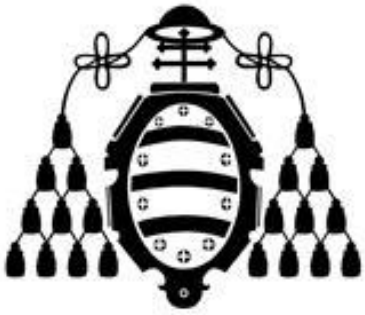


UNIVERSIDAD DE OVIEDO

CENTRO INTERNACIONAL DE POSTGRADO

MÁSTER EN INGENIERÍA MECATRÓNICA

TRABAJO DE FIN DE MÁSTER

DISEÑO DE PROTOTIPO DE RECOGIDA AUTOMATIZADA DE BOLOS MEDIANTE BRAZO ROBÓTICO Y VISIÓN ARTIFICIAL

FEBRERO 2015

ALUMNO: MANUEL GARCÍA POSADA

TUTORES: JUAN DÍAZ GONZÁLEZ Y JOSÉ MANUEL SIERRA VELASCO

AGRADECIMIENTOS

Resulta imprescindible el comenzar este texto expresando un sincero agradecimiento a las personas que han colaborado conmigo de una forma u otra para conseguir sacar adelante este proyecto:

A **Juan Diaz González, Jose Manuel Sierra Velasco** y **Miguel Ángel José Prieto**, sin los cuales ni siquiera hubiera dispuesto de la oportunidad de matricularme este año para realizar la entrega y defensa del trabajo.

A **Fernando José Soto Sánchez**, que es un compañero excepcional, y que ha estado disponible para consultas de todo tipo y para animar cuando me faltaba la motivación.

A todos los profesores que me han atendido, siempre amablemente, cuando necesité consultarles: **Rafael Corsino de los Reyes**, **Ignacio Álvarez García** y **Juan Carlos Álvarez Álvarez**.

Absolutamente necesario recordar a mis compañeros en el antiguo HUCA: **Venancio Marcos Tuñón**, **Javier Pérez Fernández**, **Alberto García García** y **José Manuel Menéndez Del Valle**, por haber estado siempre dispuestos a cambiar turnos en incontables ocasiones, para que pudiera asistir a las clases y a los exámenes.

Finalmente, agradecer a mi esposa **Tanya** y mi familia su eterna paciencia y comprensión. Tanto en lo referente a este máster como en el resto de locuras en las que acabo involucrándome.

PALABRAS CLAVE

BRAZO; BOLO; OPENCV; COORDENADAS; DETECCIÓN; IMAGEN

ÍNDICE

1. **INTRODUCCIÓN** 5

2. **MOTIVACIÓN Y CONTEXTO** 6

2.1. **LA BOLERA ASTURIANA** 6

2.2. **ANTECEDENTES: PINSETTER** 8

3. **ESPECIFICACIONES DE DISEÑO** ..... …12

4. **ALTERNATIVAS CONTEMPLADAS** 13

4.1. **SISTEMA DE DETECCIÓN** 13

4.1.1 Detectores de barrera 13

4.1.2 Fotoresistores 15

4.1.3. Sensores inductivos 16

4.1.4. Visión artificial 17

4.2. **RECOLECCIÓN Y MANIPULACIÓN** 22

4.2.1. Sistema de poleas y cuerdas 22

4.2.2. Brazo robótico 22

5. **DESCRIPCIÓN GENERAL DEL SISTEMA SELECCIONADO** 24

6. **MÓDULO 1 : VISIÓN ARTIFICIAL** 27

6.1. **ADQUISICIÓN DE IMAGENES** 27

6.2. **SOBRE OPENCV** 28

6.3. **DESCRIPCIÓN DE PROCESO DE VISIÓN ARTIFICIAL** 30

6.4. **PROCESAMIENTO DE LA IMAGEN** 31

6.4.1. Conversión a HSV y filtrado por color 31

6.4.2. *Segmentación de la imagen* 33

6.4.3. *Operaciones morfológicas: Dilatación y erosión* 34

6.4.4. Encontrar contornos 37

6.4.5. Localización de centro de masas 39

6.4.6. Alternativa para seguimiento de objetos 42

6.4.7. Calibrado de cámara 45

6.4.8 Transformación de coordenadas. Homografía 52

6.4.9. Funcionamiento de programa de detección 57

7. **MÓDULO 2 : BRAZO ROBÓTICO** 60

7.1. **DESCRIPCIÓN GENERAL** 60

7.2. **TARJETA CONTROLADORA: BOTBOARDUINO** 63
7.2.1. Arduino 68
7.2.2. Programa de control de brazo robótico 70
7.2.2.1. Cálculos cinemáticos 73
7.2.2.2. Cálculo geométrico de la cinemática inversa 78
7.2.2.3. Cinemática directa e inversa con Matlab 81
7.2.3. Calibración de servos 96
7.2.4. Comunicación 98

8. **DESARROLLO DE INTERFAZ GRÁFICA DE USUARIO** 101

8.1. **DESCRIPCIÓN GENERAL DE FUNCIONAMIENTO** 101
8.2. **DESARROLLO DE LA INTERFAZ** 105
8.2.1. Introducción 105
8.2.2. Instalación de Qt y openCV 106
8.2.3. Entorno de desarrollo de Qt 111
8.2.4. Adaptación de código de Visual Studio 114
8.2.5. Comunicación con Arduino 116
8.2.6. Aplicación de eventos con puntero 119

9. **PROTOCOLOS DE PRUEBAS** 121

9.1. **MÓDULO 1: DETECCIÓN** 121
9.2. **MÓDULO 2 : BRAZO ROBÓTICO** 124
9.3. **MÓDULO 3: INTERFAZ GRÁFICA DE USUARIO** 130
9.4. **RECOPILACIÓN DE RESULTADOS** 131
9.5. **MONTAJE DE PROTOTIPO PARA PRUEBAS** 134

10. **PRESUPUESTO** 136

11. **CONCLUSIONES** 137

11.1. **ALCANCE Y LIMITACIONES** 137
11.1.1. Alcance 137
11.1.2. Limitaciones 138
11.2. **PROPUESTAS DE LINEAS FUTURAS** 138

12. **BIBLIOGRAFÍA** 140

13. **ANEXO : MANUAL DE INSTRUCCIONES DE INTERFAZ** 141

1. INTRODUCCIÓN

El objetivo de este trabajo es el diseño de un prototipo de sistema mecatrónico para la recogida de bolos automatizado, contando adicionalmente con una interfaz que permita al usuario su manejo.

Inicialmente se plantea el desarrollo de este sistema para estudiar su posible incorporación al juego de bolos asturianos, aunque posteriormente ha sido necesario reducir el alcance del trabajo para conseguir metas factibles, que no obstante, podrían ser igualmente implementadas en un sistema que sí que cumpliera esos requisitos iniciales.

Se presentarán una serie de posibles soluciones para abordar el problema y posteriormente se realizará una valoración de cada una de ellas y se elegirá la que reúna una mayor idoneidad, siempre y cuando se ajuste a las posibilidades reales de que se disponen para construir un prototipo funcional.

El problema a resolver es ofrecer un sistema de recogida que implique la interacción de un elemento mecánico, que se encargará de la manipulación de los bolos, y un sistema de control que regule las operaciones a realizar y que ofrezca una precisión aceptable.

Para facilitar la interacción del usuario con el software de control creado para realizar la tarea que se nos presenta se ha desarrollado además una interfaz que hace más intuitivo el proceso.

Se han estudiado lenguajes de programación, entornos de desarrollo de software y librerías, se han realizado pruebas y montajes preliminares y finalmente se ha integrado todo el material disponible en la construcción de un prototipo.

En la presente memoria se expone el proceso de desarrollo de las distintas partes que integran el proyecto, así como las pruebas realizadas y los resultados obtenidos. Se ofrecerán también alternativas desechadas con los motivos pertinentes y se indicarán características y funcionalidades adicionales que se podrían añadir a posteriori y que se han descartado por falta de medios, conocimientos o tiempo.

En el apartado final se podrán encontrar las conclusiones obtenidas y el grado en el que se satisfacen las condiciones que se han propuesto inicialmente. También se hará mención de los problemas encontrados durante el proceso de diseño y montaje del prototipo y se mostrará un presupuesto desglosado.

2. MOTIVACIÓN Y CONTEXTO

La motivación de este proyecto es la de ofrecer una solución satisfactoria para la recogida de bolos en el juego de bolera asturiana en modalidad cuatreada.

El desarrollo de un sistema de estas características dotaría de más comodidad a los jugadores para poder disfrutar y centrar su atención exclusivamente en el juego haría innecesario que alguien se hiciese cargo de la recogida de los bolos y de su colocación, algo que supone un tiempo y esfuerzo que podrían reducirse con la implantación de una automatización, al igual que sucede en muchos otros terrenos, como el industrial.

2.1. LA BOLERA ASTURIANA

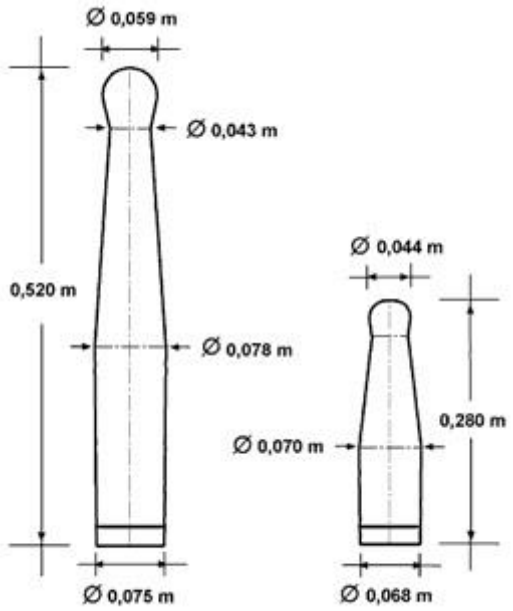
El juego de bolos se supone de origen egipcio y consiste en derribar el mayor número de bolos, que serán una pieza de madera torneada con una forma cilíndrica rematada por una cabeza esférica, mediante el lanzamiento de una bola o pieza, que será de la misma forma habitualmente de madera también.

Este juego goza de gran tradición en países como Alemania, Francia, Holanda y Estados Unidos, y en España también existen diferentes modalidades, con una serie de peculiaridades que las caracterizarán según la región en la que se practique.

La bolera asturiana es un juego tradicional que se disputa en un terreno horizontal libre de obstáculos, en el que habrá presente una estructura llamada Castro, una zona de tiro y una zona intermedia. El conjunto será denominado bolera y deberá de contar con unas dimensiones mínimas de 25m de largo por 5m de ancho.

Esquema de bolera asturiana

Los bolos utilizados para el juego serán 10, 9 de los cuales serán iguales en lo referente a sus dimensiones, mientras que habrá un décimo que será considerablemente más pequeño.



Dimensiones de bolos normales y biche

El juego consiste en lanzar las bolas desde la zona de tiro al castro, que será donde se encuentren situados los bolos. La bola debe de efectuar su caída a tierra (posada) dentro de los límites del castro.

Los bolos serán colocados sobre tacos fijados a una base para que conserven su verticalidad y serán dispuestos en tres filas de tres bolos cada uno.

La situación del biche señala la dirección del efecto que debe imprimirse a las bolas a fin de conseguir la cuatreada, que consistirá en derribar, además de uno o más bolos, el biche o en su defecto, que cruce por la zona cuatreada.

Teniendo en cuenta estas características destacadas de entre todo el reglamento, del que se ha dispuesto a través del sitio web http://www.bolosasturias.com/ , podremos considerar alguno de los requisitos que podríamos plantear a nuestro proyecto, como podría ser la capacidad para distinguir bolos de distintos tamaños y situarlos en posiciones diferentes, y que en el caso del biche supondrían además una variación en cada nuevo turno.

2.2. ANTECEDENTES: PINSETTER

Se ha tenido en cuenta la existencia de algunos sistemas de recogida de bolos automatizados, como los que están presentes en el bowling o bolera americana.

El modelo más conocido en Estados Unidos es el **pinsetter** de **Brunswick**, que fue inicialmente patentado por **Gottfried Schmidt** e introducido en el país por la **American Machine and Foundry Company** (AMF) en 1946.

Este modelo recoge los bolos de la modalidad más extendida, que es el tenpin bowling (10 bolos). En la que se desliza una bola por una calle con una superficie pulida para facilitar el desplazamiento de esta y se intentan derribar la mayor cantidad de bolos posibles. A cada lado de la calle por la que se desliza la bola existen unos canales que recogen la bola que se sale de los extremos de dicha calle y la devuelve a la zona de tiro.

El modelo **Brunswick GSX pinsetter** es el que se representa en la imagen. Es uno de los más modernos de la industria del bowling y dispone de cuatro partes principales:

- Barrido
- Elevador
- Distribuidor
- Bandeja de bolos

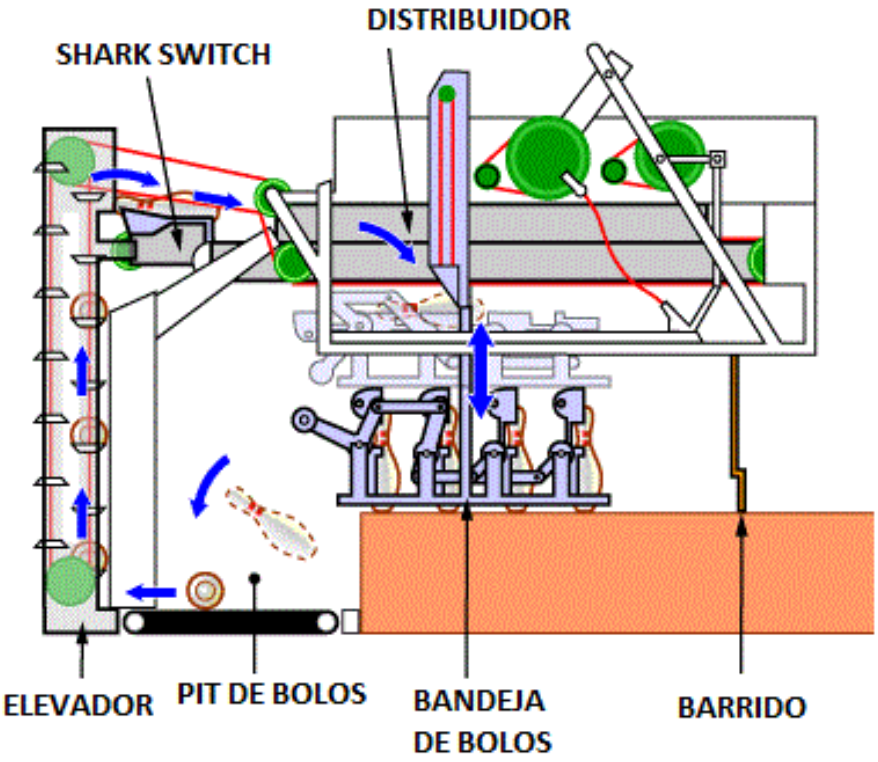
Un pinsetter automático trabaja con el doble de bolos de los que componen una jugada (20 bolos) y realizará su función en ciclos que serán ejecutados cada vez que la bola es rodada. Necesitará saber cómo está procediendo el juego y la cantidad de bolos que han sido derribados. Para ello, en la actualidad se utilizan cámaras CCD montadas al final de la calle,

aunque en muchas ocasiones, en previsión de un funcionamiento defectuoso de la cámara, se utilizan como apoyo unos "fingers", que fue el sistema que le precedió en el tiempo. Una vez que el pinsetter dispone de la información de la tirada actual, se dispone a levantar los bolos que se mantienen en pie y a barrer los derribados "Deadwood". Una vez completado el barrido, se vuelve a colocar el bolo que se mantuvo en pie para darle una oportunidad al jugador de derribarlo. El proceso dará comienzo tan pronto como la bola se encuentre rodando por la calle.

Existe un sensor colocado a una corta distancia frente los bolos que está configurado con un retraso de un segundo o más para permitir que la bola impacte en los bolos y termine en su lugar de recogida antes de empezar a realizar sus rutinas.

El barrido descenderá a una posición de defensa (guard) frente a los bolos. Este será una estructura metálica que se extiende hacia abajo frente a los bolos para protegerlos de una posible colisión con una bola lanzada durante la duración del proceso.

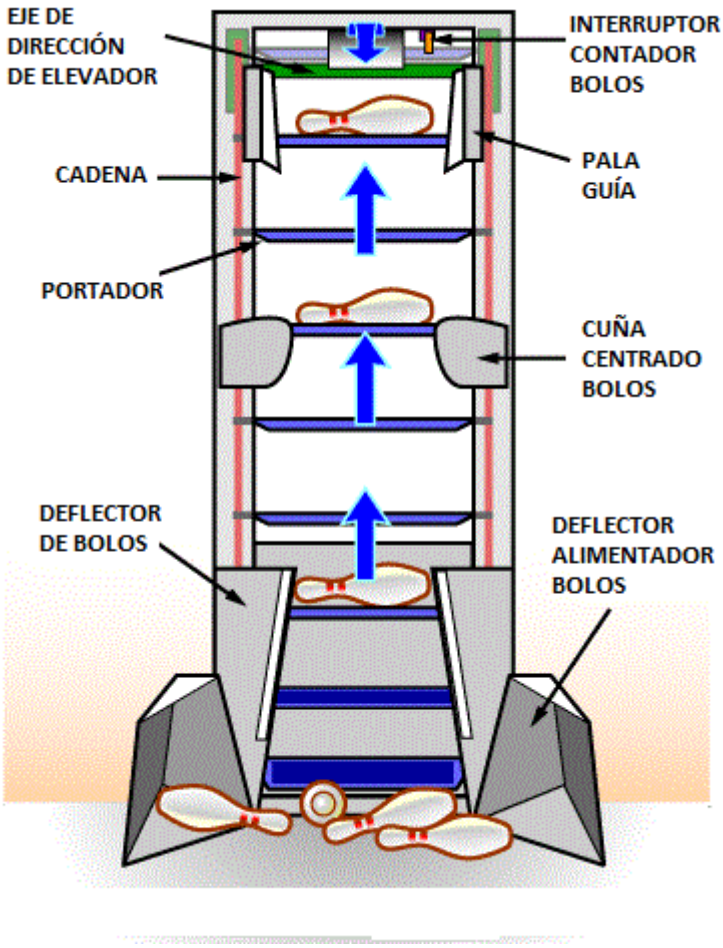
La bandeja de bolos consistirá en diez agujeros, cada uno de los cuales será lo suficientemente grande para que se pueda ajustar dentro un bolo y será depositado sobre ellos para arrastrarlos al pit. Los bolos que se recogen porque se mantuvieron en pie se fijan a los agarres con la acción de una solenoide.



Corte lateral pinsetter

Cuando los bolos se encuentran de vuelta en la calle, se dispara un interruptor normalmente, que abrirá los agarres a los que estaban sujetos los bolos que permanecieron en pie y se les deja sobre la superficie de la calle de nuevo.

Una vez dispuestos los bolos en el pit, se dispara un interruptor que transportará diez bolos con el elevador y los introducirá mediante el shark switch, que asegurará su reparto en la posición adecuada, en el distribuidor, donde se prepararán para su reposición al finalizar el segundo lanzamiento. En el transcurso de este segundo lanzamiento no será necesario recoger con las solenoides los bolos que se mantengan en pie.

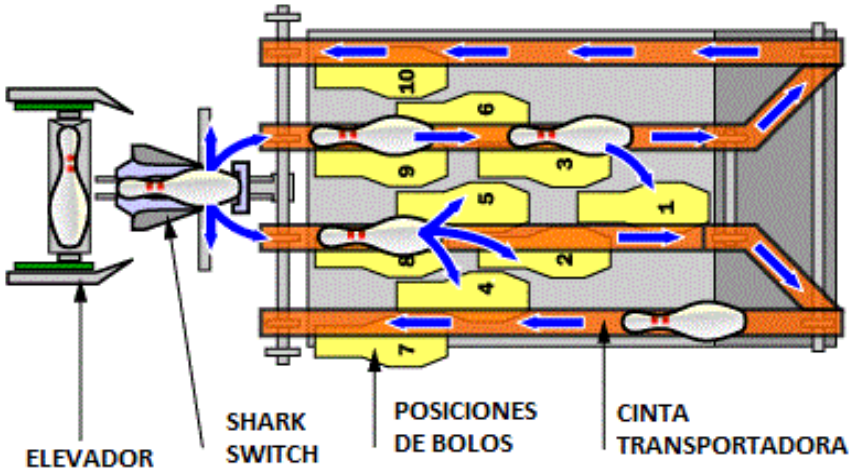


Elevador pinsetter

En el esquema sobre estas líneas se muestra el sistema de recogida y elevación de bolos: Después de una tirada, bola y bolos se recogerán en el espacio conocido como pit, para ser acarreados mediante una cinta transportadora. La bola será la única que tenga las dimensiones y peso que activen un sensor que le hace recorrer una ruta alternativa, mientras que los bolos se conducen al elevador.

Este elevador constará de una docena de bandejas instaladas en un sistema de dos poleas. Estas conducirán a los bolos hasta el mecanismo antes mencionado (shark switch), que pivotará una bandeja en forma de embudo y que los posicionará de la forma adecuada para su colocación en el distribuidor.

La presencia o no de una cámara determinará el siguiente ciclo. Si esta está presente, el ciclo consistirá en un barrido simple de los bolos y el posicionamiento de diez nuevos bolos para el siguiente turno.



Distribuidor pinsetter

Pero, a pesar de que este sistema de recogida está perfectamente diseñado y probado y realiza su cometido de manera excelente, resulta necesario la consideración de ciertos factores que diferencian el desarrollo del juego: En la bolera asturiana, al contrario que en la americana, se juega al aire libre. La bola es de unas dimensiones más reducidas y no se desliza por la pista, sino que vuela hasta impactar con los bolos, que saldrán despedidos dentro de un espacio delimitado que es conocido como Castro. En la bolera asturiana los bolos tampoco son todos iguales, ya que existen unos de dimensiones más reducidas conocidos como biches, que cambiarán su localización además a lo largo de la partida. Justo es también mencionar que las dimensiones de la bolera asturiana requerirían de un inversión económica considerable para poder desarrollar un prototipo funcional que realizase el mismo servicio que este modelo.

3. ESPECIFICACIONES DE DISEÑO

Se han planteado los siguientes requisitos a cumplimentar para la consecución del objetivo expuesto anteriormente y que, al término de este trabajo, serán correspondientemente evaluados con el fin de determinar el grado en el que se han satisfecho y cuál ha sido el alcance de los resultados obtenidos y sus limitaciones:

- El sistema ha de tener un alto grado de autonomía y ha de permitir también la operación manual al usuario.
- Se ofrecerá una solución económicamente viable que implique una instalación de menor complejidad estructural que las actualmente disponibles y que sea posible adaptar para su uso en distintas boleras con modificaciones mínimas.
- El sistema debe de adaptarse a las características específicas del desarrollo del juego de bolos asturianos.
- Se debe de integrar de manera correcta un elemento mecánico que sirva para efectuar la recogida de bolos con un sistema de control informático.
- Se debe de garantizar la correcta comunicación entre todos los elementos del sistema.
- Se implantará un sistema que reconozca los bolos por su tamaño y color para su posterior colocación.
- Será necesario distinguir entre bolos a recoger (derribados) y bolos que se mantendrán en su posición, al no haber sido derribados.
- El sistema de detección y reconocimiento de bolos deberá de proveer de la información suficiente para que la recogida y manipulación de estos se realice de una forma adecuada.
- El sistema mecánico escogido para la manipulación y recogida de bolos ha de tener un grado de precisión adecuado que garantice que se produzca la operación de localización, recogida y colocación con un alto grado de repetibilidad.
- El proceso de recogida y manipulación de bolos se realizará sin que existan colisiones con elementos estructurales del prototipo y sin interrupciones.
- Se asegurará que el control del sistema asigne a los bolos recogidos una posición predeterminada que esté disponible, a la que conducir a estos para su colocación.
- El usuario interactuará con el sistema a través de una interfaz gráfica que permitirá realizar todas las operaciones posibles y a través de la cual se monitorizarán todos los procesos.

