, Sahar. (2009). Location tracking in GPS using Kalman Filter through SMS. 1707 - 1711. 10.1109/EURCON.2009.5167873.*

[30*]* GitHub, tomstewart89, *BasicLinearAlgebra*. [En línea] https://github.com/tomstewart89/BasicLinearAlgebra

Anexo: Enlace a repositorio

En el siguiente enlace está disponible la carpeta de Drive donde se han guardado alguno de los códigos más importantes empleados en el proyecto.

https://drive.google.com/drive/folders/1Sic4UOHL4XyRdxlznHHbmo_V7js8EkgJ?usp=sharing