4. ALTERNATIVAS CONTEMPLADAS

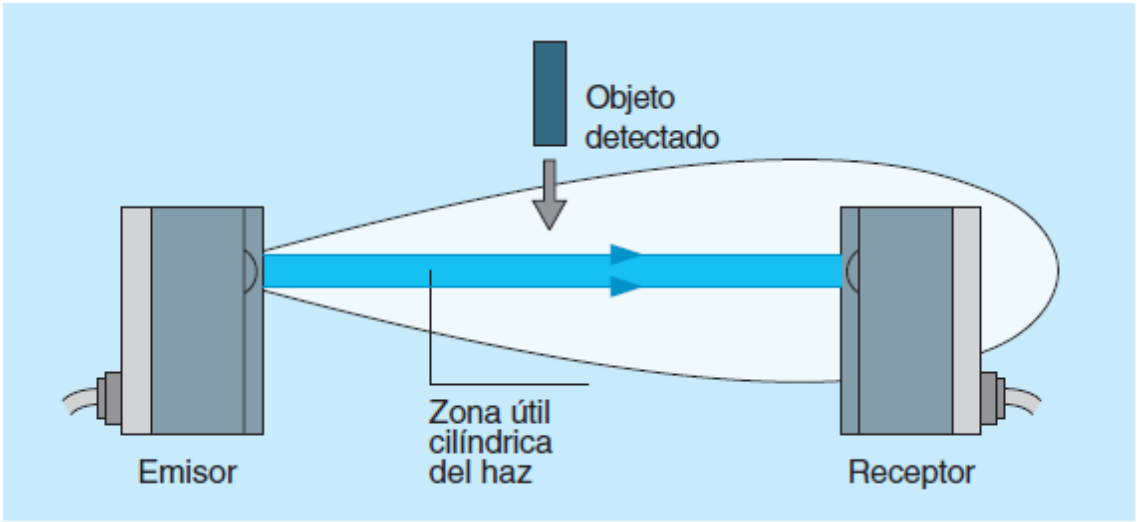
Tras plantear el problema a solucionar y los requisitos para su cumplimiento, se han contemplado una serie de posibles acercamientos, que se recopilarán en este apartado con una valoración sobre su posible idoneidad y los motivos que han llevado a su implantación o a su descarte.

4.1. SISTEMA DE DETECCIÓN

Para garantizar una correcta detección de bolos, en un sistema que nos ofrezca un posicionamiento con alto grado de exactitud y nos proporcione a su vez información adicional sobre las características del objeto detectado se han tomado en consideración las siguientes alternativas:

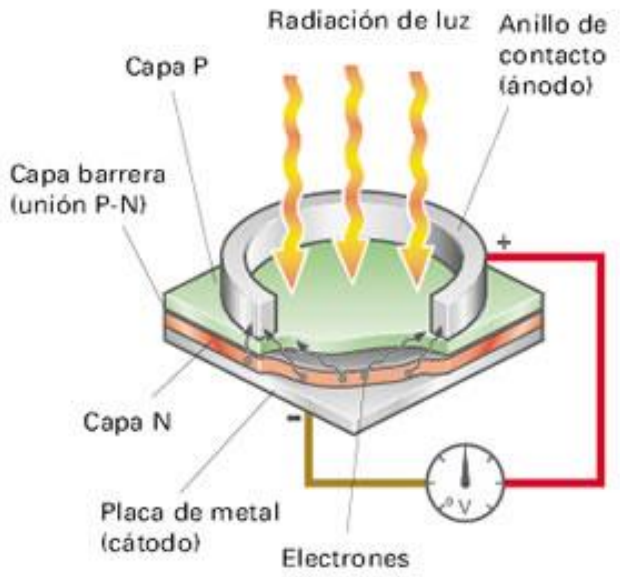
4.1.1. DETECTORES DE BARRERA

Se ha estudiado la posible utilización de detectores de proximidad ópticos para realizar la detección de los bolos caídos. Estos dispositivos están compuestos de una fuente de luz (LED, generalmente) y un fotodetector (fotodiodo). Estos se encontrarán en extremos opuestos y se dispondrán alineados. El emisor generará un haz de luz modulado que se dirige directamente al receptor, el cual recibirá la luz del emisor a través de un filtro que elimina las radiaciones de frecuencia diferente a la generada por el emisor y un diafragma, que reduce el ángulo de paso de luz. Cuando un objeto interrumpe el haz de luz entre el emisor y el receptor la señal del receptor cambia de estado.



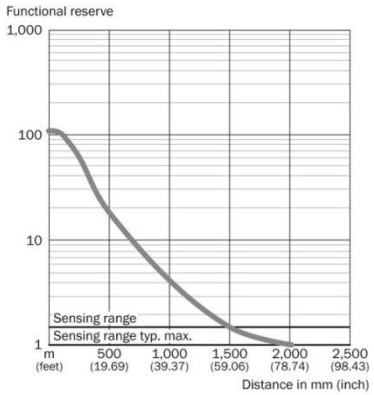
Esquema de funcionamiento de detectores de barrera

Un fotodiodo es un diodo en el que se generará un par electrón-hueco al incidir sobre él un fotón con una energía mayor que la correspondiente a la banda prohibida del semiconductor. Esta generación ocurrirá en la zona P, en la N y en la zona de transición. Si ocurre en esta última zona, el campo eléctrico acelera los electrones hacia la zona N y los huecos hacia la P, apareciendo carga negativa en la zona N y positiva en la P. Si cerramos un circuito externo entre el cátodo y el ánodo del fotodiodo, los electrones fluirán desde N y los huecos desde P hasta los electrodos opuestos, generando una corriente eléctrica, denominada fotocorriente.



Funcionamiento fotodiodo

En la documentación de los sensores fotoeléctricos que emplearemos en detector de barrera podremos encontrar, aparte de características como la resolución, sensibilidad, rango máximo, tipo de luz emitida, tiempo de respuesta, temperatura ambiente o corriente de salida, una curva de exceso de ganancia, que nos servirá para predecir si se adaptará al uso que se pretende dar.



En función del ambiente en el que pretendamos utilizar el sistema, debemos de contar con un exceso de ganancia determinado. En nuestro caso sería un ambiente sucio, al ser utilizado en una pista de tierra, lo que supondría la necesidad de un exceso de ganancia de 50x aproximadamente.

Este sistema, por tanto nos ofrecería las siguientes ventajas:

- Rango de detección elevado
- Adecuado para ambientes sucios
- Detección muy precisa debido a su amplio margen de ganancia
-

Los inconvenientes que tendría serían:

- Supondría un coste elevado el realizar la instalación, debido a que el emisor y receptor están separados y requiere más cableado
- Sería complicada su instalación debido a las características del terreno en el que se pretende implantar. No sería posible construir una estructura en la que se pudieran instalar los sensores sobre el terreno en el que se fijan los bolos sin tener que hacer modificaciones sobre este para facilitar su tarea
- Imposibilidad de distinguir el biche del resto de bolos

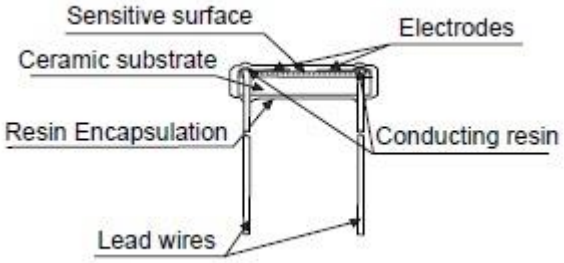
Tras la consideración de estos inconvenientes mencionados, podremos por tanto concluir que no será aplicable para nuestro trabajo y se decide, en consecuencia, descartar su aplicación.

4.1.2. FOTORESISTORES

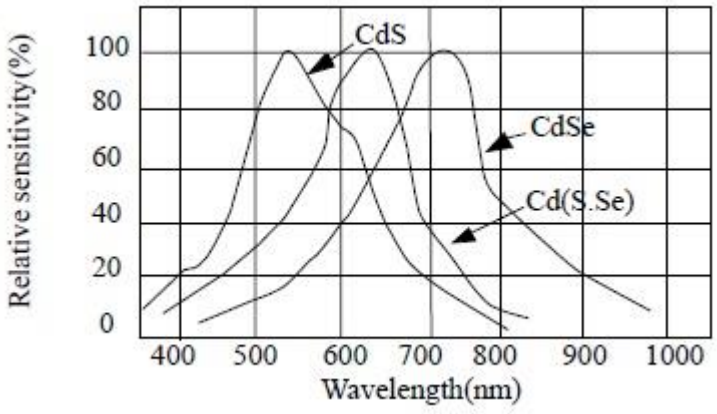
Otra opción que se ha estudiado es la incorporación de baterías de fotoresistores (**LDR**) para detección de color. La disposición de estos sensores sería análoga a la comentada con los sensores de barrera, de forma que cuando un bolo fuera derribado y entrase en el campo de

detección de los sensores, estos responderían con un cambio en el valor de su resistencia que se podrá interpretar y localizar, de esta forma, el bolo.

El principio de fucionamiento de este sistema sería aplicar una luz determinada, con un LED por ejemplo, sobre los objetos a identificar. En función de la luz reflejada por dichos objetos, es posible determinar si el objeto es de un color u otro debido a la variación en la resistencia que experimentará el fotoresistor.



Model	Vmax (VDC)	Pmax (mW)	Ambient Temp (°C)	Spectral Peak (nm)	Photo Resistance (10Lx) (KΩ)	Dark Resistance (MΩ)min	γ min	Response Time (ms)	
								Rise	Decay
PGM2000-PP	500	500	-30 ~ +70	560	2~5	1.0	0.6	30	40



En la gráfica inferior se puede apreciar la variación de la sensibilidad de los fotoresistores en dependencia de la longitud de onda de la luz a la que están expuestos, con picos de tolerancia de ± 50nm.

El valor de la resistencia eléctrica de estos fotoresistores es bajo cuando hay luz incidiendo sobre él (hasta 50 ohmios) y muy alto cuando está a oscuras (1 o más megaohmios)

Ejemplo de fotoresistor con LEDS para detección de color

Los fotoresistores tienen diferentes niveles de sensibilidad para diferentes colores. Un valor óptimo sería de 520nm. También tiene influencia la distancia a la que se encuentran del objetivo. Sería también muy importante conseguir reducir el impacto de los cambios de la luz ambiente, para lo cual habría que proteger o aislar de alguna forma los sensores empleados

Las ventajas que supondría utilizar un sistema de estas características serían:

- Alta sensibilidad.
- Fácil instalación a bajo coste.

Y las desventajas observadas son las siguientes:

- Efecto de histéresis.
- Falta de linealidad entre resistencia e iluminación.
- Gran influencia de los cambios de iluminación ambiente.

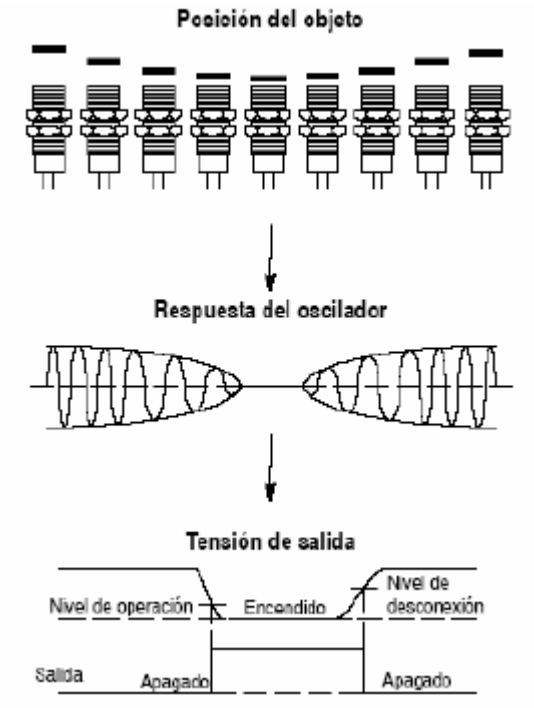
Al ser los bolos asturianos un juego que se realiza al aire libre, y debido a las complicaciones que podrían suponer realizar una instalación de este tipo y no poder garantizar la integridad de los sensores, se descarta esta opción en la fase de estudio del problema.

4.1.3. SENSORES INDUCTIVOS

Otra posibilidad contemplada durante la etapa de estudio del problema ha sido la de la detección de los bolos mediante sensores de proximidad inductivos, tras modificar el cuello de estos (los bolos) añadiéndoles una anilla metálica.

Estos dispositivos son de uso muy frecuente en plantas industriales, ya que presentan buenos resultados en la detección de objetos metálicos.

Su principio de funcionamiento es el siguiente: Se aplica la salida de un oscilador a una bobina de núcleo abierto capaz de generar un campo electromagnético en sus proximidades; la presencia de objetos metálicos en la zona modificaría el campo y se manifestarían algunos cambios en las magnitudes eléctricas de la bobina. Los cambios podrían detectarse y saber así si existe o no un objeto metálico dentro del radio de acción del sistema.



Funcionamiento de sensores inductivos

Para la elección del sensor inductivo deberíamos tener en cuenta que la bobina sea apantallada o no, ya que esto modificaría la distribución del campo electromagnético y se produciría un cambio en la distancia de detección. En caso de no ser apantallada, esta distancia sería mayor. También se debería de tener en cuenta el tamaño de objeto a detectar, ya que cuanto mayor sea este, mayor será la distancia a la que se pueda

detectar. Por último será necesario tener en cuenta el material del objeto a detectar, porque las corrientes inducidas serían más o menos grandes en función de él y, por tanto, tendrá igualmente mayor o menor influencia sobre el circuito resonante. Sería necesario aplicar factores de corrección para determinar la distancia efectiva de detección.

Las condiciones de uso del sensor, como puede ser la temperatura ambiente, podrán afectar a la distancia de detección, para lo cual se puede determinar una distancia nominal de detección $\mathbf{D_n}$, que se fija a temperatura ambiente (20º), que suele ser alrededor de un 20% menos que $\mathbf{D_n}$ y que se asegura para todos los dispositivos en todas las condiciones de trabajo permisibles.

Las ventajas que presentaría este sistema serían las siguientes:

- No necesitan contacto directo con el objeto a detectar
- No sufren desgaste
- Tienen un tiempo de reacción muy reducido
- Insensibles a polvo y humedad

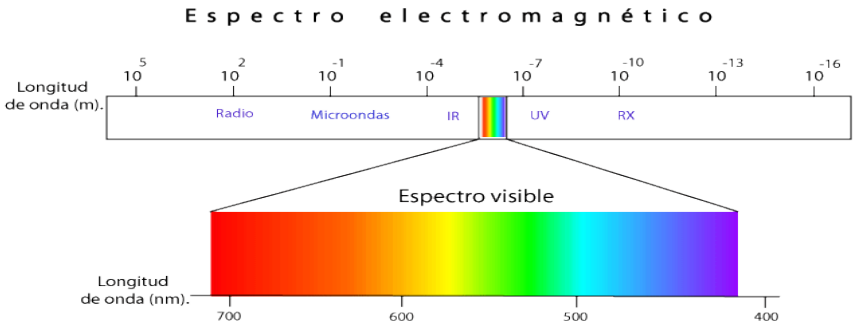
El principal inconveniente que plantearía esta posible opción sería el margen de operación, que es muy corto en comparación con el que ofrecen otros sensores (0,5 a 50 mm) y también tendríamos que diseñar una estructura sobre la que colocarlos bajo la tierra de la zona de juego. Serían necesarios una gran cantidad debido a la superficie que ocupa el campo de trabajo, así que no se adapta a nuestras necesidades y esta opción queda por tanto descartada.

4.1.4. VISIÓN ARTIFICIAL

Visión artificial es un campo de la inteligencia artificial que comprende unas técnicas a través de las cuales se busca modelar de una forma basada en procedimientos matemáticos los procesos de percepción visual propios de los seres vivos y trasladar estas capacidades a programas y aplicaciones por computadora. La visión por computadora permitiría el análisis de propiedades y entorno de un espacio dinámico tridimensional a partir de la captura de imágenes desde uno o varios puntos para construir una escena.

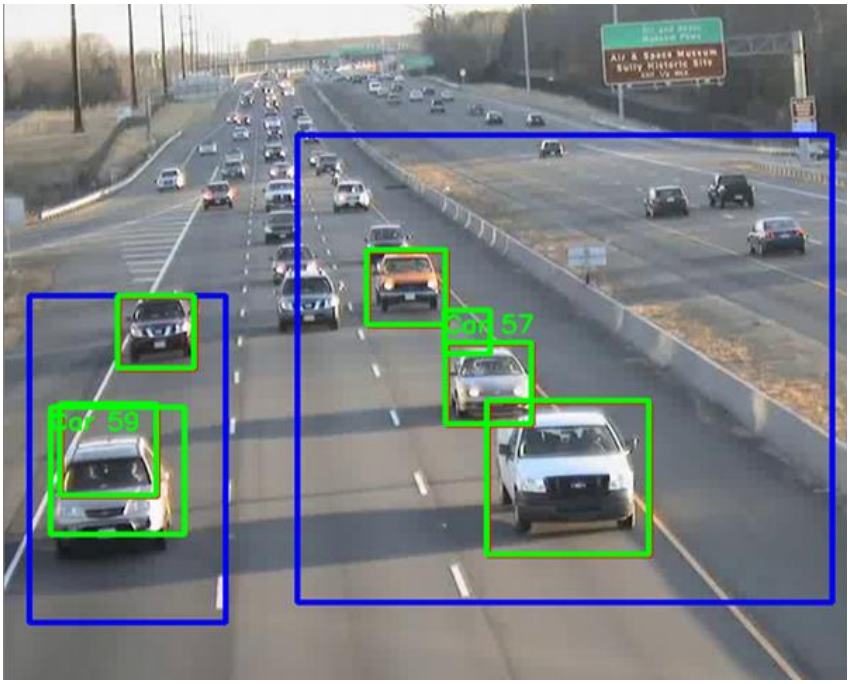
La visión artificial tiene como objetivo adquirir información visual del entorno físico y extraer características de relevancia o útiles. A pesar de no estar tan desarrollada como la visión en los seres vivos, goza de ciertas características que las hacen idóneas para desempeñar tareas que, por su repetitividad por ejemplo, son extenuantes para el ojo humano. Necesario

es además hacer especial hincapié en ciertos aspectos, como la capacidad limitada de la que disponemos los seres humanos para captar un rango de frecuencias y amplitudes dentro del espectro electromagnético. Limitación con la que no cuentan los sistemas de visión artificial, que a parte de la luz visible tienen capacidad para percibir infrarrojos, ultravioletas, rayos gamma, microondas…



Se debe de tener en cuenta también la velocidad de respuesta menor de la visión humana frente a la de las cámaras digitales, la mayor precisión de que disponen. Estos sistemas tampoco sufrirán los inconvenientes creados por el efecto de la fatiga o estados emocionales, ni serán tan sensibles a la exposición a atmósferas peligrosas, entornos de riesgo biológico, zonas de temperaturas extremas, o incluso estando sometidos a radiación.

El uso que en la actualidad se hace de estos sistemas se encuentra actualmente en auge debido a la creciente potencia de los microprocesadores y sus campos de aplicación son cada vez más variados. En el terreno industrial, por ejemplo, es de uso muy extendido en control de procesos y de calidad. Fuera del terreno industrial podemos encontrar aplicaciones ligadas a la seguridad, control de tráfico o incluso en ocio informático.



Visión artificial aplicada al control de tráfico

Estos procesos serían la obtención de imágenes, ya sean monocromáticas o en color, a través de un dispositivo como, en nuestro caso, sería una webcam, la extracción de información espectral, espacial y temporal de interés (por ejemplo a la iluminación e intensidad de la imagen, forma, tamaño, color) y posteriormente interpretar esta información para poder gestionar tareas a realizar a continuación. En el campo en el que desarrollamos en el trabajo, que sería la robótica, podremos desempeñar tareas tan importantes y útiles como son la identificación y localización espacial de objetos, la preparación para planificar el movimiento de nuestro equipo robótico estableciendo una relación espacial o correspondencia entre el sistema de coordenadas de éste y el de 2D que captamos a través de nuestro dispositivo adquisidor de imágenes. La visión artificial sería, en resumen, la forma que tendría nuestro robot de comprender e interpretar su entorno y un medio para poder planificar su interacción con éste.

Hemos tenido en cuenta los siguientes factores al valorar esta opción:

- Resulta económico
- No es necesario realizar adaptaciones en el terreno de juego
- Vamos a poder extraer información de los objetos detectados como pueden ser color, contornos, formas y características geométricas
- Altamente flexible
- Basado en software y fácilmente modificable

- Es posible analizar objetos en movimiento

Su único inconveniente se podría señalar como la posibilidad de obtener errores en la detección dependiendo del grado de iluminación disponible, lo que podría requerir de una calibración.

Haciendo un balance entre las ventajas y desventajas de todas las propuestas surgidas en la etapa inicial de acercamiento al problema, se ha concluido que un sistema de visión artificial se adaptaría mejor a nuestras necesidades.

4.2. RECOLECCIÓN Y MANIPULACIÓN

Se han estudiado dos posibles alternativas para solucionar el problema de la recogida de los bolos. Se han tenido en cuenta a la hora de valorar su inclusión en el trabajo la posibilidad de integración con el sistema de detección, la forma en la que se adaptaría a las características del campo en el que se trabaja y su repetibilidad.

4.2.1. SISTEMA DE POLEAS Y CUERDAS

Esta alternativa no ha sido muy desarrollada y fue contemplada en una etapa inicial de estudio del problema. Su principio de funcionamiento sería el siguiente:

Se instalaría una superficie bajo el terreno de juego con orificios situados en los puntos en los que se colocarían los bolos, y, ancladas a las bases de los bolos, se dispondría de unas cuerdas que serían accionadas con un sistema de poleas accionadas por motores de corriente continua.

A pesar de que podría resultar una opción económicamente viable y la dificultad para su construcción no sería elevada, se descarta esta opción porque en caso de producirse el derribo de varios bolos existiría la posibilidad de que se produjeran enmarañamientos con las cuerdas.

4.2.2. BRAZO ROBÓTICO

La inclusión de un brazo robótico para realizar las funciones de recogida y manipulación de los bolos fue la segunda opción contemplada y la que, a la postre, sería la definitiva.

Los criterios que se han tenido en cuenta para la elección de este dispositivo han sido los siguientes:

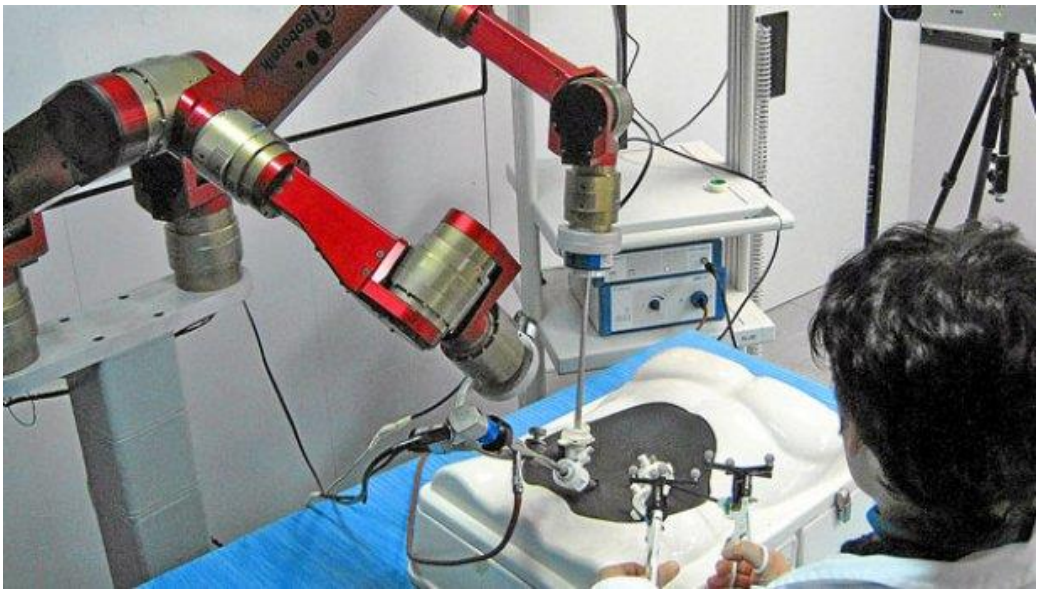
- Fácil integración con el sistema de detección escogido
- Alta precisión y repetibilidad
- La instalación resulta sencilla
- Modificable y adaptable a través de software
- Facilidad de manejo superior para el usuario

Los inconvenientes que hemos tenido en cuenta y asumido son los siguientes:

- Sería necesaria una inversión inicial considerable que podría no ser realmente rentable para esta aplicación.
- Se necesitaría un personal especializado o entrenado para realizar tareas de mantenimiento y operación con este dispositivo
- En nuestro caso, hubiera sido recomendable el diseño y desarrollo de un sistema para desplazar el brazo robótico sobre la zona de juego para poder realizar determinadas operaciones.

5. DESCRIPCIÓN GENERAL DEL SISTEMA SELECCIONADO

Es algo ya habitual y muy frecuente contar en procesos industriales y sistema de producción automatizados con la presencia de brazos robóticos, que facilitan enormemente las tareas reiterativas de manufactura, aseguran precisión, fiabilidad y que, a pesar de la inversión inicial y los ineludibles costes de mantenimiento, reducen los gastos (lamentablemente en ocasiones a costa de la mano de obra humana) y mejoran ostensiblemente la productividad. Pero en la actualidad, cada vez se encuentran aplicaciones de estos elementos en campos muy diversos, como pueden ser la medicina, donde existen ya equipos que tienen la capacidad de realizar intervenciones quirúrgicas con un grado de precisión extrema. Es, por tanto, un terreno fértil y muy adaptable para el que pueden crearse todo tipo de aplicaciones que hagan más llevaderas tareas que requieran cierto grado de destreza y repetibilidad. En nuestro caso necesitamos un dispositivo que nos permita recoger objetos y ¿Qué mejor forma de recoger objetos que un aparato que imita el procedimiento tal como lo realizaría un ser humano?. Esto nos ha llevado a plantear su utilización para la consecución del objetivo propuesto.



Brazo robótico en aplicaciones del campo de la medicina

Por otro lado, el campo de visión artificial o visión por computador está en proceso de crecimiento y cada vez existen herramientas más desarrolladas y más accesibles, que permiten a usuarios con grados de conocimiento menos avanzados diseñar aplicaciones que se adapten a sus necesidades. Por otra parte, integrar este tipo de sensorización al brazo

robótico le añade al proyecto un grado de percepción diferente al que tendría con otros métodos. La visión es probablemente el sentido humano que más información nos proporciona del medio y su implementación en procesos del tipo que abordamos en este trabajo otorgará a nuestro conjunto unas capacidades de percepción del medio que necesitarían de montajes muy complejos y menos versátiles si nos decantásemos por otros sistemas de sensorización. De cara al usuario, además, resulta mucho más intuitiva la operación con este tipo de equipos si se dispone de información visual y, a pesar de la dificultad técnica que supone implementar una solución precisa que cumpla los requisitos planteados, el resultado es una herramienta que responderá mucho mejor en su entorno y que tendrá una mayor capacidad de respuesta y de interacción.

Brazo robótico industrial equipado con un sistema de visión artificial

El tipo de producto que se desarrollará será un prototipo, ya que se asume el hecho de que se van a producir altos grados de error y la complejidad general del proceso supera las capacidades del autor y los medios a su alcance. De todas formas, el sistema desarrollado puede portarse con relativa facilidad a equipos más complejos y adaptarse para un uso más preciso con los medios adecuados en un futuro.

Para poder realizar una recogida adecuada de todos los bolos sería necesario disponer de un sistema para trasladar el brazo robótico en el interior de la zona de juego, lo cual facilitaría también la planificación de

movimiento y evitar obstáculos. En este trabajo no se ofrece una solución para dicho problema y se deja propuesto para una posible ampliación.

Para conseguir la identificación de bolos se obtendrán imágenes en tiempo real de la zona de juego a través de una webcam, que será instalada en una zona elevada por encima de la pinza o manipulador de nuestro brazo robótico y se empleará la biblioteca libre de visión artificial **OpenCV** para desarrollar un programa utilizando el código C++ en el entorno de **Visual Studio 2010** con el que se realizará un tratamiento de estas imágenes, posibilitando así el llevar a cabo una correcta identificación de los bolos a recoger y trasladar y de las posibles ubicaciones para estos que se encuentran disponibles. Tras la obtención de los centroides de los bolos a mover, se procederá a convertir las coordenadas de los cuerpos de estos, obtenidas en píxeles, a coordendas del mundo real (homografía), con las que sea posible suministrar al robot unos datos que se correspondan a los que manejará para su ubicación en el espacio. Se comunicarán las órdenes a ejecutar a un brazo robótico **Lynxmotion AL5A**, de 5 grados de libertad, que dispone de una placa **Botboarduino** (**Arduino duemilanove** modificada para poder utilizar un mayor número de servos) para su control, el cual será realizado mediante un programa creado para la ocasión, que tendrá que ser capaz de recibir las órdenes enviadas a través de la comunicación serie desde el ordenador desde el que se ejecuta el programa de **Visual Studio** e interpretarlas. El brazo robótico recogerá entonces los bolos derribados y los colocará en su correspondiente lugar.

Por último se desarrollará una aplicación en **Qt** para crear una interfaz gráfica de usuario que permita a un operario realizar diferentes operaciones de recogida y monitorización del proceso.

Se dispondrá de esta forma un punto de partida económico y que sería fácilmente modificable para añadirle mejoras e innovaciones en caso de disponerse del presupuesto adecuado, que serán comentadas más adelante en éste documento.

6. MÓDULO 1 - VISION ARTIFICIAL

Se comenzará por este apartado, al ser probablemente el que tiene un mayor peso dentro del desarrollo del proyecto y el que más tiempo ha requerido tanto a nivel de investigación como de aplicación.

6.1. ADQUISICIÓN DE IMÁGENES

La forma en la que se propone adquirir las imágenes será digital, de forma que se contará con un dispositivo con un sensor electrónico, equipado con una serie de unidades fotosensibles que, tras unos muestreos realizados a intervalos de tiempo espaciados regularmente, a través del efecto fotoeléctrico, convertirán la luz en una señal eléctrica, que será digitalizada y guardada en una memoria. Los valores que se obtendrán en cada punto dependerán del brillo de la imagen original en dichos puntos.

La imagen obtenida de forma digital será considerada como una cuadrícula, compuesta de unos elementos llamados Píxeles (Pixel= Picture element), que serán las unidades más pequeñas homogéneas en color que forman parte de la imagen.

La cámara que ha sido seleccionada para obtener las imágenes a procesar será una webcam **Logitech c525**, que permitirá grabar vídeo en HD 720p y la obtención de fotos de 8 megapíxeles (mejoradas por software). También dispone de enfoque automático y la posibilidad de modificar su soporte y plegarse para poder adaptarla a la superficie sobre la que se pretende fijar. La elección se hizo de acuerdo al listado de cámaras compatibles con la versión de **OpenCV** que utilizamos (2.4.6).

Logitech c525

6.2. SOBRE OPENCV

OpenCV (Open Source Computer Vision Library) es una librería de software libre con licencia BSD. Esta nos proporcionará todas las herramientas necesarias para el tratamiento de imágenes obtenidas a través de una webcam y la identificación de objetos. El lenguaje de programación nativo de **OpenCV** es C++ y lo utilizaremos en el entorno de **Microsoft Visual Studio 2010** y **Qt**.

OpenCV es un software multiplataforma que, al ser gratuita y de código libre, dispone de una gran comunidad y de un desarrollo continuo. Esto favorece la resolución de problemas que van dándose durante la realización del proyecto y nos hace disponer de una gran cantidad de documentación y de ejemplos para consulta.

La creación del proyecto **OpenCV** es fruto de una iniciativa de investigación de Intel para la creación de aplicaciones para una serie de proyectos que incluían por ejemplo la proyección de imágenes en 3D en pantallas de pared o trazado de rayos.

Algunas de las áreas en las que se usa actualmente **OpenCV** son: reconocimiento facial, reconocimiento de gestos, interacción hombre-computador, seguimiento e identificación de objetos, sistemas de vigilancia de vídeo o realidad aumentada.

Esta librería dispone de más de 500 logaritmos optimizados para análisis de imagen y vídeo. Desde la versión 2.2, la librería **OpenCV** se divide en varios módulos, que se incluyen dentro de la carpeta lib. Son los siguientes:

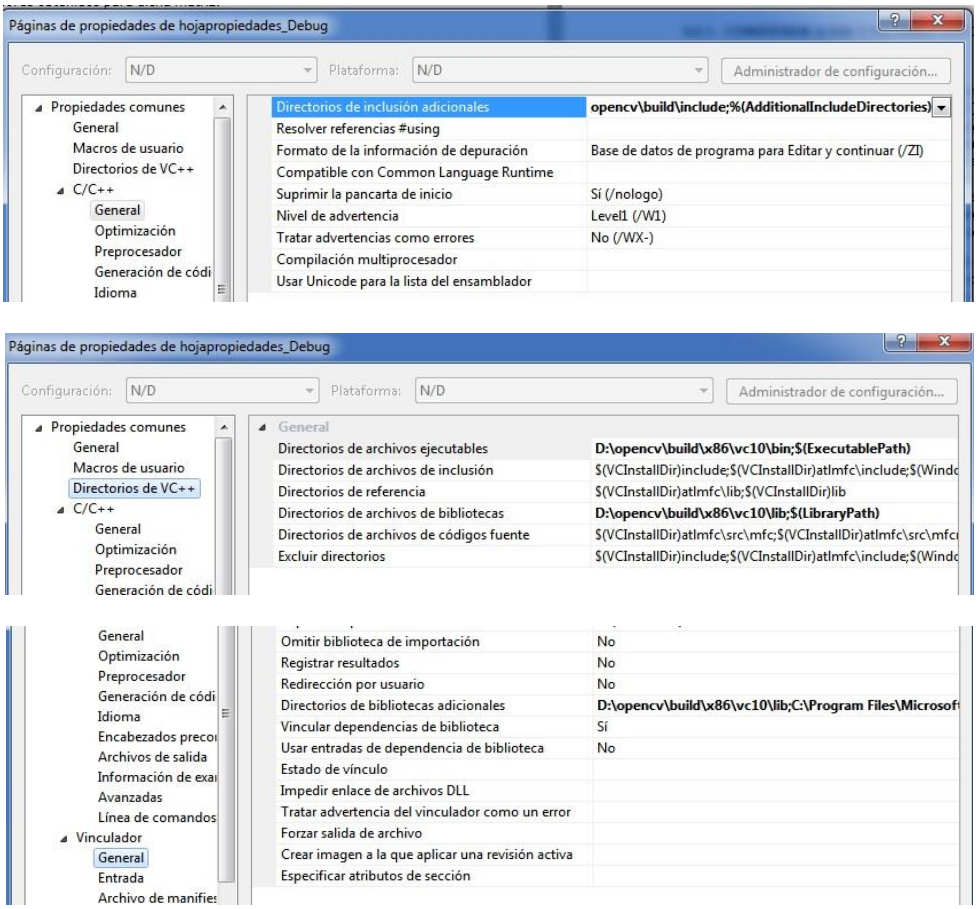
- **Core**: Contiene las funcionalidades básicas de la librería, en particular las estructuras básicas de datos y funciones aritméticas.
- **Imgproc**: Contiene las funciones principales de procesado de imágenes.
- **Highgui**: Contiene las funciones de lectura y escritura de imagen y video, junto con otras funciones de interfaz de usuario.
- **Features2d**: Detectores de puntos y descriptores y correspondencia de puntos de interés.
- **Calib3d**: Calibración de cámara, estimación de geometría y funciones estereoscópicas.
- **Video**: Estimación de movimiento, funciones de seguimiento de características y sustracción de fondo.
- **Objdetect**: Funciones de detección de objetos como caras o personas.

Todos estos módulos tienen un archivo de cabecera asociado a ellos (se puede encontrar en la carpeta include), y deberán de ser incluidos en el códido C++ al comienzo. Por ejemplo:

#include <opencv2/core/core.hpp>

La instalación de **OpenCV** en el equipo sólo requiere visitar su sitio web oficial y descargar la última versión. En versiones actuales no es necesario más que abrir el archivo ejecutable y proceder a seleccionar la carpeta en la que se desea hacer la instalación.

Sin embargo, para su utilización es requisito indispensable agregar la carpeta build al path de nuestro sistema y también hacer unas modificaciones en nuestro proyecto de Visual Studio para poder hacer uso de sus librerías. Este proceso puede realizarse una sola ocasión y después guardar la hoja de propiedades del proyecto para poder utilizarla en nuevos proyectos.



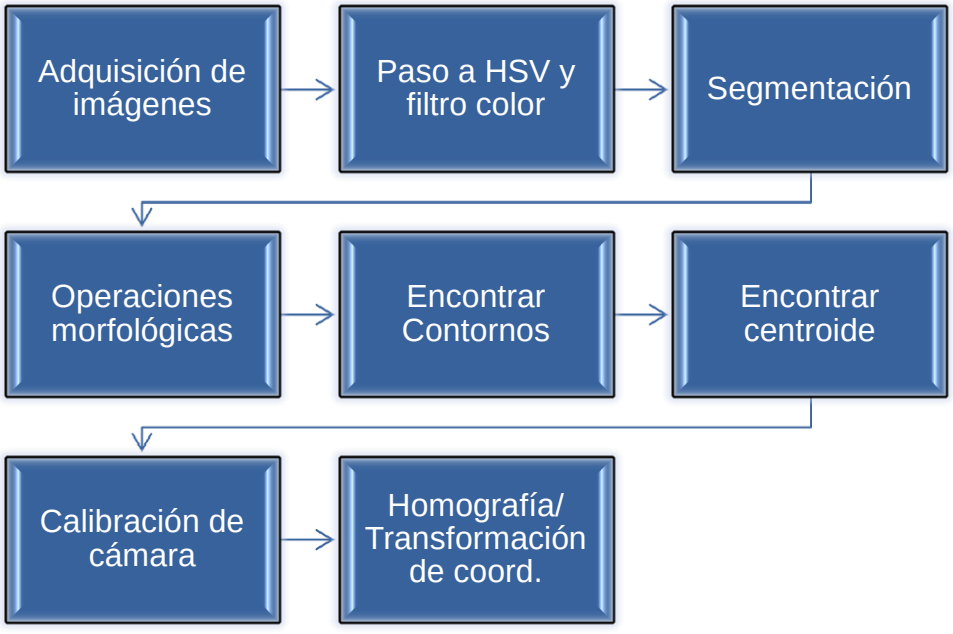
Configuración de hoja de propiedades a añadir a proyecto para utilizar librería OpenCV

6.3. DESCRIPCIÓN PROCESO VISIÓN ARTIFICIAL

La meta del programa desarrollado con la librería **OpenCV** es la de detectar los bolos que están caídos e ignorar los bolos que están aún en pie.

En este proyecto el objetivo será detectar los bolos y las posiciones en las que es posible situarlos según su rango de color.

Al trabajar con colores, será necesario identificar los rangos de colores en cada uno de los canales. A pesar de que generalmente se utiliza el sistema de color **RGB** (Red, Green, Blue), en este caso no resultará el más adecuado para la percepción de color y se considera oportuno decantarse por **HSV** (Hue, Saturation,Value), más similar a la percepción de la visión humana, que será el sistema de color que nos garantizará unos mejores resultados para realizar las detecciones. Tras esa transformación, se aplicaría una segmentación para binarizar la imagen según un umbral, que definiremos a partir de un filtrado o estudio de color y a continuación se aplicarían unas operaciones morfológicas (erosión y dilatación) para poder detectar de manera óptima los bolos y evitar ruidos e interferencias. Cuando se haya superado esta primera etapa de operaciones, se procederá a encontrar los contornos del objeto detectado y su centroide- centro de masas. Para concluir tendremos que transformar las coordenadas de dicho centroide en píxeles a coordenadas del mundo real, en las que se va a desplazar nuestro brazo robótico. Esto se realizará mediante un procedimiento de geometría proyectiva que consiste en hallar la matriz de homografía de unos puntos determinados de una imagen y a continuación calcular la transformación de perspectiva utilizando los valores obtenidos para dicha matriz.



6.4. PROCESAMIENTO DE LA IMAGEN

El procesamiento de imágenes es un método para convertir una imagen a forma digital con el fin de poder realizar una serie de operaciones sobre esta de forma que nos permita extraer ciertas características e información útil. En este apartado se procederá a explicar los procedimientos seguidos para realizar el procesamiento de imagen en el presente proyecto.

6.4.1. CONVERSIÓN A HSV Y FILTRADO POR COLOR

HSV (HUE-SATURATION-VALUE) consistirá en combinar el tono (**Hue**), saturación (**Saturation**) y la intensidad, o valor (**Value**) de la siguiente forma: H determinará el tono de color, en un rango que oscila entre 0 y 360 grados. El tono o Hue de un color será su posición en una rueda donde ,por ejemplo, rojo correspondería a 0 grados, verde a 120 y azul a 240. Si tuviéramos un color amarillo anaranjado en **RGB** eso significaría que tenemos valores de rojo y verde altos y bajo de azul. Sin embargo en una rueda de color el ángulo correspondiente sería algo menos de 60 grados. Para colores neutros como blanco, gris o negro sería 0 grados.

S es la saturación o cantidad de tonalidad de grises del color en un rango entre 0 y 1. Blanco, gris y negro tendrán todos un valor de saturación de 0 mientras que colores más puros y brillantes tendrán un valor de 1.

V es la intensidad o cantidad de brillo del color también en un rango entre 0 y 1. No hará distinción entre blanco y colores puros, siendo todos ellos de valor 1. Si el valor de **V**, por ejemplo fuera 0, el color resultante sería negro, independientemente de los valores de las otras dos componentes. De la misma forma, si **S** fuera 0 y **V** fuera 1, tendríamos blanco, sin importar el valor de **H**.

El sistema **RGB** es el utilizado por el hardware normalmente, y así sucede con la cámara web empleada en este trabajo, de manera que necesitamos convertir **RGB** a **HSV**. Existen unas fórmulas para asignar valores **HSV** a valores **RGB**:

Dados tres números para valores de **R**, **G** y **B**, cada uno entre 0 y 255 podremos definir **m** y **M** con las relaciones:

$$M = \max\{R, G, B\}$$

$$M = \min\{R, G, B\}$$

Y a continuación V y S serían definidos por las ecuaciones:

$$V = M/255$$

$$S = 1- n/M \qquad si\ M > 0$$

$$S = 0 \qquad si\ M = 0$$

Hue será definido según las ecuaciones:

$$H = \cos^{-1} [\ (R - ½G - ½\ B)/\sqrt{R^2 + G^2 + B^2 - RG - RB - GB}\ ] \qquad si\ G \geq B$$

o

$$H = 360 - \cos^{-1}[\ (R - ½G - ½B)/\sqrt{R^2 + G^2 + B^2 - RG - RB - GB}\ ]\ si\ B > G$$

En **OpenCV** se dispondrá de la función **cvCvtColor**, que nos evitará realizar estos cálculos. Esta función nos permitirá introducir una imagen definida en un sistema de color y nos devolverá otra convertida al sistema de color que le indiquemos.

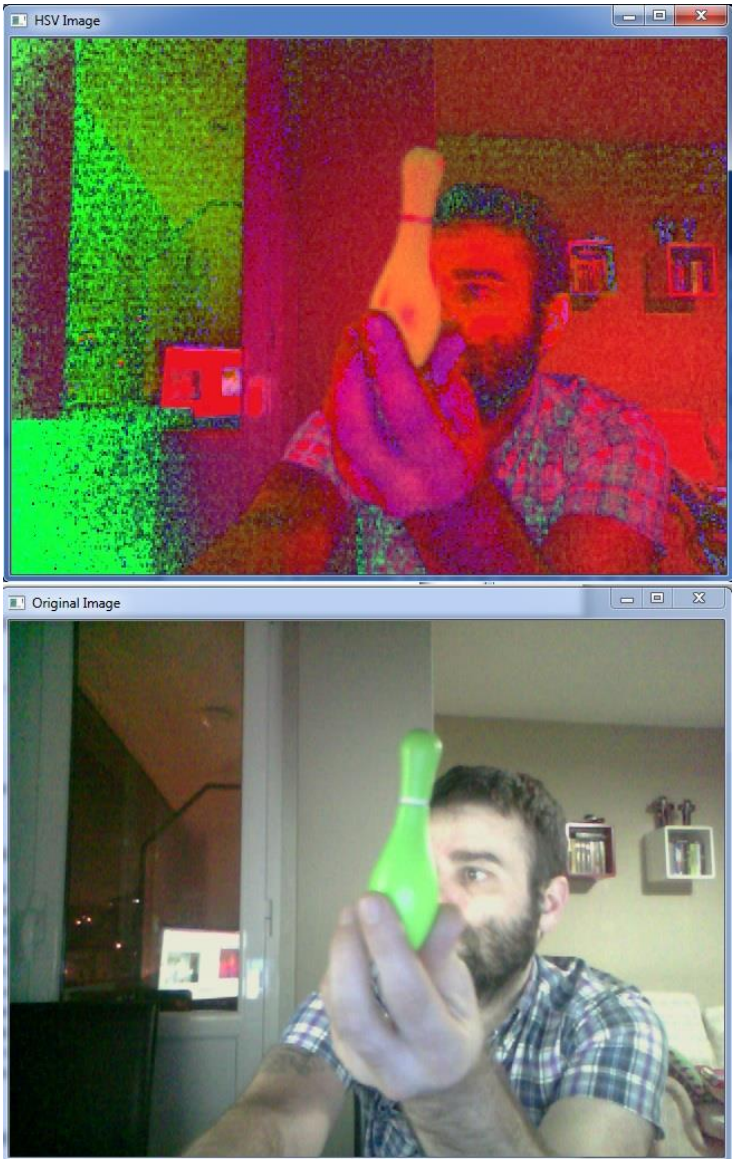
Esta función puede utilizarse con imágenes de 8, 16 o 32 bits. Es recomendable utilizar imágenes de 32 bits si vamos a realizar aplicaciones que necesiten todo el rango de colores. Si guardásemos la imagen con 8 bits no se podrían alcanzar valores mayores de 255 (rango de valores entre 0-255) y, como antes se había indicado, H varía entre 0 y 360, lo que implicaría tener información perdida. Los cambios en los valores, en caso de decantarse por los 8 bits serían los siguientes:

$$V = 255V$$

$$S = 255S$$

$$H= H/2$$

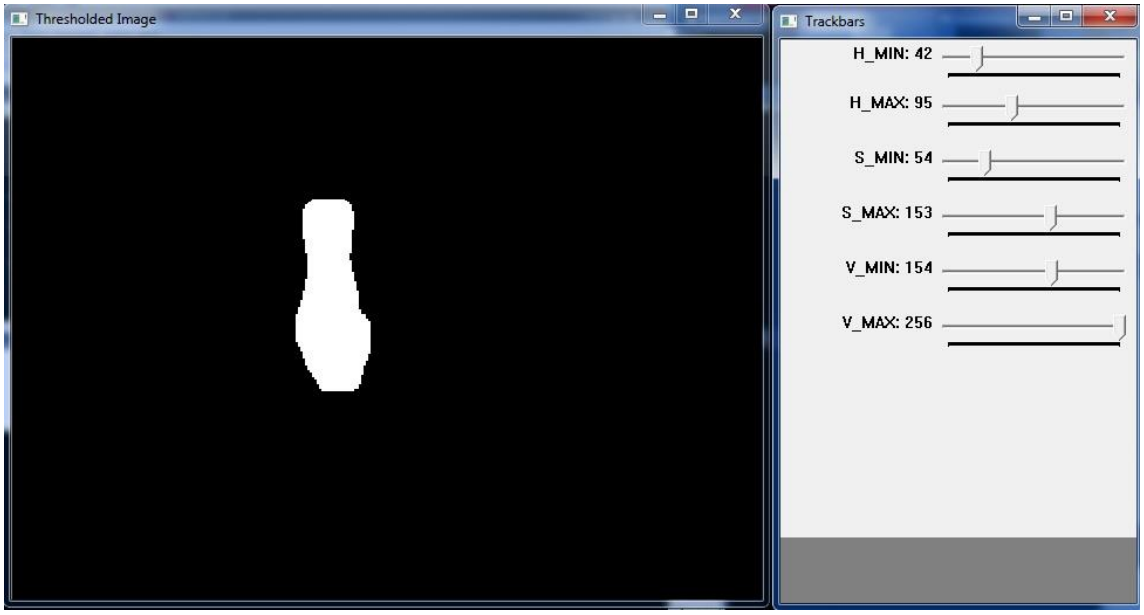
Con 32 bits no habría conversión en los valores de **HSV**.



En esta imagen se puede apreciar el efecto de aplicar la función CvtColor

6.4.2. SEGMENTACIÓN DE LA IMAGEN

A continuación se realizará una segmentación de la imagen. Esto se conseguirá introduciendo un valor umbral, que nos permitirá seleccionar píxeles en función de sus valores **HSV** y destacarlos del resto. En nuestro caso nos interesa destacar los píxeles de un bolo. Para ello deberíamos de poder encontrar el rango de valores de color que caracterizan a dicho bolo. Si aplicamos una binarización a la imagen, los píxeles del bolo destacados quedarían marcados en blanco y el resto serían negros. Más adelante, en el apartado dedicado a explicar el código implementado en **OpenCV** se comentará el método que utilizaremos para conocer los valores característicos **HSV** de los bolos y posiciones.



Tras aplicar el filtrado con los valores HSV introducidos por sliders y las operaciones morfológicas

6.4.3. OPERACIONES MORFOLÓGICAS. DILATACIÓN Y EROSIÓN

Después de la segmentación será necesario aplicar las operaciones morfológicas en nuestra imagen binaria, que consistirán en eliminar o en añadir píxeles, obteniendo como resultado otra imagen binaria. Estas operaciones se conocen como dilatación y erosión y son muy utilizadas en visión artificial.

La dilatación consistirá en añadir píxeles a los límites del objeto: Si tenemos un elemento A del espacio euclídeo y un componente estructural B, la dilatación de A por B será un conjunto de puntos x tales que B_X (que será la traslación de B cuando el origen es x) forma una intersección no vacía con A:

$$A \oplus B = \{x \,/\, B_X \cap A \neq \Phi\}$$

Otra explicación sería que la dilatación es una convolución de una imagen, que llamaremos A con un kernel (núcleo) que llamaremos B. El kernel, que podrá ser de cualquier forma o tamaño tiene un punto de anclaje definido. Este kernel a menudo suele ser tomado como un cuadrado sólido o un disco con el punto de anclaje en su centro. El kernel puede ser considerado una plantilla o máscara y su efecto para dilatación es el de un operador máximo. Cuando el kernel B se desplaza sobre la imagen A se hará una computación del valor máximo de pixel solapado por B y se reemplazará el píxel de la imagen bajo el punto de anclaje con ese valor

máximo. Esto causa que regiones iluminadas dentro de una imagen aumenten de tamaño.

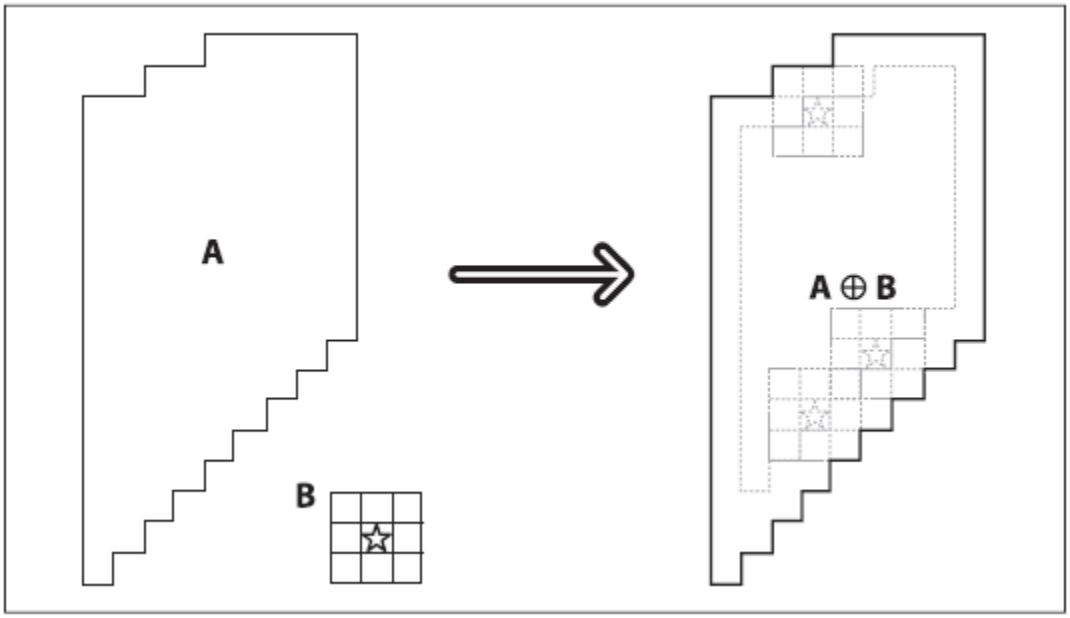


Ilustración 1. Se puede observar como el elemento B se hace pasar por la imagen A, aumentando el tamaño de esta.

La erosión será la operación inversa. Para una imagen A y un elemento estructural o kernel B de Z^2, la erosión será definida por la siguiente ecuación:

$$A \ominus B = \{x \,/ (B)x \subseteq A\}$$

De donde entendemos que la erosión de A por B será el conjunto de puntos x, tal que B trasladado por x, estará contenido en A.

La acción del operador de erosión será la equivalente a computar un mínimo local sobre el área del kernel. La erosión generará una imagen nueva a partir del original según el siguiente algoritmo: Mientras B es deslizado sobre la imagen, se computan los valores mínimos de los píxeles solapados por B y se reemplaza el píxel de la imagen bajo el punto de anclaje con ese valor mínimo.

En resumen: La dilatación aumentará la región A con lo que se conseguirá suavizar el efecto de concavidades y la erosión reducirá la región A, suavizando a su vez las protusiones. El objetivo será, por tanto, eliminar todo el ruido posible y conservar únicamente contenido significativo y también conservar la unidad de componentes en casos de que se produzca la rotura de una región.

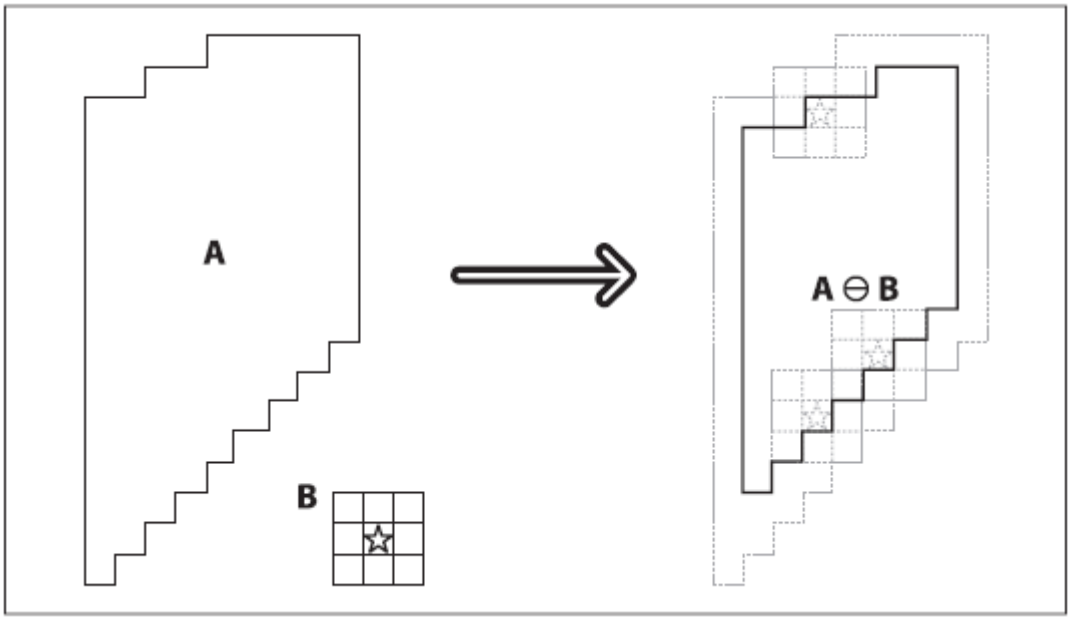


Ilustración 2. Al deslizar B sobre la imagen A se observa como se reduce la región de A

Existen funciones específicas de **OpenCV** para realizar estas operaciones morfológicas**: cvErode()** y **cvDilate().** Ambas se utilizan introduciendo una imagen de entrada, especificando el nombre de la imagen destino o de salida. Un tercer argumento sería el elemento o kernel B que se ha venido mencionando, que por defecto es NULL (kernel de 3x3 con anclaje en su centro) y un cuarto argumento será el número de iteraciones, que tienen como valor por defecto 1.

Para poder distinguir entre bolos derribados aplicaremos un filtro de tamaño. Teniendo en cuenta que la webcam estará dispuesta en una estructura que la ubicará sobre el castro, cuando un bolo haya sido derribado tendrá una superficie mayor a la de los bolos que aún están en pie. Es una forma sencilla, pero efectiva de reducir la imagen para que se muestren únicamente los bolos derribados, que será sobre los que se deba actuar. Normalmente se aplica este filtrado o restricción para evitar la detección de ruido. Los píxeles de los elementos de la imagen que tengan una superficie menor a la indicada pasan a tener un valor 0.

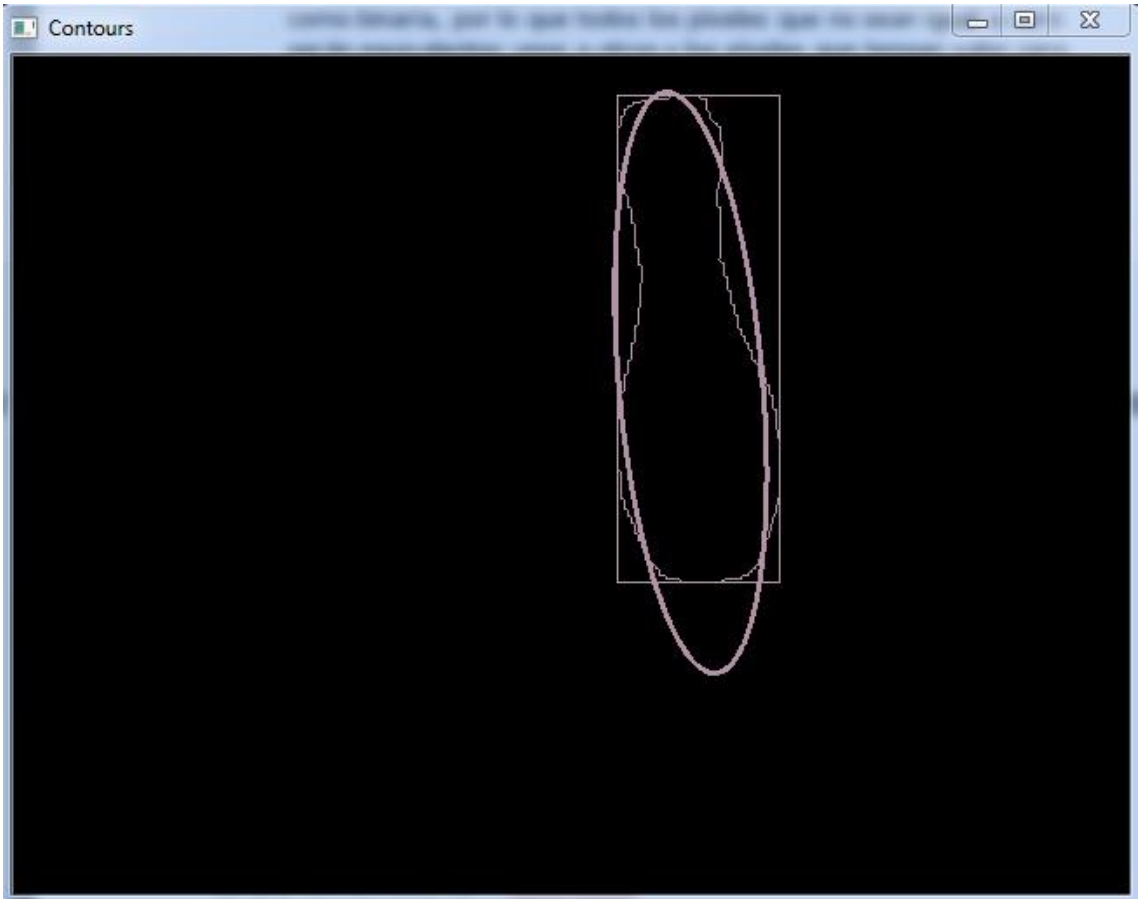
6.4.4. ENCONTRAR CONTORNOS

El siguiente paso, una vez detectado el objeto de interés (bolo) nos interesará encontrar su contorno para posteriormente poder obtener su **centroide**. Se dice contorno a una lista de puntos que representan una curva en una imagen. En **OpenCV** los contornos se representan mediante secuencias de puntos en las cuales cada entrada de dichas secuencias nos proporcionará información sobre la localización del siguiente punto en la curva.

Emplearemos la función **CvFindContours(),** que calcula y extrae contornos de imágenes binarias. Estas imágenes pueden ser como las que nosotros hemos obtenido anteriormente mediante la función **cvThreshold()**, en la que los bordes de la imagen son límites entre regiones de valores positivos y negativos

Para trabajar con la función **cvFindcontours()** debemos de introducir los siguientes argumentos:

- Imagen de entrada, que debe de ser de 8 bit y que será interpretada como binaria, por lo que todos los píxeles que no sean igual a cero serán equivalentes unos a otros y los píxeles que tengan valor cero seguirán siendo cero. Esta función modificará la imagen durante la extracción de contornos, así que crearemos una copia de la imagen binaria que conseguimos anteriormente con el umbral.
- Contornos detectados. Cada contorno se almacenará como un vector de puntos.
- Jerarquía (hierarchy), que será un vector de salida opcional con información sobre la topología de la imagen y tendrá tantos elementos como número de contornos.
- El modo, que será el tipo de recogida de contornos. En nuestro caso utilizaremos el modo **CV_RETR_CCOMP**, que recoge todos los contornos y los organiza en una jerarquía de dos niveles: En el superior estarán los límites externos de los componentes y en el segundo nivel los límites con los agujeros. Si hubiera un contorno en el interior de un agujero sería incluido también en el nivel superior.
- Método de aproximación de contornos. En nuestro caso será el **CV_CHAIN_APROX_SIMPLE**, que comprime segmentos horizontales, verticales y diagonales y deja sólo sus terminales. El contorno de un cuadrado, por ejemplo, quedaría representado con cuatro puntos (sus esquinas).
- Offset opcional, que en nuestro caso no vamos a utilizar.



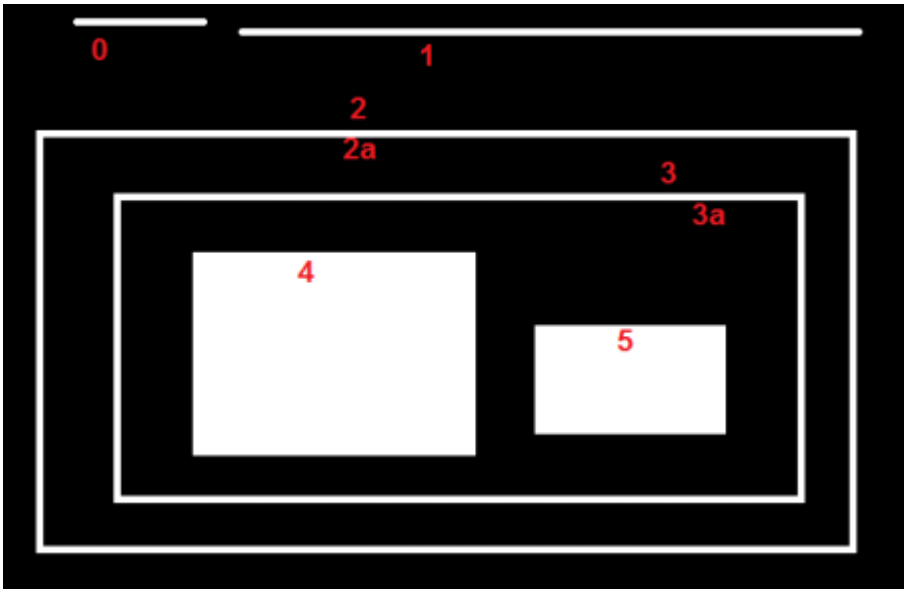
Contornos detectados a los que hemos añadido el rectángulo y elipse en el que está inscrito el objeto

Es preciso aclarar el concepto de jerarquía (**hierarchy**) para comprender el funcionamiento de la función **findContours**.

Normalmente se utiliza la función **findContours()** para detectar objetos en una imagen. En ocasiones los objetos están en localizaciones distintas, pero en otras ocasiones, algunas formas pueden estar anidadas dentro de otras. En este caso podríamos llamar padre a una forma exterior e hijo a otra interior. De esta forma, los contornos en una imagen tienen un parentesco y podemos especificar como un contorno está conectado al otro. La representación de este parentesco es llamada jerarquía (**hierarchy**).

En la siguiente imagen podremos apreciar una serie de formas numeradas de 0 a 5. 2 y 2ª distingue los contornos exteriores e interiores de la caja más grande.

Los contornos 0,1 y 2 son exteriores y podremos decir que están en una jerarquía 0, o simplemente que están en el mismo nivel de jerarquía.



Ejemplo de jerarquía de contornos

A continuación el contorno 2ª puede ser considerado como hijo del contorno 2. Por tanto estará en jerarquía 1. De manera análoga, el contorno 3 es hijo de contorno 2 y viene en el siguiente nivel de jerarquía. Los contornos 4 y 5 son hijos de del contorno 3a y vienen en el último nivel de jerarquía. Podría decirse que 4 es el primogénito del contorno 3.

De esta forma, cada contorno dispondrá de su propia información respecto al nivel de jerarquía en el que se encuentra y sus relaciones de parentesco. **OpenCV** representa esto como un array de valores: **[Siguiente, Anterior, Primer hijo, Padre]**

- **Siguiente** será el siguiente contorno en el mismo nivel de jerarquía.
- **Previo** indica el contorno previo en el mismo nivel de jerarquía.
- **Primogénito** indica el primer contorno hijo.
- **Padre** indica el índice de su contorno padre, que será lo opuesto a primogénito.

6.4.5. LOCALIZACIÓN DE CENTROIDE

Una vez encontrados los contornos nuestro interés radicará en obtener el centroide del bolo, y emplearemos para ello el método de momentos.

En visión artificial se utilizan los momentos de las imágenes para interpretación de propiedades y descripción de los objetos después de la segmentación. Estas pueden ser el área, orientación y, para éste caso en concreto, su centroide. Para una imagen, que tomaremos como una función continua 2D f(x,y), el momento general, de orden (p+q) será definido como

$$\mathbf{M_{pq}} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^{p} y^{q} f(x,y)\, dx\, dy$$

Para **p**,**q** = 0,1,2… adaptando esto a una imagen en escala de grises con la identidad de píxeles **I**(**x,y**), el momento general de una imagen $\mathbf{M_{ij}}$ será calculado de la siguiente forma:

$$\mathbf{M_{ij}} = \sum_x \sum_y x^{i} y^{j} I(x,y)$$

Los sumatorios se ejecutarán para todas las columnas y filas de la imagen; $\mathbf{p_{j,k}}$ serán valores de los píxeles, $\mathbf{x_k}$ y $\mathbf{y_j}$ son coordenadas de píxel, **m** y **n** son exponentes enteros que definen el orden del momento.

El momento central $\mathbf{U_u(m,n)}$ será el momento espacial calculado en relación al centro de masas o centroide $\mathbf{(x_0,y_0)}$:

$$\mathbf{U_U(m,n)} = \sum_j \sum_k (x_k - x_0)^m (y_j - y_0)^n P_{j,k}$$

Donde:

$$\mathbf{x_0 = M_U(1,0) / M_U(0,0)}$$

$$\mathbf{y_0 = M_U(0,1) / M_U(0,0)}$$

El área de una imagen binaria sería el momento $\mathbf{M_{00}}$ y el centroide tendría las coordenadas:

$$\{ \bar{x}, \bar{y} \} = \{ M_{10}/M_{00}, M_{01}/M_{00} \}$$

La función **Moments** de **OpenCV** calculará momentos hasta de tercer orden de una forma vectorial o rasterizada.

Los parámetros que se utilizan en esta función son:

- Array: imagen rasterizada (un canal, 8 bit o array 2D coma flotante) o un array de puntos 2D.
- Imagen binaria. Todos los píxeles con valor distinto de cero son tratados como 1.
- Momentos salida.

Con esta función, los momentos $\mathbf{m_{ji}}$ serán calculados de la siguiente forma:

$$\mathbf{m_{ji}} = \sum_{x,y} (\mathbf{array(x,y)} \cdot x^{j} \cdot y^{i})$$

Los momentos centrales **mu_{ji}** serán calculados:

$$\mathbf{Mu_{ji}} = \sum_{x,y}(\mathbf{array\ (x,y).(x-}\overline{x})^{\mathbf{j}}\ \mathbf{.\ (y-}\ \overline{y})^{\mathbf{i}})$$

Donde **$(\overline{x},\overline{y})$** es el centro de masas.

Los momentos de un contorno se definen de la misma forma, pero son calculados utilizando la fórmula de **Green**, que da una relación entre una integral de línea alrededor de una curva cerrada simple **C** y una integral doble sobre una región plana **D** limitada por **C**. Si **L** y **M** son funciones de **(x,y)** definidas en una región que contenga a **D** y que tengan derivadas parciales contínuas en dicha región, entonces:

$$\oint_C(\mathbf{L\ dx + M\ dy}) = \iint_D(\frac{\partial M}{\partial x} - \frac{\partial L}{\partial y})\ \mathbf{dxdy}$$

En geometría plana se utiliza esta fórmula para determinar el área y centroide de figuras planas integrando alrededor del perímetro de la curva cerrada.

```
Moments moment = moments((cv::Mat)contours[index]);
double area = moment.m00;
```

```
x = moment.m10/area;
y = moment.m01/area;
```

Utilizando el método de momentos también podremos obtener el ángulo de orientación del eje principal del objeto, calculando este mediante la siguiente expresión:

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{b}{a-c}\right)$$

Teniendo en cuenta que los valores de a,b y c serán los siguientes:

$$a = \frac{M_{20}}{M_{00}} - x_c^2$$

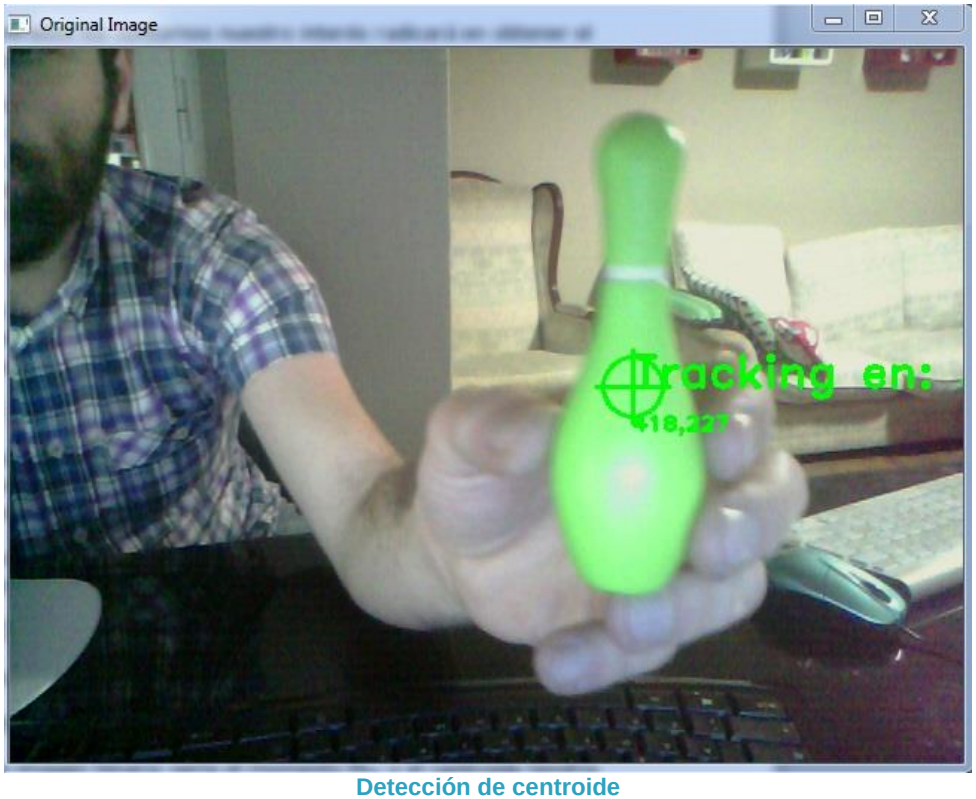
$$b = 2\left(\frac{M_{11}}{M_{00}} - x_c y_c\right)$$

$$c = \frac{M_{02}}{M_{00}} - y_c^2$$

La expresión que hemos utilizado para el código será, por tanto, la siguiente:

```
float angle = atan2((float)(-2)*moment.mu11, (moment.mu20 - moment.mu02))/2;

angle = (( angle/PI) * 180);
```

En la última línea hemos convertido el ángulo obtenido a grados.



Detección de centroide

6.4.6. ALTERNATIVA PARA SEGUIMIENTO DE OBJETOS

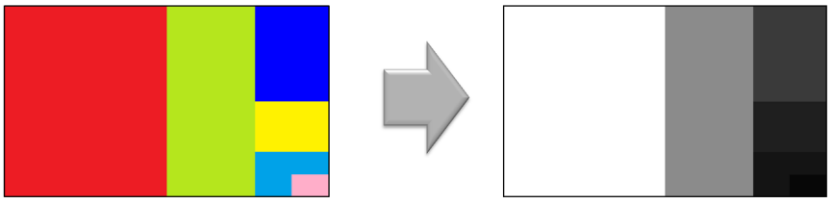
Una alternativa que ha sido contemplada es el algoritmo **CamShift.** Este algoritmo se utiliza frecuentemente para detección de objetos, especialmente caras, en **OpenCV**. Está basado en el algoritmo **MeanShift** y su función es aplicar el mencionado **MeanShift** en cada frame de una captura de video y recoger los resultados. **CamShift** extraerá el centro de gravedad, orientación y área de la imagen situada en la región de interés (**ROI**).

Las partes que componen la funcionalidad de **CamShift** podríamos enumerarlas como:

1. **Back projection** : Es un método que, por medio del histograma de una imagen, muestra las probabilidades de que los colores aparezcan en cada píxel. Con **OpenCv** transformaríamos el espacio de la imagen a un espacio **HSV** con la función **CvtColor**, a continuación apartaríamos el canal **Hue**, como si se tratase de una imagen en escala de grises, obtendríamos su histograma (**calcHist**) y lo normalizaríamos (**normalize**). Finalmente procederíamos a utilizar la función **calcBackProject()** . Esta función básicamente lo que hace es calcular el peso de cada color en la imagen mediante el histograma y cambiar el valor de cada pixel por el peso de su color en la imagen total. Si por ejemplo el color de un pixel es amarillo y el peso de dicho color en la imagen es del 20%, o sea, que existen un 20% de pixeles con ese color en la imagen, cambiaríamos el valor de este pixel de amarillo a 0,2. Haríamos lo propio con el resto de pixeles de la imagen.

   En la siguiente imagen se explica este procedimiento:

Color	Occurrence Counts	Occurrence Probability (%)	Re-defined Gray Level Color (0~255)
PURPLE	3	3%	15
GREEN	25	25%	127
YELLOW	7	7%	36
BLUE	13	13%	66
PINK	1	1%	5
RED	50	50%	255



   Los colores con menos probabilidades de aparecer en el siguiente frame serán mostrados en negro mientras que los de mayor probabilidad serán resaltados en blanco.

2. **MeanShift**: Es un algoritmo que encuentra medias (modos o picos) en un conjunto de muestras de datos, representando una función de probabilidad de densidad.

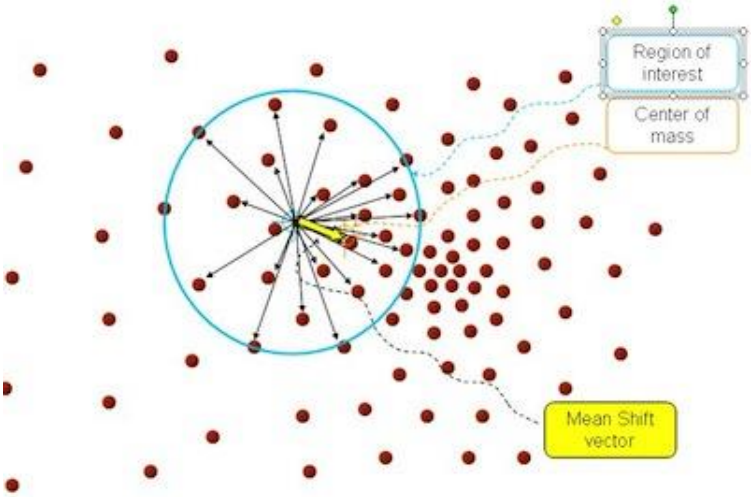
   El procedimiento que sigue para localizar la región de máxima densidad es iterativo. Se empieza con una estimación inicial x y se obtiene una función Kernel que determinará el peso de los puntos cercanos para la estimación de la media.

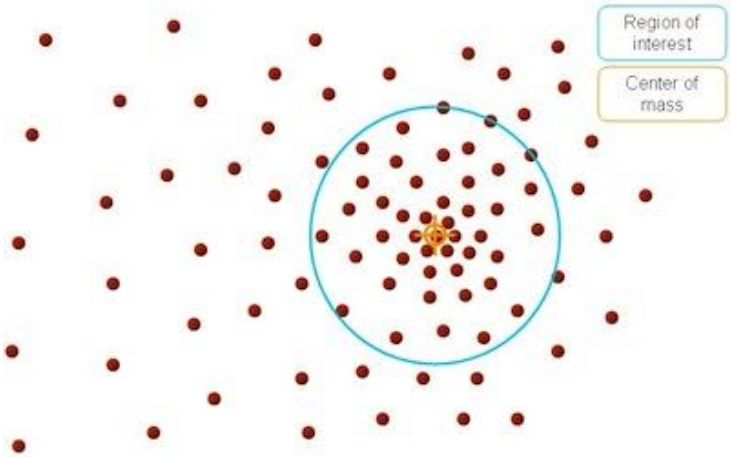
La media ponderada de la densidad en la sección determinada por **K** es:

donde **N(x)** es un conjunto de putos para los que **K(x)** no son cero.

El algoritmo **mean-shift** desplaza **x** a **m(x)** y repite la estimación hasta que converge con **m(x)**.

En la imagen se puede apreciar un conjunto de puntos en el espacio bidimensional y una región delimitada por una circunferencia con centro en **C** y radio **r** como kernel desplazando iterativamente este kernel a regiones con densidad mayor hasta que se produce la convergencia. Cada desplazamiento se define por un vector **mean shift**. Este vector siempre apuntará en la dirección de mayor incremento de densidad y en cada iteración el kernel (núcleo) se irá aproximando hasta el centroide de la media de puntos.





En **OpenCV**, la función **meanShift()** se configura de la siguiente forma:

```
cvMeanShift( const void* imgProb, CvRect windowIn,
CvTermCriteria criteria);
```

Donde **imgProb** es la distribución de probabilidad de objetos 2D, resultado de la back projection del primer punto, **windowIn** es un rectángulo que representa el tamaño inicial de la ventana, **TermCriteria** es el criterio de terminación y se debe configurar antes de la siguiente forma:

```
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )
```

En este caso serían 10 iteraciones o un desplazamiento de al menos un punto.

3. Para realizar el seguimiento o tracking se implementa la función **camShift** :

```
RotatedRect CamShift(InputArray probImage, Rect&window, TermCriteria criteria)
```

Donde los parámetros serían:

- **probImage** – Back projection del histograma del objeto.
- **Window** – Ventana de búsqueda inicial.
- **Criteria** – Criterio de terminación para el **meanShift**.

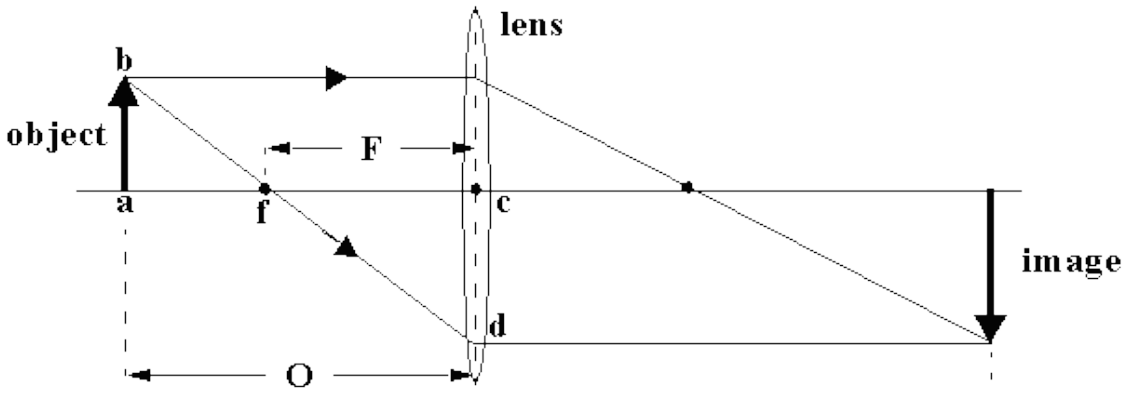
El funcionamiento de este algoritmo implica la detección del centro del objeto utilizando **meanShift** y después ajustar el tamaño de la ventana y encontrando la rotación adecuada. La función devolverá la estructura rectángular rotada que incluye la posición del objeto, tamaño y orientación.

6.4.7. CALIBRADO DE CÁMARA

Tras haber obtenido las coordenadas del centroide, el siguiente paso sería poder convertir estas coordenadas en píxeles a coordenadas del mundo real. Para poder realizar esta operación será fundamental realizar primero un calibrado de la cámara. Una vez se haya realizado este calibrado, será posible proyectar puntos en la imagen al mundo físico y a la inversa. Esto es: Dada una localización en píxeles de un objeto de interés (en nuestro caso un bolo derribado), podemos calcular dónde se encuentra respecto a nuestro sistema de referencia espacial 3D real.

Antes de explicar el proceso que se realiza en **openCV** para el calibrado de la cámara sería conveniente aclarar una serie de conceptos referentes al campo de la geometría proyectiva.

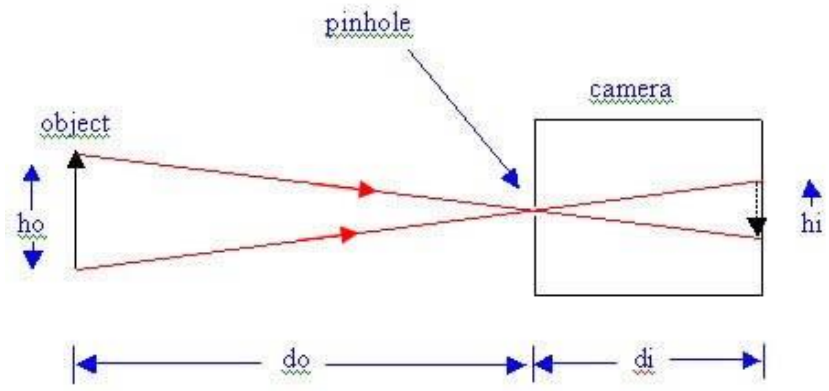
La geometría proyectiva es la herramienta que utilizamos para describir y caracterizar, en términos matemáticos, el proceso de formación de la imagen. Fundamentalmente, este proceso sigue siendo el mismo que el utilizado desde los principios de la fotografía. La Luz entrante de una escena observada es capturada por una cámara a través de una apertura frontal los rayos de luz capturados impactan en un plano de imagen (o sensor de imagen) localizado en la parte trasera de la cámara. Se utilizan además unas lentes para concentrar los rayos provenientes de los distintos elementos de la escena.



Esquema de una lente

O es la distancia desde las lentes al objeto observado, d será la distancia de la lente al plano de imagen y **F** será la longitud focal de las lentes.

En visión artificial se simplifica este modelo: Primero podemos eliminar el efecto de las lentes considerando una cámara con una apertura infinitesimal, ya que teóricamente esto no alteraría la imagen. Sólo se tendría en cuenta el rayo central. Después tendríamos en cuenta que la mayor parte del tiempo tendremos que **O > d** y podríamos dar por hecho que el plano imagen está localizado a la distancia focal. Por último, de la geometría del sistema observaremos que la imagen en el plano está invertida. Podremos obtener una imagen idéntica, aunque colocada en la misma orientación si ponemos el plano imagen delante de las lentes. Esto no es factible físicamente, aunque desde un punto de vista matemático que poniendo el plano detrás de las lentes. Esta simplificación del sistema es lo que se conoce como el modelo “Pin-hole camera” (cámara estenopeica).



Esquema de cámara tipo pin-hole

Interpretando esta ilustración y mediante la aplicación del teorema de semejanza de triángulos, podríamos obtener la ecuación proyectiva básica:

$$hi = f \cdot h_o / d_o$$

El tamaño **hi** de la imagen de un objeto sería inversamente proporcional a su distancia do de la cámara. Esta relación permite que la posición de la imagen de un punto de una escena 3D pueda ser predecida en el plano imagen de una cámara.

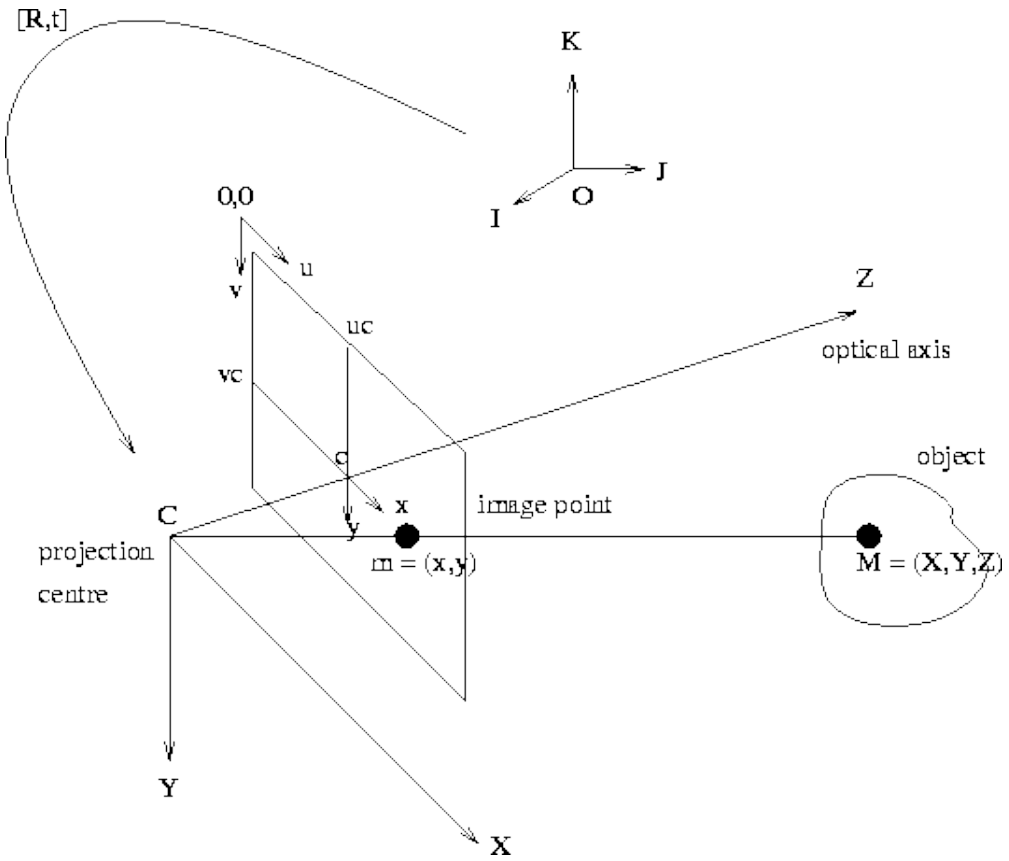
Si tuviéramos un sistema de coordenadas en tres dimensiones cuyo origen está en el centro de proyección y cuyo eje **Z** está sobre el eje óptico, tal como se muestra en la imagen de abajo, lo llamaríamos sistema de coordenadas estándar de la cámara. Un punto **M** en un objeto con coordenadas **(X,Y,Z)** será reflejado en algún punto **m = (x,y)** en el plano imagen. Estas coordenadas serán con respecto a un sistema de coordenadas cuyo origen este en la intersección del eje óptico y el plano imagen y cuyos ejes **x** e **y** son paralelos a los **X, Y**. La relación entre los dos sistemas de coordenadas **(c, x, y)** y **(C, X, Y)** estará dada por:

$$X = \frac{X.f}{Z} \quad \text{e} \quad y = \frac{Y.f}{Z} \qquad (1)$$

Esto puede ser escrito en coordenadas homogéneas como:

$$\begin{bmatrix} sx \\ sy \\ s \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Donde s ≠ 0 es un factor de escala.



Proyección de imágenes en el plano de la cámara

Las coordenadas reales en píxeles **(u,v)** son definidas con respecto a un origen situado en la esquina superior izquierda del plano imagen y deben de cumplir:

$$\mathbf{u = U_c + \frac{x}{ancho\ de\ pixel}} \quad \text{y} \quad \mathbf{v = v_c + \frac{y}{alto\ de\ pixel}} \qquad (2)$$

Podemos expresar la transformación de coordenadas en tres dimensiones a coordenadas en píxeles utilizando una matriz de 3 x 4. Esto

se consigue sustituyendo la expresión de x de la ecuación (1) en (2) y multiplicando todo por Z, para obtener:

$$\mathbf{ZU} = \mathbf{ZU_c} + \frac{\boldsymbol{Xf}}{\boldsymbol{ancho\ de\ pixel}}$$

$$\mathbf{Zv} = \mathbf{Zv_c} + \frac{\boldsymbol{Yf}}{\boldsymbol{alto\ de\ pixel}}$$

De otra forma,

$$\begin{bmatrix} \boldsymbol{su} \\ \boldsymbol{sv} \\ \boldsymbol{s} \end{bmatrix} = \begin{bmatrix} \frac{\boldsymbol{f}}{\boldsymbol{ancho\ de\ pixel}} & \mathbf{0} & \boldsymbol{Uc} & \mathbf{0} \\ \mathbf{0} & \frac{\boldsymbol{f}}{\boldsymbol{alto\ de\ pixel}} & \boldsymbol{Vc} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{X} \\ \boldsymbol{Y} \\ \boldsymbol{Z} \\ \mathbf{1} \end{bmatrix}$$

Donde el factor de escala s tiene valor **Z**. En forma abreviada escribiríamos esto como:

$$\tilde{\mathbf{u}} = \mathbf{P}.\ \tilde{\mathbf{M}}$$

Donde $\tilde{\mathbf{u}}$ representa el vector homogéneo de coordenadas en píxeles de la imagen, **P** es la matriz de proyección perspectiva y $\tilde{\boldsymbol{M}}$ es el vector homogéneo de coordenadas reales. Una cámara podría ser entonces considerada como un sistema que lleva a cabo una transformación proyectiva lineal del espacio $\mathbf{P^3}$ al plano $\mathbf{P^2}$.

Hay cinco parámetros de cámara: longitud focal **f**, ancho de píxel, alto de pixel, parámetro $\mathbf{u_c}$ que es la coordenada en el centro óptico del píxel **u** y el parámetro $\mathbf{v_c}$ que es la coordenada en el centro óptico del píxel **v**. Sólo cuatro parámetros pueden resolverse, porque hay un factor de escala arbitrario incluido en **f** y en el tamaño de pixel. Podremos decir:

$$\boldsymbol{\alpha_u} = \frac{\boldsymbol{f}}{\boldsymbol{ancho\ de\ pixel}}$$

$$\boldsymbol{\alpha v} = \frac{\boldsymbol{f}}{\boldsymbol{alto\ de\ pixel}}$$

Los parámetros $\boldsymbol{\alpha_u}$, $\boldsymbol{\alpha_v}$, $\mathbf{u_c}$ y $\mathbf{v_c}$ no dependen de la posición y orientación de la cámara en el espacio y son los conocidos como **parámetros intrínsecos**.

$$\tilde{u}' = P.\ K.\ \tilde{M}$$

Donde K es una matriz homogénea de transformación.

$$K = \begin{bmatrix} R & t \\ 0_3^T & 1 \end{bmatrix}$$

La **R** de la esquina de arriba es una matriz de rotación de 3x3 y guarda información sobre la orientación de la cámara respecto al origen; la t de la esquina superior derecha es un vector homogéneo t que guarda el desplazamiento de la cámara desde el origen. La matriz **K** tendrá seis grados de libertad, tres para orientación y tres para traslación de la cámara. Estos parámetros son los conocidos como **parámetros extrínsecos** de la cámara.

La matriz de cámara **P** de 3x4 y la transformada homogénea 4x4 **K** se combinan para formar una matriz 3x4 llamada **C**, llamada matriz de calibración de cámara. Podremos escribir la forma general de **C** en función de los parámetros intrínsecos y extrínsecos.

$$C = \begin{bmatrix} \alpha_u r_1 + u_c r_3 & \alpha_u t_x + u_c t_z \\ \alpha_{uv} r_2 + v_c r_3 & \alpha_v t_y + v_c t_z \\ r_3 & t_z \end{bmatrix}$$

Donde los vectores $\mathbf{r_1}$, $\mathbf{r_2}$, y $\mathbf{r_3}$ son los vectores de fila de la matriz **R** y $\mathbf{t = (t_x, t_y, t_z)}$.

Para realizar el calibrado de la cámara se utilizará la función **CalibrateCamera2()**. El método de calibración será recoger con la cámara una estructura conocida que tenga muchos puntos individuales e identificables. Observando esta estructura desde varios ángulos o perspectivas será posible calcular la localización relativa y la orientación de la cámara en el momento de cada imagen.

El objeto utilizado habitualmente para la calibración en **OpenCV** es un tablero de ajedrez. Dada la imagen de un tablero de ajedrez, podremos utilizar la función de **OpenCV FindChessboardCorners()** para localizar las esquinas de dicho tablero. Los argumentos de dicha función serán:

- **Imagen**. Sería la que contenga el tablero. Debe ser en escala de grises de 8 bit.
- **pattern_size**, indica cuantas esquinas hay en cada fila y columna del tablero. Esto es referido al número de esquinas interiores. Para un tablero normal el valor correcto sería CvSize(7,7)
- **corners**, es un puntero a un array donde las localizaciones pueden ser guardadas. Este array debe de ser creado previamente y debe de

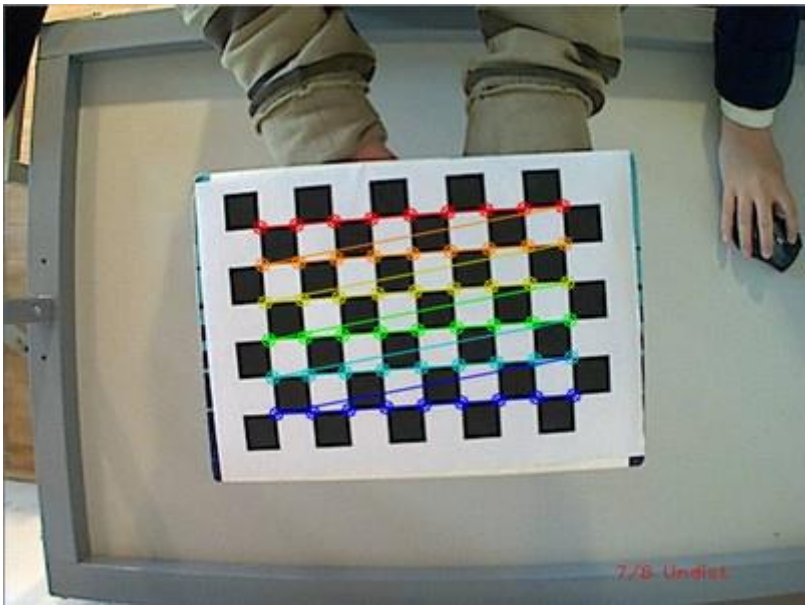
ser suficientemente grande como para almacenar todas las esquinas del tablero, o sea, 49.

- **corner_count** es un argumento opcional; si es distinto a NULL será un puntero a un entero, donde el número de esquinas encontrado puede ser guardado. Si la función consigue recoger todas las esquinas, entonces el valor que devuelve no será un cero. Si no lo consigue el valor devuelto será 0.
- **flags** puede ser utilizado para implementar uno o más pasos de filtración para ayudar a localizar las esquinas. En nuestro caso lo dejaremos por defecto, que tendrá el efecto de realizar primero un umbralizado de la imagen basado en el brillo promedio.

Las esquinas devueltas por esta función son sólo una aproximación. En la práctica esto significaría que las localizaciones sólo serían correctas dentro de los límites de nuestro dispositivo de captura de imagen , o sea, que tendrían una precisión de un píxel. Emplearemos una función distinta para calcular la localización exacta de las esquinas para poder tener una precisión de un subpíxel. Esta función será **FindCornerSubPix()**.

Se dibujarán las esquinas encontradas en una imagen para poder comprobar si las proyecciones obtenidas se corresponden con las observadas en la imagen. Se utilizará para ello la función **DrawChessboardCorners()**, que dibujará las esquinas localizadas por la función **FindChessBoardCorners()** en una imagen que especifiquemos, que será normalmente la misma que habíamos utilizado para calcular las esquinas. Si no se encontraron todas las esquinas, se marcarán las que falten con círculos rojos. Si se localizó todo el patrón, las esquinas se representarán de distintos colores, variando por filas y se conectarán por líneas, que representan el orden de identificación.

Se ha utilizado para este proyecto el código de calibración descrito en el libro learning **OpenCV** de **Bradski**, obteniendo como resultado dos matrices de parámetros intrínsecos y de distorsión.



Resultado de proceso de calibración en tablero de ajedrez

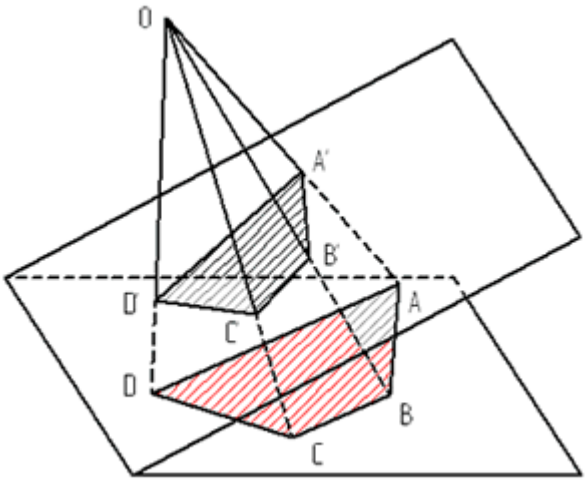
```
<?xml version="1.0"?>
<opencv_storage>
<Distorsion type_id="opencv-matrix">
  <rows>5</rows>
  <cols>1</cols>
  <dt>f</dt>
  <data>
    2.59289201e-002 -2.86308289e-001 2.86477129e-003 4.49233130e-003
    3.71410251e-001</data></Distorsion>
</opencv_storage>
```

```
<?xml version="1.0"?>
<opencv_storage>
<Intrinsics type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>f</dt>
  <data>
    7.57980957e+002 0. 3.25747406e+002 0. 7.59475830e+002
    2.74468842e+002 0. 0. 1.</data></Intrinsics>
</opencv_storage>
```

Matrices obtenidas: De distorsión arriba y parámetros intrínsecos abajo

6.4.8. TRANSFORMACIÓN DE COORDENADAS. HOMOGRAFÍA

Entendemos como homografía a la transformación proyectiva que determina una correspondencia entre dos figuras geométricas planas, de forma que a cada uno de los puntos y las rectas de una de ellas le corresponden, respectivamente, un punto y una recta de la otra.



Homografía espacial

Los puntos contenidos en un plano pasan por una transformación de perspectiva cuando son vistos a través de la cámara. Los parámetros para esta transformación están contenidos en una matriz de homografía H.

En visión artificial una homografía planar es una proyección de un plano a otro. De esta forma, el mapeado de puntos en una superficie plana de 2 dimensiones al receptor de nuestra cámara es un ejemplo de homografía planar. Si tenemos coordenadas homogéneas para expresar un punto concreto **P** y el punto **p** en el receptor de imágenes al cual **P** es mapeado podríamos expresar la acción de la homografía como:

$$P = s.H.P$$

Donde s será un factor de escala arbitrario.

La función **FindHomography()** cogerá una lista de correspondencias y nos devolverá la matriz de homografía que describa mejor esas correspondencias. Necesitamos un mínimo de 4 puntos para obtener **H**, pero se pueden utilizar más, y añadir más puntos es beneficioso porque es inevitable que existan ruido y otras inconsistencias cuyo efecto sería recomendable minimizar.

cvFindHomography (src_points, dst_points, homography), donde:

los **arrays src_points** y **dst_points** pueden ser matrices de **N** por 2 o de **N** por 3. En caso de que sean del primer tipo los puntos son coordenadas en píxeles y en caso de que sean del segundo los puntos son coordenadas homogéneas. El argumento **homography** es una matriz 3 por 3 que será rellenada por la función de manera que el error de la retro-proyección sea minimizado.

Una vez obtenida la matriz de homografía, si queremos saber la correspondencia de unas coordenadas en píxeles con coordenadas en el espacio euclídeo utilizaríamos la función **perspectiveTransform()**, que realiza la transformación de vectores de la matriz de transformación.

Se llama a la función de la siguiente forma: **perspectiveTransform(src,dst,m)** , Donde:

src = Array de entrada float de dos o tres canals; Cada elemento es un vector 2D o 3D a transformar..

dst = Array de salida del mismo tamaño y tipo que src.

m = Matriz de transformación de 3x3 o 4x4.

Utilizaremos la matriz **H** de homografía encontrada con la función anterior como matriz de transformación en esta función para hallar la correspondencia que nos interesa.

El procedimiento llevado a cabo para obtener la matriz de homografía ha sido el siguiente:

Se ha creado una plantilla con 9 puntos de color rojo formando una cuadrícula y hemos utilizado un programa modificado a partir del desarrollado para la detección de bolos, de forma que hemos podido extraer la localización en coordenadas de píxeles de dichos 9 puntos. Este código primero realiza una captura con la webcam y posteriormente aplicamos las mismas operaciones que en detección, hasta que localizamos sus centroides y conseguimos las coordenadas de dichos puntos.

A continuación se ha elegido otro punto que servirá de origen de coordenadas reales. Obtendremos de igual forma sus coordenadas en píxeles, que posteriormente utilizaremos para realizar una comprobación.

Creamos un vector de puntos en el que almacenaremos los valores de las coordenadas en píxeles de estos centroides:

```
std::vector<cv::Point2f> coordImagen;
 coordImagen.push_back(cv::Point2f(280,52));
 coordImagen.push_back(cv::Point2f(377,54));
 coordImagen.push_back(cv::Point2f(474,54));
 coordImagen.push_back(cv::Point2f(474,148));
```

….

Guardaremos una captura de dichas coordenadas representadas sobre la imagen original de la cuadrícula

Se tomarán ahora los valores de las coordenadas en el mundo real de los puntos de la cuadrícula teniendo como referencia el nuevo origen de coordenadas que hemos introducido anteriormente. Con un instrumento de medida realizamos la medidas medidas correspondientes sobre la plantilla en mm y las anotamos.

El siguiente punto consiste en crear un vector de puntos análogo al de coordenadas en píxeles, en el que introduciremos las coordenadas del mundo real que acabamos de obtener respecto al origen de coordenadas de nuestra elección:

```
std::vector<cv::Point2f> coordMundo;
    coordMundo.push_back(cv::Point2f(20,102));
    coordMundo.push_back(cv::Point2f(20,140));
    coordMundo.push_back(cv::Point2f(20,180));
    coordMundo.push_back(cv::Point2f(65,180));
…
```

Una vez que tengamos estos dos vectores, podremos calcular la matriz de homografía mediante la función **findHomography**, antes explicada:

```
Mat imagenaMundo = findHomography(coordImagen,coordMundo)
```

La inversa de esta matriz imagenaMundo que acabamos de obtener será la que utilizaremos para calcular los valores en píxeles teniendo a nuestra disposición las coordenadas reales de los puntos que nos interesen. Será la que utilicemos para comprobar la precisión de nuestro mapeado con el origen de coordenadas. La llamaremos mundoaImagen.

```
Mat mundoaImagen = imagenaMundo.inv();
```

Para ello se hace otro vector (LocMundo) en el que se meten valores reales y otro (LocImagen), que será rellenado con transformaciones de los puntos contenidos en el vector LocMundo mediante la matriz mundoaImagen.

```
vector<cv::Point2f> LocMundo;
vector<cv::Point2f> LocImagen;

Point2f = transformPoint(LocMundo[i], mundoaImagen);
```

Dibujaremos los puntos transformados y pasaremos al origen:

```
cv::Point2f origenCoord = cv::Point2f(50,10);          //50,10 serán sus coordenadas en píxeles, o sea, en el sistema de referencia de la imagen.
```

se dibujará este punto en la imagen y posteriormente se hará la siguiente operación:

```
cout<< transformPoint(origenCoord, imagenaMundo) << endl;
```

con esto lo que haremos será comprobar en la ventana de la consola de comandos de windows los valores que corresponderían al mundo real de las coordenadas del origen. Deberían de ser (0,0) idealmente pero, debido a las distorsiones de lente, será difícil llegar a esa precisión.

En nuestro caso, el valor obtenido ha sido el siguiente:

```
[2.455201, 1.732573]
```

En la imagen inferior se puede apreciar la leve desviación entre los puntos representados anteriormente en píxeles y los que hemos dibujado después de aplicarles la transformación. Rodeado por un círculo blanco, en el extremo superior izquierdo, se puede apreciar la localización del orígen de coordenadas.

Para guardar la matriz de homografía que hemos calculado y poder utilizarla posteriormente utilizaremos la clase **FileStorage** para crear un objeto de escritura con el que crearemos un archivo en formato .yml y en el que guardaremos la matriz imagenaMundo de la siguiente forma:

```
FileStorage storage("test.yml",FileStorage::WRITE);
storage <<"img" <<imagenaMundo;
storage.release();
```

Después de esta última operación, dispondremos del archivo .yml, conteniendo la matriz de homografía, en la carpeta de nuestro proyecto.

```
%YAML:1.0
img: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [ -7.6150638871978135e-002, 1.6965931456594738e-004,
    3.6298508626166118e+001, -2.8715499335918565e-001,
    -9.6008838780723471e-002, 1.7065580298107653e+002,
    -2.0630776336242806e-003, -1.6489372138301190e-005, 1. ]
```

Matriz de homografía obtenida

Para su posterior utilización se debe tener en cuenta que la función **findHomography** devuelve una matriz 3x3 usando coordenadas homogéneas. Por tanto se procederá a transformar los puntos cartesianos a coordenadas homogéneas para poder aplicar la matriz **H** de homografía. Después será necesario volver a convertir el punto de nuevo a coordenadas cartesianas.

Si el punto del que estamos hablando fuera por ejemplo un punto **P (x,y)**, su transformada homogénea sería **P'(x,y,1)**. La transformación perspectiva que será realizada con la matriz homogénea sería:

$$\mathbf{P' = P * H =} \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} h_0 * x + h_1 * y + h_2 \\ h_3 * x + h_4 * y + h_5 \\ h_6 * x + h_7 * y + h_8 \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

convertir **P'** (homogéneas) a **P** (cartesianas) :

$$(t_x, \ t_y, \ t_z) \ \mathbf{=>} \left({}^{t_x}/_{t_z} \ , \ {}^{t_y}/_{t_z} \right)$$

Para poder realizar esta operación se ha implementado esta función:

```
Point2f transformarPunto(Point2f actual, Mat transformacion)

Point2f puntotransformado;

puntotransformado.x = actual.x * transformacion.at<double>(0,0)
+     actual.y * transformacion.at<double>(0,1) +
transformacion.at<double>(0,2);

puntotransformado.y = actual.x * transformacion.at<double>(1,0)
+ actual.y * transformacion.at<double>(1,1) +
transformacion.at<double>(1,2);


    float z = actual.x * transformacion.at<double>(2,0) +
actual.y * transformacion.at<double>(2,1) +
transformacion.at<double>(2,2);

    puntotransformado.x /= z;
    puntotransformado.y /= z;

  return puntotransformado;
```

Se llama a la función con dos argumentos: El punto a transformar (actual) y la matriz de homografía con la que se transformará (transformación). La función devolverá el punto transformado en formato **Point2f**.

6.4.9. FUNCIONAMIENTO DEL PROGRAMA DE DETECCIÓN

En este apartado se expondrá brevemente el funcionamiento del programa realizado para la detección en **Visual Studio**, que posteriormente

será adaptado al entorno **Qt** para poder realizar una interfaz de usuario. Podríamos decir que es la versión alpha del programa, que ya contiene prácticamente la totalidad de la funcionalidad del programa, pero que aún no está plenamente desarrollado. El funcionamiento será el siguiente:

Al iniciar el programa, con el objetivo de seleccionar un valor umbral que nos permita una selección de píxeles de la imagen que recibimos a través de la webcam introduciremos una modalidad de calibrado que activamos o desactivamos en el código poniéndole un valor true. Si éste modo de calibración ese encuentra activo, crearemos una ventana con deslizadores (trackbars) con los que modificaremos los valores **HSV** para encontrar un rango con el que podamos identificar objetos por su color. A continuación se crea un objeto de **VideoCapture**, que es la clase de la librería de **OpenCV** que nos permitirá la adquisición de imágenes a través de nuestra cam. Inicializamos el proceso de captura especificando la cámara que vamos a utilizar (0) y se especifican los parámetros de la captura (ancho y alto, básicamente). Es necesario iniciar un bucle que tenga una condición que se cumpla siempre (While (1), en nuestro caso) para poder adquirir las imágenes de la webcam en tiempo real. Se declara otra matriz que será la que reciba las imágenes de la cámara web, la cual será transformada con la función **CvtColo**r de color tipo **RGB** (o BGR) a **HSV** y esta imagen transformada se alojará en otra matriz creada que llevará el nombre **HSV**. Se creará también una matriz **Threshold** o umbral en la que se aplicarán las restricciones de rango **HSV** que hayamos introducido. Si por ejemplo queremos crear un umbral (**threshold**) para un bolo de color verde colocaríamos éste frente a la webcam y, atendiendo a la imagen mostrada en la ventana de umbral se trabajaría con los deslizadores para introducir valores **HSV**, mostrándose los píxeles que vamos destacando en la ventana de umbral y quedando el resto descartados. Cuando en la ventana de umbral sólo tengamos la silueta del bolo azul sabremos que esos valores de **HSV** son los que deseamos para su posterior detección.

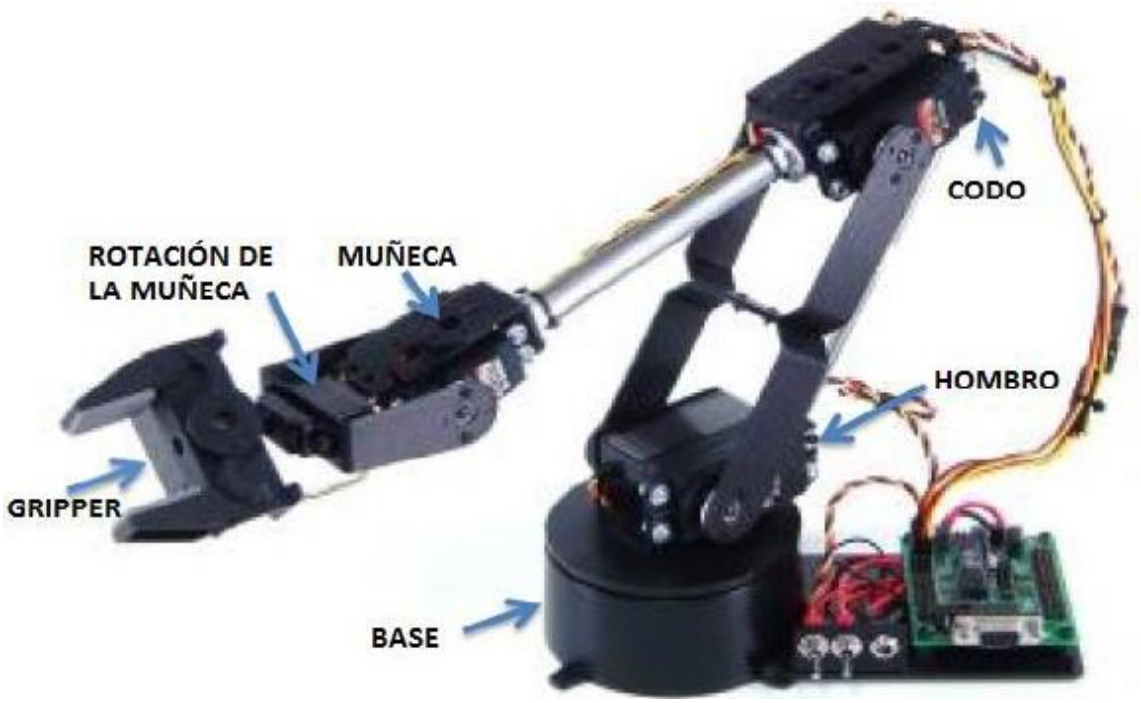
Después de este proceso se aplicarán las operaciones morfológicas de erosión y dilatación, que están descritas en una función específica, a la matriz umbral y se llama a la función de tracking del objeto filtrado, donde se creará una matriz temporal en la que se copiará la matriz umbral. Se inicializará una variable booleana (objetoencontrado) con valor falso que será modificada cuando tengamos una detección que cumpla las condiciones que le impongamos, y se crea un vector de puntos que será para los contornos y otro para la jerarquía (ambos son necesarios para llamar a la función de **OpenCV findContours**. Se aplicará entonces una detección de contornos con la función **findContours** (este procedimiento modifica la imagen, por eso se realiza una copia de la matriz umbral para trabajar con ella y no modificar la original). La función devolverá unos valores que se

quedan guardados en los vectores contornos y jerarquía y creamos una variable de número de objetos encontrados, que será igual al valor de jerarquía devuelto por **findContours**. Si este número de objetos es mayor que cero y menor que el número máximo de objetos que tenemos definidos, a continuación se aplicará el método de momentos para obtener primero el valor del área detectada; Si esta área está entre los valores máximos y mínimos aceptados y es mayor a su vez que el área de referencia anteriormente específicado, se procederá a encontrar los valores de las coordenadas de los centroides de los objetos localizados, el ángulo de orientación con respecto al eje horizontal y se actualiza el valor del área de referencia. También se actualiza el valor de la variable objetoencontrado a true y se recupera la matriz de homografía del archivo .yml que habíamos creado en el proceso descrito en el apartado anterior y se llama a la función antes descrita transformarPunto para convertir los puntos con las coordenadas de los centroides con la matriz de homografía. Se hace una media de diez valores obtenidos para poder enviarle a nuestra tarjeta Arduino un valor estable. Si se termina el proceso, se llama a la función **drawobject** para que dibuje un circulo alrededor del centroide, los valores de las coordenadas en píxeles y el nombre del tipo de objeto que ha sido reconocido.

7. MÓDULO 2: BRAZO ROBÓTICO

7.1. DESCRIPCIÓN GENERAL

Se ha seleccionado como manipulador el brazo robótico **Lynxmotion AL5A**. Se trata de un brazo antropomórfico con 4 ejes y 4 grados de libertad. La base es rotativa y hay articulaciones de hombro, codo y muñeca que se mueven arriba o abajo en el mismo plano. Se ha instalado también la opción de rotación de muñeca que le confiere un grado más de libertad. Dispone de una pinza o gripper, que permite que realice operaciones de "Pick and place". Esto quiere decir que podemos recoger objetos y situarlos en otras coordenadas dentro de nuestro espacio de trabajo. Dispone de 6 servomotores, situados en base, hombro, codo, abatimiento de muñeca, rotación de muñeca y apertura/cierre de pinza. Tiene un alcance horizontal de 14,6 cm y puede manejar una carga de hasta 110 gramos.



El robot está construido principalmente en piezas de aluminio anodizado en los brazos/soportes de servos, una base y un sistema de apertura/cierre de pinza de plástico, tornillería metálica para fijación de piezas y servos y un muelle que se instala entre el codo y el hombro para evitar que el brazo se abata hacia adelante cuando no hay tensión en los servos. También se dispone de rodamientos de bola en la base y un cojinete en la muñeca.

La base lleva incorporada una extensión en la que se puede fijar la tarjeta electrónica para el control del brazo y donde están alojados una toma para alimentación desde la red y dos switches para poder alimentar o quitar tensión tanto a servos como a la placa si se decide suministrarles tensión desde esta conexión externa.

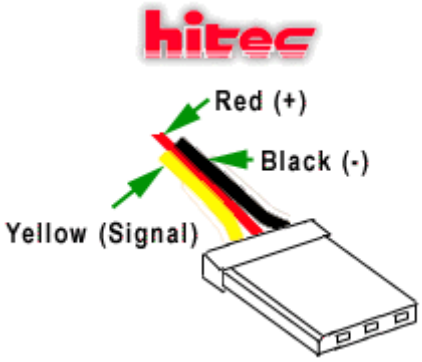
Los servos instalados son de la marca **HI-TEC**, que proporcionan un rango de movimiento por eje de hasta 180 grados. Se alimentan a un voltaje de 4,8 a 6V DC. Los modelos de servo son los siguientes:

- 3 Unidades de **HS-422**: Servo estándar, con un rango de rotación de 180º que proporciona una velocidad de 0,16 s/60º sin carga y un par,a 6 V, de 5,5 Kg.cm. Su ciclo de pulso es de 20 ms y un rango de ancho de pulsos de trabajo entre 900-2100 µs. (1500 µs- al centro). Este modelo de servo está instalado en la base, articulación de la muñeca y apertura y cierre de pinza.

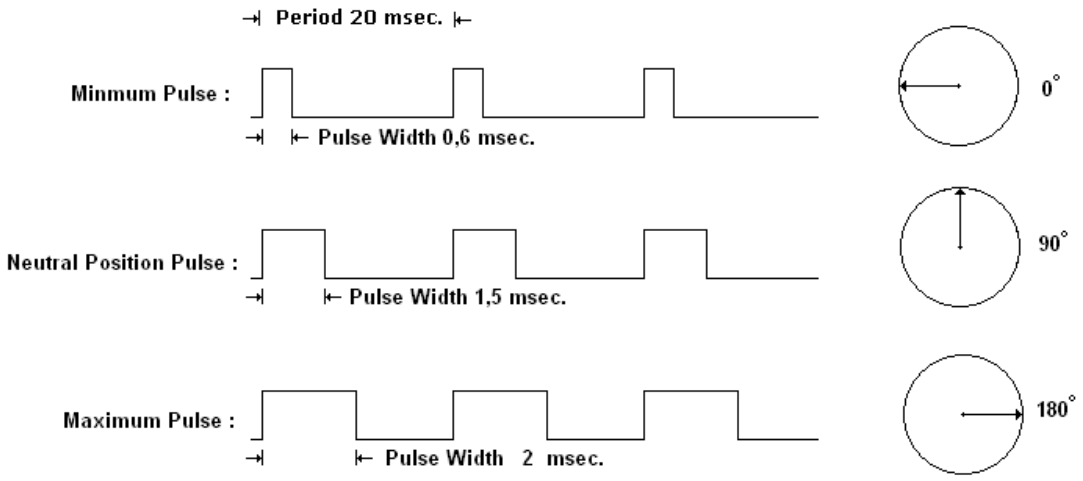
- 1 Unidad de **HS-645MG**: Servo Ultra-torque con engranaje metálico. Proporciona una velocidad de 0,20 s/60º. Tiene un rango de rotación de 180º, ciclo de pulso de 20 ms y un rango de ancho de pulsos de trabajo entre 900-2100 µs. Esta unidad viene instalada en la articulación del codo.

- 1 Unidad de **HS-755HB**: Servo con sistema de transmisión de Karbonite, que asegura una mayor resistencia. Tiene un gran par, de 13,20 kg.cm. Velocidad sin carga de 0,23 s/60º. Rango de 180º, ciclo de pulso 20 ms y rango de ancho de pulsos de trabajo entre 900-2100 µs. Este servo viene acoplado a la articulación del hombro en nuestro brazo robótico.

Todos los servos llevan el siguiente sistema de conexión a la placa de control:



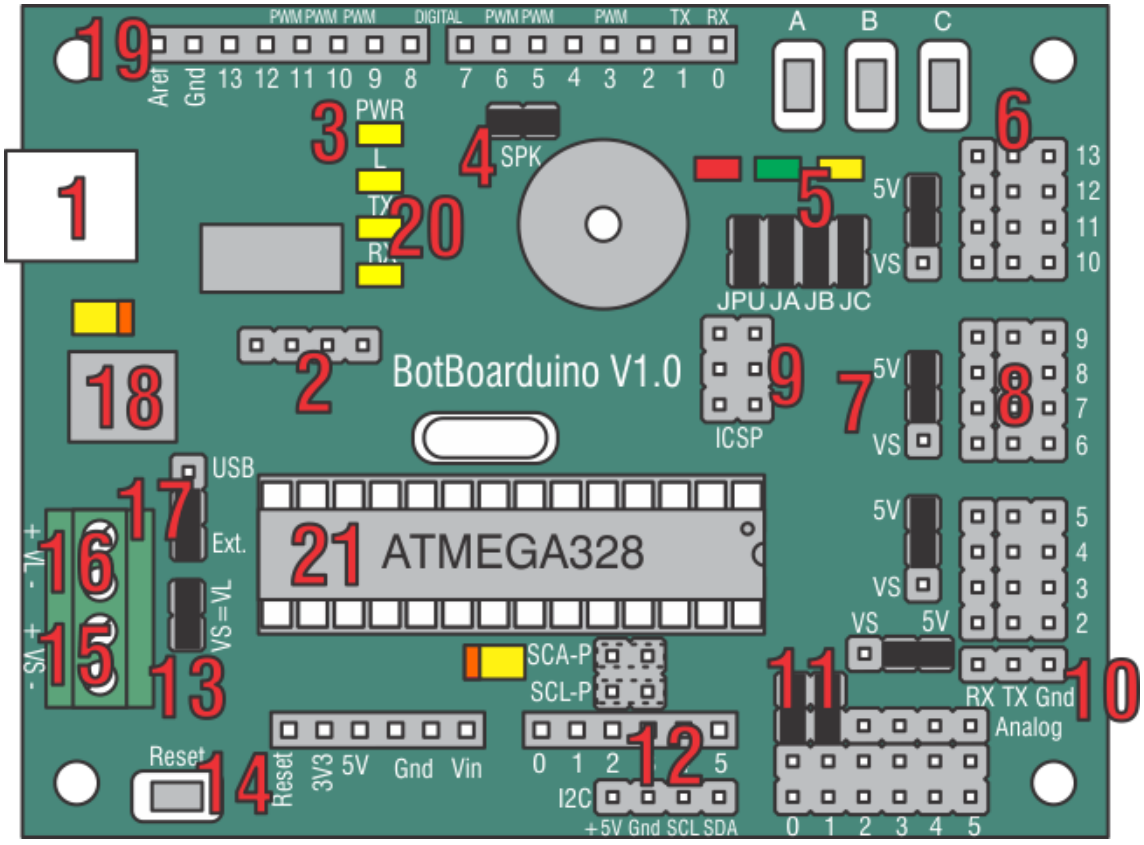
Donde los cables rojo y negro son de alimentación (+ y – respectivamente) y el cable amarillo será de control, por tanto indicará la posición deseada al circuito de control mediante señales **PWM**. Estas señales serán pulsos positivos cuya duración será proporcional a la posición deseada del servo y que se repiten cada 20 ms (50Hz).



Como se puede observar en la ilustración, cuando la anchura de pulso o ciclo útil es de 1,5 ms de duración el servo estará en una posición centrada o neutra. Si el ciclo útil es de 0,6 ms, el servo girará 90 grados en sentido contrario a las agujas del reloj y si es un valor intermedio entre 0,6 y 1,5 ms se desplazará la parte proporcional. Si el ciclo útil es de 2 ms, el servo girará 90 grados en sentido horario. Se han utilizado estos valores, entre 0,6 y 2 milisegundos y no los indicados por el fabricante (Entre 0,9 y 2,1) ya que en la práctica los valores suelen estar más cercanos a los que indicamos aquí.

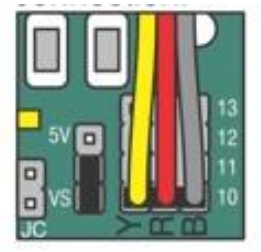
7.2. TARJETA CONTROLADORA: BOTBOARDUINO

De las posibles opciones para el control del brazo robot se ha decidido incorporar una placa compatible con **Arduino**. En este caso el controlador que utilizaremos con el brazo robótico será una placa **Botboarduino**, que es un diseño de **Lynxmotion** modificado a partir de una **Arduino Duemilanove** para poder controlar un mayor número de servos y ofrecer más prestaciones a los que piensen en utilizarla en robótica. A continuación especificaremos algunas características de esta tarjeta:



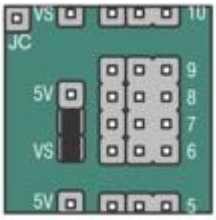
Los componentes de la placa numerados en la imagen serán descritos a continuación:

1. Puerto USB para conectar la placa al ordenador e intercambiar información, así como para ser programada.
2. Pines conectados a las señales **CTS**, **DSR**, **DCD** y **RI** de puertos virtuales **FTDI**.
3. LED indicador de que la placa tiene tensión.
4. Se conecta o desconecta un jumper aquí para activar/desactivar el altavoz incorporado de la placa.
5. Se dispone de este trío de leds y botones por si el usuario desea utilizarlos para interactuar con su programa. Se deben de configurar los pines E/S 7-9 como entradas.
6. Puntos de conexión de servos, controladores de motores, sensores, etc… al microcontrolador. De las tres líneas de pines la exterior es tierra, la central es la alimentación y la interior son los pines E/S.



Ejemplo de conexión de servo.

7. Selector de fuente de tensión para los pines de alimentación mencionados en el punto anterior. Si se coloca el jumper como se ve en la imagen utilizará 5 VDC del regulador interno de la placa, y si se coloca en la opción VS obtiene la tensión de una fuente externa conectada a la toma habilitada para ello.



Selección de alimentación de servos a través de fuente externa.

8. Este puerto puede utilizarse para conectar tanto servos como un mando de Playstation.
9. Estos pines sirven para programar el **Bootloader** del microcontrolador **Atmega** en otro microcontrolador Atmega. Es necesario utilizar un programador AVR para esta operación.
10. Para poder conectar el **Botboarduino** con una placa **SSC-32** de la misma casa **Lynxmotion** con un cable de extensión como el que usan los servos.
11. Permite que las entradas **VL/VS** mencionadas anteriormente se conecten a dos de las entradas análogicas a través de un divisor de tensión de relación 4:1.
12. Para utilizar pines A4 y A5 con el protocolo de comunicación I2C.
13. Para poder alimentar los servos y la placa desde la misma batería. Conecta la entrada VS con la VL.
14. Resetea el microcontrolador cuando se pulsa.
15. Entrada de tensión para servos (VS), entre 4,8 y 7,3 vdc. Se puede alimentar aparte de los servos la placa si se coloca un jumper tal como se comentó en el punto 13.
16. Tensión para placa. Se utilizará normalmente una batería de 9 vdc.

17. Aquí se puede seleccionar el tipo de alimentación que tendrá la placa. Si se escoge USB se alimentará del ordenador por el puerto USB y si se escoge EXT se alimentará de la terminal VL.

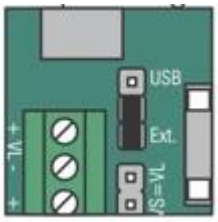


Ilustración 3.Selección de tipo de alimentación de tarjeta

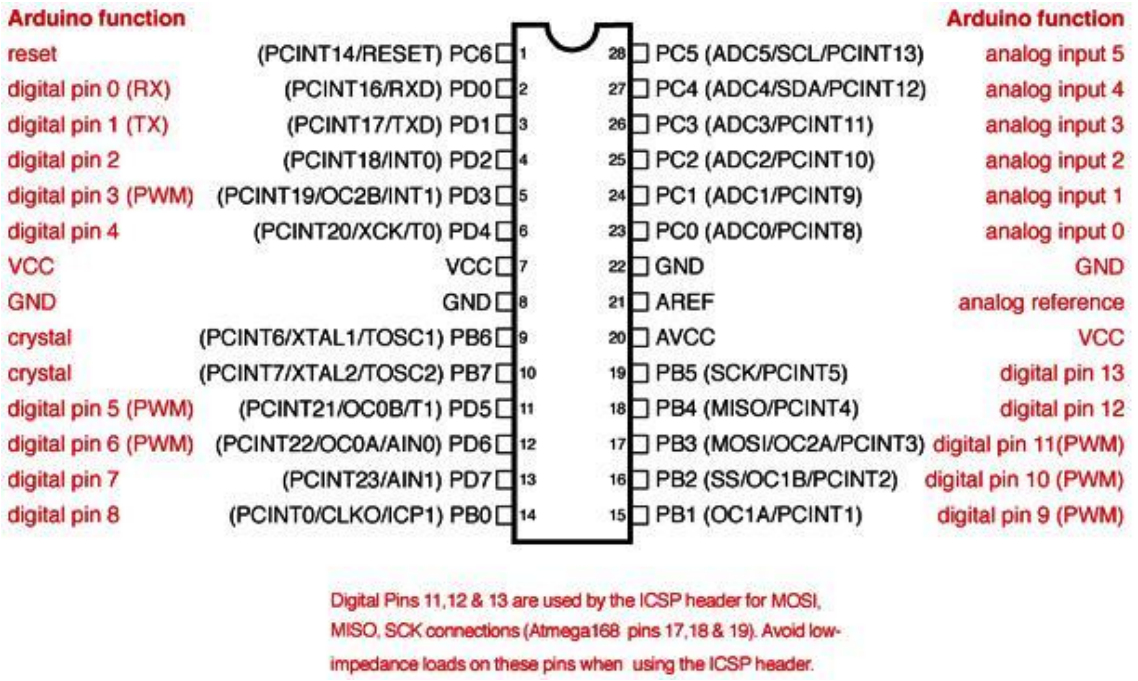
18. Regulador de tensión.
19. Aquí se podrán conectar Shields y extensiones.
20. Leds de estatus: L- Directamente conectado al pin 13 del chip **Arduino**, TX- parpadea cuando envía datos, RX- parpadea cuando recibe datos.
21. Utiliza un microcontrolador compatible con **Arduino ATMEGA 328**:

Frabricado por **ATMEL**, este microcontrolador es de arquitectura **AVR**, basado en microcontroladores **RISC**. Tiene 32 registros de propósito general de 8 bits. Cuenta con 32 KB de memoria flash, 2KB de memoria **RAM** y 1KB de memoria **EEPROM**. Su frecuencia máxima de operación es de 20 MHz y el rango de tensión de operación que requiere es de 1,8 a 5,5 V.

Otras características a destacar son:

- 23 líneas de entrada/salida programables.
- 6 canales conectados a un conversor A/D 10 bits.
- 2 timers de 8 bits con prescaler y modo comparación.
- 1 timer de 16 bits con prescaler, modo comparación y modo captura.
- 6 canales **PWM**.
- Puerto **USART** para comunicación serie programable.

La distribución de pins en el chip **Atmega 328** será la siguiente:



Donde:

- **Port B**: Puerto E/S de 8 bit, bidireccional, con resistencias pull-up internas.
- **Port C**: Puerto E/S de 7 bit, bidireccional, con resistencias pull-up internas.
- **PC6/RESET**: Si el fusible **RSTDISBL** está programado, **PC6** se usará como un pin E/S. Si el fusible no está programado, **PC6** será el que actúe como entrada de Reset. Un nivel bajo en este pin durante un tiempo superior a la longitud de pulso mínima generará un Reset.
- **Port D**: Puerto **E/S**, 8 bit, bidireccional con pull-up internas.
- **AV_{CC}**: Toma tensión para el Conversor analógico/digital.
- **AREF**: Pin de referencia analógica para el conversor **A/D**.

Por último, la tarjeta lleva también montado un cuarzo de 16MHz como oscilador externo para el microcontrolador.

Los motivos por los que se ha seleccionado esta tarjeta son su adaptabilidad a proyectos de robótica en los que se requiera utilizar hasta 12 servos y por otro lado su compatibilidad con **Arduino**, lo que nos permitirá programarla a través de la **IDE** de **Arduino**.

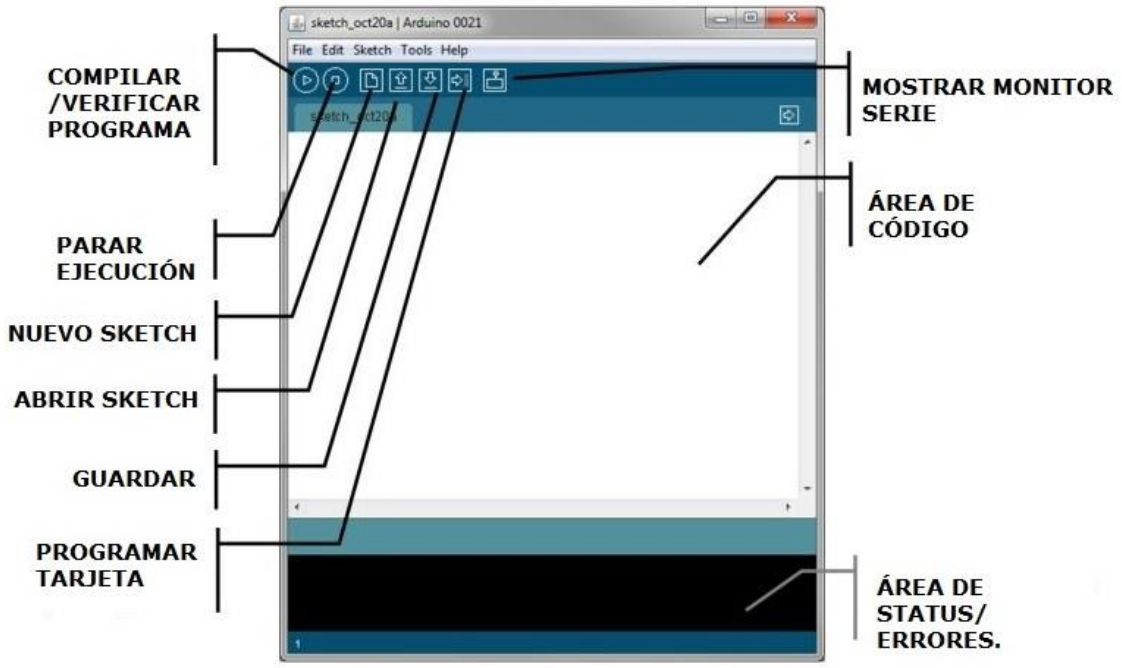
7.2.1. ARDUINO

Arduino es una plataforma libre de hardware que desarrolla tarjetas sencillas con un microcontrolador y un lenguaje de programación que basado en Wiring. Puede ser programado para trabajar de manera autónoma o conectado a un PC o Mac, y es compatible para establecer comunicación con muchos tipos de software (Processing, flash,..). Esta plataforma también suministra de manera gratuita un entorno de desarrollo integrado, basado en **Processing**, (IDE) que se puede descargar a través de su página web www.arduino.cc/es.

Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en Java, pero con una sintaxis simplificada de fácil utilización. Fue inicialmente diseñado para desarrollar aplicaciones interactivas en proyectos de arte o pedagógicos.

Las ventajas de **Arduino** respecto a la competencia son varias, pero las principales son la asequibilidad de su hardware, la adaptabilidad y compatibilidad con todos los sistemas operativos de su software y que por supuesto este software está publicado bajo licencia libre y recibe aportaciones en forma de librerías y actualizaciones muy frecuentemente gracias a su comunidad.

El lenguaje de programación de **Arduino** es similar al C, y este es uno de los motivos para su selección, para poder facilitar la programación de código para el brazo robótico. El entorno de programación es muy asequible para principiantes o usuarios con conocimiento limitado de programación y se dispone de librerías de C++. Mediante la **IDE** de **Arduino** podremos escribir nuestro programa , verificarlo, guardarlo en un archivo para poder recuperarlo posteriormente y programar el controlador a través del cable USB. Esta herramienta nos ofrece una ventana sobre la que escribiremos nuestro programa en lenguaje **ARDUINO**. Estos programas se conocerán como “sketch”.



IDE de Arduino.

Para empezar a escribir programas, primero debemos escoger nuestra placa en el menú de Herramientas. En nuestro caso sería la **Arduino Duemilanove**.

Al instalar los drivers de **Arduino** Windows asigna automáticamente un canal de comunicaciones para la tarjeta y debemos comprobar en Panel de control/Administrador de dispositivos/Puertos (**COM&LPT**) el canal **COM** al que se ha asignado.

Un programa escrito en la **IDE** de **Arduino** se debe de estructurar en 3 secciones:

1. Al comienzo del programa se hace definición de constantes, variables, etiquetas, etc… que vayan a ser usadas por él.
2. **Setup**: Se configuran las líneas digitales que deben de actuar como entradas y salidas. La función **setup()** sólo se ejecuta una vez y su misión es contener todas las sentencias que deben ser ejecutadas siempre que se inicia el sistema. En nuestro programa también inicializamos la comunicación serie , asignamos los servos a sus correspondientes pines y llamamos a una función definida para que el brazo se asiente en una posición inicial definida.
3. **Loop**: Cuerpo principal del programa. Todas las sentencias se ejecutan indefinidamente en un bucle, como su propio nombre indica. Se empieza ejecutando la primera instrucción y se continúa hasta la última y se vuelve a empezar por la primera.

7.2.2. PROGRAMA CONTROL BRAZO ROBÓTICO

Nuestro programa **Arduino** para controlar el brazo robótico consistirá en un sistema para comunicación serie con Visual Studio que nos permita recibir las coordenadas del bolo derribado que hay que recoger y varias funciones:

- Una función **pos_brazo**() para situar el brazo en una coordenadas determinadas para el que desarrollamos la cinemática inversa necesaria siguiendo el procedimiento que será mencionado en el siguiente apartado.
- Una función **servo_inicio**() para posicionar el brazo en una situación inicial.
- Una función **brazo_recogida**() que gestiona los movimientos que conforman la recogida de un objeto.
- Una función **servo_coloca**() que gestiona los movimientos que conforman la forma en que se deposita el bolo en el lugar que les corresponde.

Utilizaremos una librería llamada **VarSpeedServo.h**, que es una adaptación de la librería estándar **Servo.h**.

Esta librería nos permite controlar motores servo. Los servos estándar, que serán los que utilicemos en éste trabajo, permiten que el eje se sitúe en varios ángulos, por lo general entre 0 y 180 grados. La librería servo permite hasta 12 motores. Es necesario crear unas variables servo en la sección de código inicial, en la que se realizan las definiciones y declaraciones de variables.

Esta librería ofrece las siguientes funciones:

- **attach**(): Asocia la variable Servo a un pin. ***servo.attach(pin)*** o ***servo.attach (pin,min,max)*** serán formas de emplearla. Min y max serán el ancho de pulso en microsegundos correspondientes al ángulo del servo mínimo o máximo respectivamente. Por defecto, si no se especifican, los valores serán (544,2400).
- **write**(): Escribe un valor en ángulo para la posición del eje del servo. ***servo.write(angulo)***.
- **writeMicroseconds**(): Escribe un valor en microsegundos en el servo, controlando el eje. En un servo estándar (nuestro caso) lo

llevará al ángulo que le corresponda. Como habíamos comentado anteriormente, un valor de 600-700 microsegundos llevará el servo completamente a la izquierda, 2000-2100 hará lo propio a la derecha y 1500 lo dejará en posición neutra. **servo.writeMicroseconds(valor**).

- **read**() : Lee el ángulo actual del servo. Devuelve el ángulo en grados**. servo.read().**
- **attached**(): Comprueba si la variable Servo está asociada a un pin. Devuelve true si está asociado o false si no lo está. **servo.attached()**.
- **detach**(): Desasocia la variable Servo de su pin. **servo.detach()**.

Como habíamos comentado, utilizaremos una librería que es una modificación de Servo.h, y que aparte de las funciones que hemos comentado añade la siguiente mejora, que nos va a ser útil para el control del brazo:

- **myservo.slowmove**(): Esta función nos permitirá que los servos se desplacen a la velocidad que nosotros le ordenemos. **myServo.slowmove(newpos, speed)**. La velocidad puede tener valores entre 1 y 255, siendo 1 la más lenta posible y 255 la más rápida.

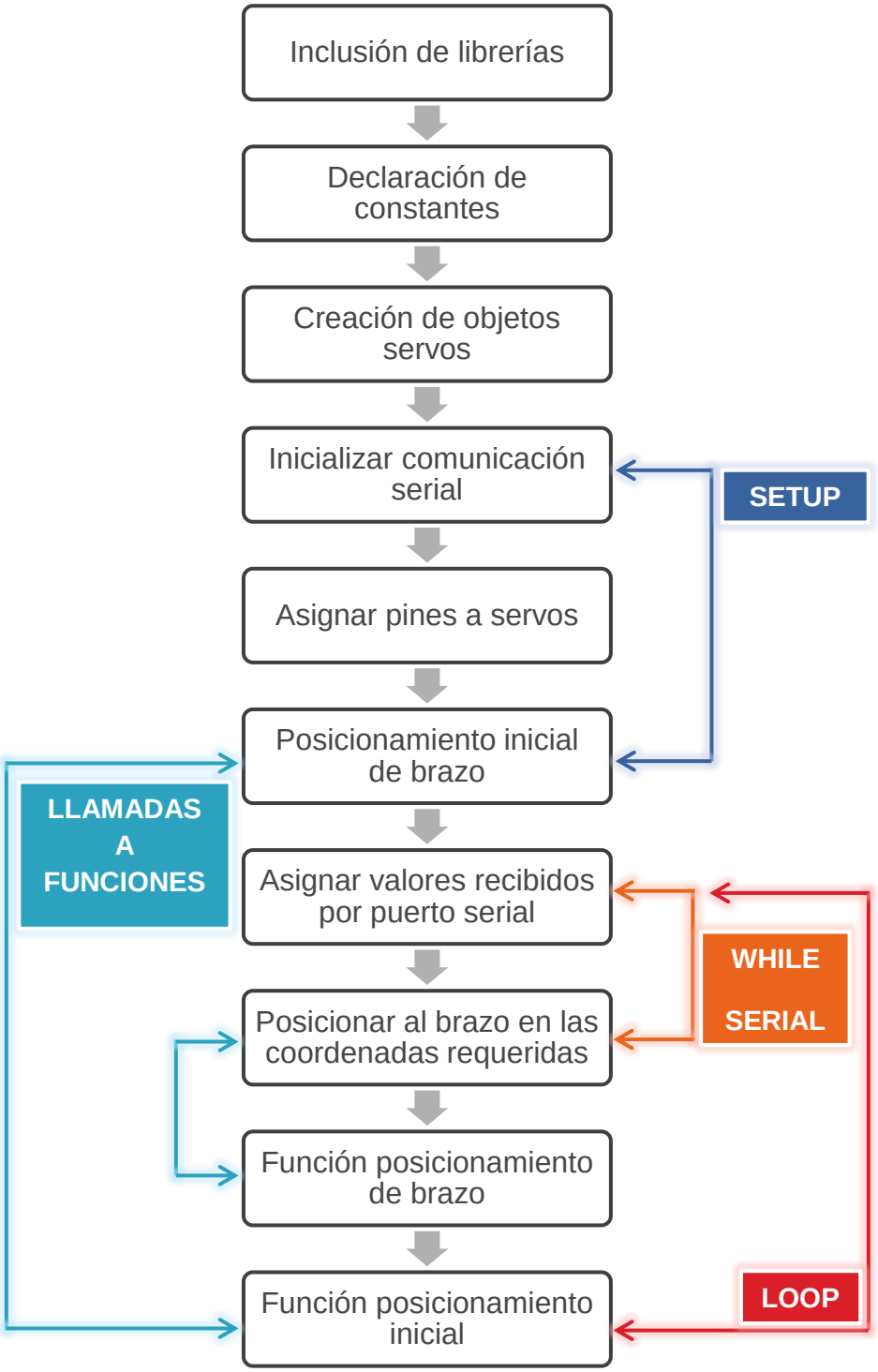
El motivo de que utilicemos esta versión de la librería es que si se escribe las posiciones de los servos con microsegundos utilizando la librería servo resulta complicado efectuar un control de la velocidad de estos y se producen unos movimientos muy bruscos, que pueden llegar a ser perjudiciales para la estructura del brazo, aparte de generar movimientos de latigazo no deseados que harían imposible un control efectivo. Se debe de tener en cuenta que no se pueden utilizar ambas librerías (Servo y **VarSpeedServo**) simultáneamente, al definir ambas algunas funciones de modo común, lo que generaría conflictos en la compilación. Las variables de los servos también se declaran una de forma ligeramente distinta:

VarspeedServo myservo

En lugar de la forma de la librería estándar:

Servo myservo

La estructura del programa desarrollada en la IDE de Arduino para el control del brazo será la siguiente:


Inclusión de librerías
Declaración de constantes
Creación de objetos servos
Inicializar comunicación serial
SETUP
Asignar pines a servos
Posicionamiento inicial de brazo
LLAMADAS A FUNCIONES
Asignar valores recibidos por puerto serial
WHILE SERIAL
Posicionar al brazo en las coordenadas requeridas
Función posicionamiento de brazo
Función posicionamiento inicial
LOOP

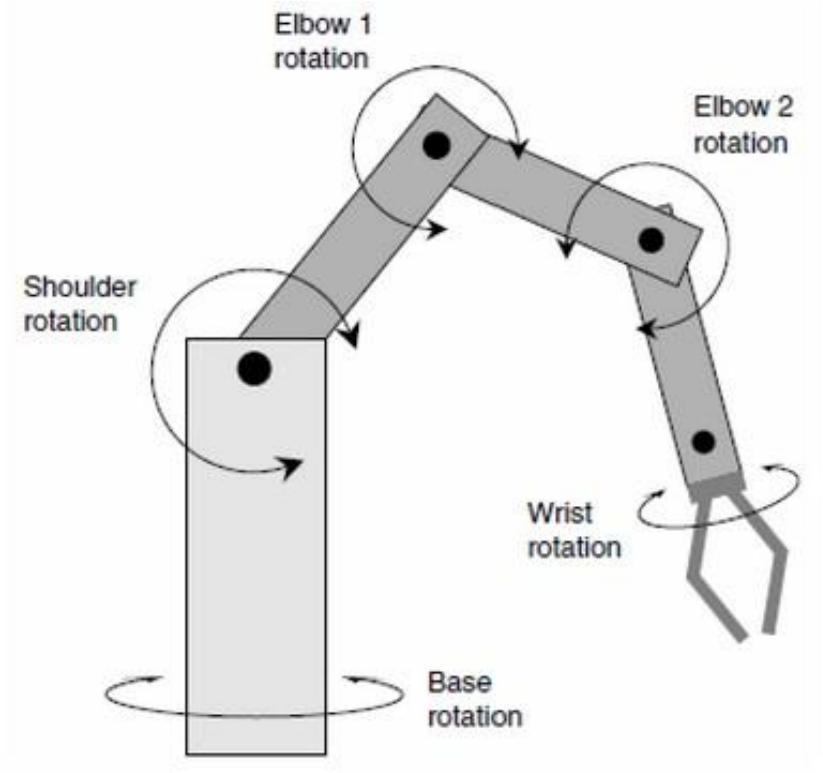
7.2.2.1. CÁLCULOS CINEMÁTICOS

El análisis cinemático considera los conceptos de posición, orientación, velocidad y aceleración entre los distintos componentes de un sistema.

El objetivo de este modelo es entender el movimiento de los eslabones y articulaciones del brazo robótico a través del análisis de las capacidades de movimiento de cada articulación que maneja dichos eslabones.

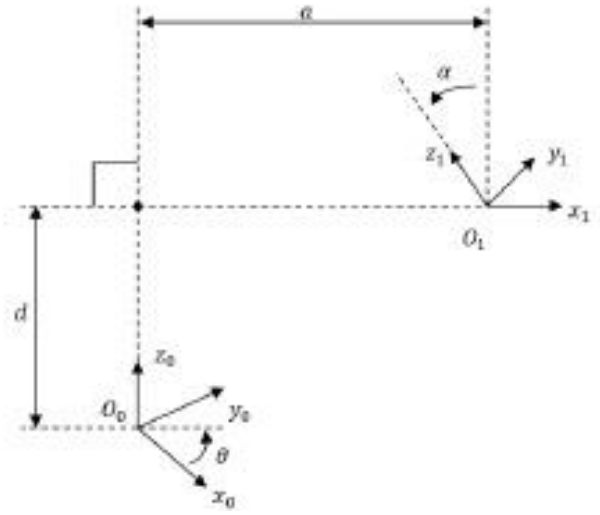
El primer paso a realizar para llevar a cabo este análisis es determinar los grados de libertad que aportan las articulaciones. Entendemos grados de libertad como las coordenadas independientes sobre las cuales pueden describirse las configuraciones del sistema robótico.

En el gráfico bajo estas líneas se pueden apreciar las articulaciones que tendrán información que pueda contribuir para definir la configuración del sistema.



Al considerar que el tipo de articulaciones instaladas en el robot son rotacionales planares y sólo aportarán un grado de libertad cada una, podemos concluir que disponemos de un robot con 4 grados de libertad, ya que la rotación de la muñeca no va a tener trascendencia para condicionar la localización del extremo del manipulador.

Una condición que ha de cumplir cada elemento de un sistema mecatrónico de estas características es que debe contar con un sistema coordenado adherido a su estructura, sobre la cual se realizará la definición de su posición y su orientación, con respecto a un sistema coordenado base. Este sistema coordenado de cada elemento es comúnmente situado sobre el eje de cada actuador.



En la imagen se pueden apreciar dos sistemas de coordenadas **x,y,z** que estarían centrados en las articulaciones, de manera que la rotación de estas se produciría respecto del eje **z**. Cada eslabón recibirá la acción de una articulación. Teniendo en cuenta esto, es posible estimar el movimiento de toda la cadena a partir del movimiento de cada uno de los eslabones.

Las matrices homogéneas, de la forma $^{0}\mathbf{T}_{1}$ determinarán las relaciones geométricas de cada articulación y su eslabón correspondiente en la cadena, hasta llegar al eslabón final. De forma matemática se puede considerar la cinemática directa como el producto de las matrices que representan cada uno de los eslabones, comenzando desde el primero y terminando por el efector, o actuador final.

Dicha expresión de la cinemática directa tendrá una gran dependencia respecto al vector **q** , denominado vector de estado o de coordenadas generalizadas. Este vector contiene el valor instantáneo de cada uno de los grados de libertad.

En nuestro caso, con 4 grados de libertad, tendríamos un vector **q** de la siguiente forma:

$$q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix}$$

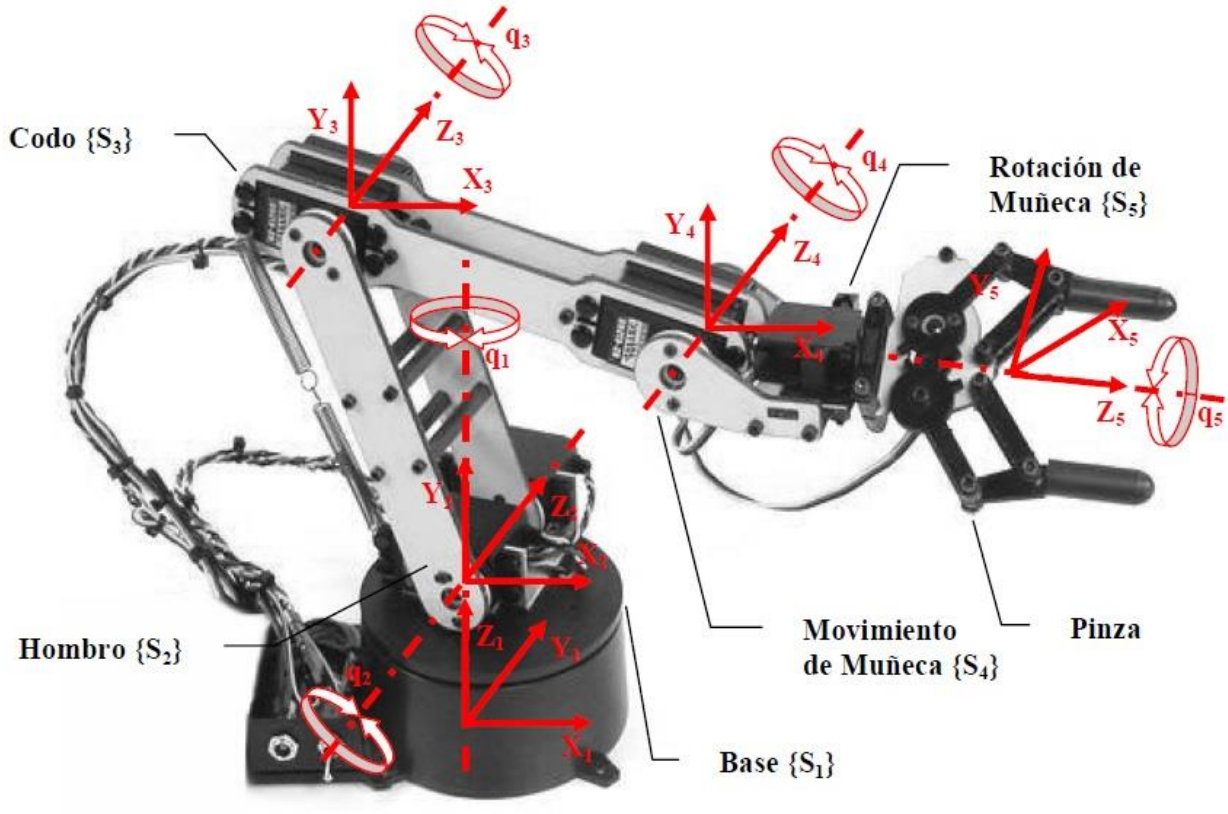
donde $\boldsymbol{\theta_1, \theta_2, \theta_3}$ **y** $\boldsymbol{\theta_4}$ serían los valores de cada grado de libertad.

La convención Denavit-Hartenberg (**DH)** permitirá representar la relación geométrica desde la base de un sistema hasta su último eslabón, considerando la cadena cinemática compuesta por articulaciones y eslabones. Se expresa a través de la transformada homogénea ${}^{0}\mathbf{T}_{e}$. Se necesita una matriz homogénea por cada elemento de la cadena cinemática y cuando se disponga de todas las matrices de todos los elementos se obtendrá el producto de todas ellas, quedando así definida la transformada de toda la cadena, desde la base hasta el extremo-efector. Las matrices transformadas homogéneas de los elementos tienen la siguiente forma:

$$\mathbf{{}^{n-1}T_n\ (\theta_n)} = \begin{bmatrix} \boldsymbol{cos\theta n} & \boldsymbol{-sen\theta n.cos\alpha n} & \boldsymbol{sen\theta n.sen\alpha n} & \boldsymbol{an.cos\theta n} \\ \boldsymbol{sen\theta n} & \boldsymbol{cos\theta n,cos\alpha n} & \boldsymbol{-cos\theta n.sen\alpha n} & \boldsymbol{an.sen\theta n} \\ & \boldsymbol{sen\alpha n} & \boldsymbol{cos\alpha n} & \boldsymbol{dn} \\ & & & \mathbf{1} \end{bmatrix}$$

Los pasos más importantes de esta convención serán los siguientes:

- Enumerar articulaciones consecutivamente empezando desde el 0.
- Colocar el eje **z** en cada sistema coordenado, de forma que apunte en la misma dirección del eje de rotación de la articulación.
- Ajustar dirección de eje **x** para que apunte al origen del siguiente sistema de coordenadas.
- Medir la longitud del eslabón desde el origen de un sistema de coordenadas hasta el origen del siguiente en la dirección del eje **x**. Este será el parámetro **a**.
- Determinar el ajuste o separación entre eslabones consecutivos para conseguir el parámetro **d**.
- Si hace falta realizar un ajuste de rotación alrededor del eje **x**, para que el eje de rotación de una articulación coincida con el de la anterior, el ángulo a rotar será **α**.
- Rellenar la tabla **DH** con los parámetros obtenidos para cada eslabón. Cada fila representará un eslabón.



En la figura podemos ver representados los parámetros a introducir en la tabla DH

Abajo se muestra un ejemplo de tabla **DH** para eslabones y articulaciones rotacionales.

Eslabón	α	a	d	Variable
L_1	α_1	a_1	d_1	θ_1
L_2	α_2	a_2	d_2	θ_2
…	…	…	…	…
L_n	α_n	a_n	d_n	θ_n

Para abordar el problema de la cinemática inversa, vamos a utilizar el método iterativo a través de la **Robotics Toolbox** de **Matlab** y también el método basado en consideraciones geométricas. Éste último nos ofrecerá las expresiones matemáticas que luego adaptaremos para introducir en el programa **arduino** creado para el control del brazo robótico.

El objetivo del estudio de la cinemática inversa es discutir métodos para definir el valor de las coordenadas generalizadas solicitadas para que una cadena cinemática alcance una posición y orientación requerida. Esta posición se expresa comúnmente a través de una matriz de transformación.

Por tanto, si nos atenemos a la terminología que venimos utilizando hasta el momento, la solución del problema de cinemática inversa será el vector de coordenadas generalizadas **q** requerido para llevar la cadena cinemática a la posición y orientación definida por la matriz homogénea **T**.

Matemáticamente, la solución del problema de cinemática inversa considera cada uno de los componentes de q, es decir, para el ejemplo de nuestro manipulador antropomórfico, dichos valores serán $\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$, $\boldsymbol{\theta_3}$, $\boldsymbol{\theta_4}$, $\boldsymbol{\theta_5}$. De esta forma, la solución al problema de cinemática inversa se puede expresar de la siguiente forma:

Si tenemos la matriz homogénea de la cadena cinemática **T**, se calculará:

$$\mathbf{q} = [\boldsymbol{\theta_1}\ \boldsymbol{\theta_2}\ \boldsymbol{\theta_3}\ \boldsymbol{\theta_4}\ \boldsymbol{\theta_5}]^T$$

Los inconvenientes que se plantean durante la resolución de este problema son los siguientes:

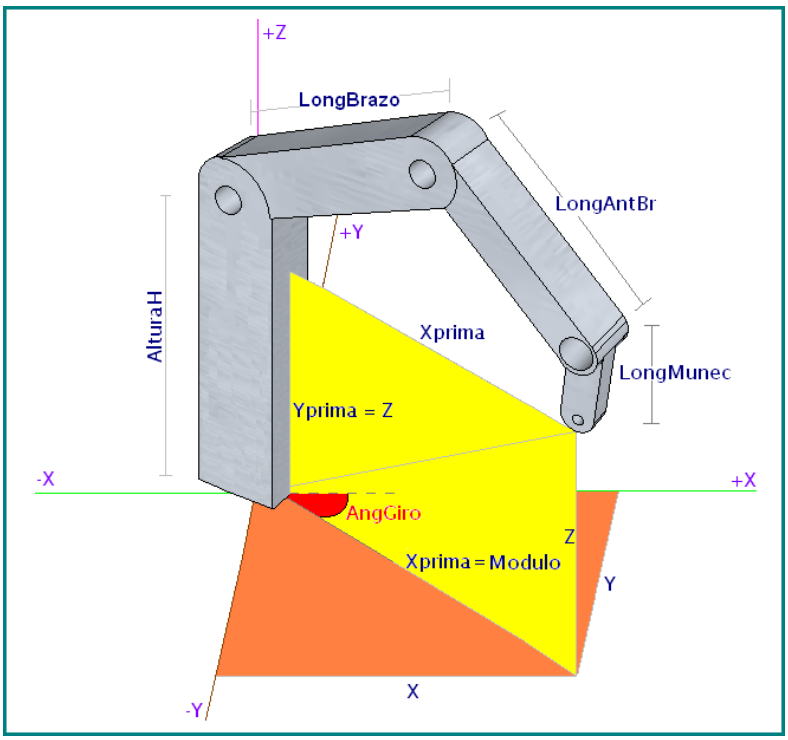
- Es posible tener distintas soluciones para una misma localización deseada para el actuador final o efector. Este es un problema común en cadenas de cinco o más grados de libertad, que pueden alcanzar la misma posición en el espacio con diferentes configuraciones.
- Las expresiones que se deben resolver para definir la cinemática inversa son generalmente no lineales, lo que nos impedirá en ocasiones encontrar una solución cerrada.
- Hay riesgo de encontrar soluciones, que debido a las limitaciones mecánicas de un sistema robótico, pueden ser inadmisibles.

7.2.2.2. CÁLCULO GEOMÉTRICO DE CINEMÁTICA INVERSA

Deseamos alcanzar una posición en el espacio con el extremo de nuestro actuador, pero necesitamos conocer los ángulos de las articulaciones requeridos para conseguirlo.

En el caso de nuestro robot, que será un brazo con 3 segmentos y 5 grados de libertad (aunque descartaremos en los cálculos el giro de muñeca, porque no tendrá trascendencia para determinar la localización del actuador final), se puede calcular una solución analítica mediante el uso de métodos geométricos, que consisten en la utilización de relaciones trigonométricas y resolución de triángulos formados por los segmentos y articulaciones del robot.

Si analizamos los grados de libertad y el tipo de movimiento que tiene cada elemento de nuestro brazo podremos observar que la base se desplazará rotando sobre el eje que llamamos **X**. Tendremos otro eje **Y** que pasará a ser la profundidad y finalmente un eje **Z** que será la altura. En realidad, las articulaciones del hombro, codo y muñeca van a producir un movimiento de abatimiento que se puede analizar entre los ejes **Y,Z**. Con estas consideraciones iniciales será más sencillo proceder al estudio de la cinemática inversa.



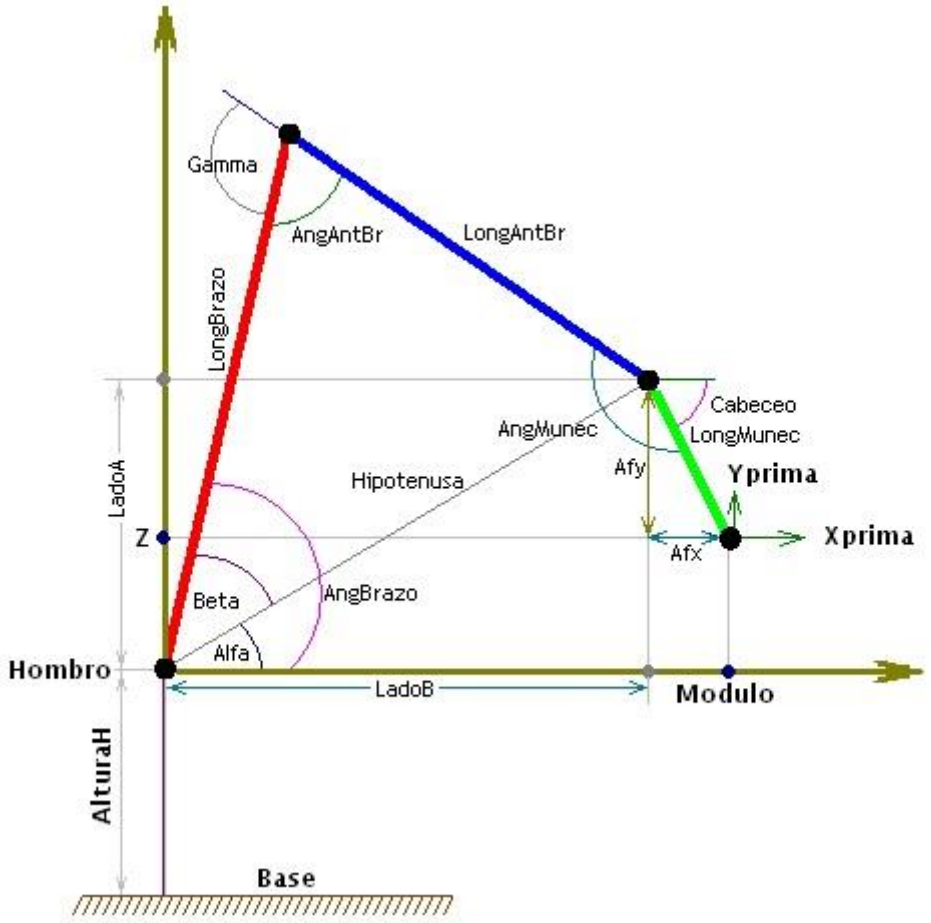
Hallaremos primero el ángulo de giro del brazo de la siguiente forma:

$$\mathbf{AngGiro = Atan2(y,x)}$$

$$\mathbf{Modulo = x^2 + y^2}$$

Este valor representará el radio desde el eje de rotación de la base en el plano **x,y**

Teniendo en cuenta que el brazo tiene 4 grados de libertad, habrá infinitas soluciones posibles para que el brazo alcance un punto **(x,y,z)**. Debemos introducir una condición para mantener la pinza en un ángulo específico respecto la horizontal y calcular unas nuevas coordenadas **(x_1, y_1, z_1)** del eje de la muñeca. Trabajaremos con un sistema plano basado en los dos ejes **Xprima= Modulo** e **Yprima= Z**.



De aquí podremos extraer las siguientes operaciones:

$$\mathbf{Afx = cos(Cabeceo)*LongMunec}$$

Que será la proyección sobre el eje **Xprima** de la muñeca

$$\mathbf{LadoB = Xprima - Afx}$$

Serán la distancia al eje **Yprima** de la articulación de la muñeca

$$\mathbf{Afy = sin(Cabeceo)* LongMunec}$$

Será la proyección en el eje **Yprima** de la muñeca

$$\mathbf{LadoA = Yprima - Afy - AlturaH}$$

Será la distancia al eje **XPrima** de la articulación de la muñeca

$$\mathbf{Hipotenusa = sqr(LadoA^2 + LadoB^2)}$$

Será la distancia entre la articulación del hombro y la articulación de la muñeca

$$\mathbf{Alfa = Atan2(LadoA, LadoB)}$$

Será el ángulo que forma el segmento que une la articulación del hombro y la articulación de la muñeca con el eje **Xprima**

$$\mathbf{Beta = Acos((LongBrazo^2 - LongAntBr^2 + Hipotenusa^2) / (2* LongBrazo * Hipotenusa))}$$

Será el ángulo que forma el segmento que une la articulación del hombro y la articulación de la muñeca con el brazo

$$\mathbf{AngBrazo = Alfa + Beta}$$

Este será el ángulo del hombro

$$\mathbf{Gamma = Acos((LongBrazo^2 + LongAntBr^2 - Hipotenusa^2) / (2* LongBrazo * LongAntBr))}$$

Ángulo que forma el segmento del hombro con el segmento del antebrazo

$$\mathbf{AngAntBr = -(180-Gamma)}$$

Éste será el ángulo del codo

$$\mathbf{AngMunec = Cabeceo - AngBrazo- AngAntBr}$$

Ángulo de abatimiento de la muñeca.

Estas expresiones han sido incluidas en el código desarrollado en **Arduino** para el control de la posición de nuestro brazo robótico.

7.2.3. CINEMÁTICA DIRECTA E INVERSA CON MATLAB.

Para realizar el estudio de la cinemática directa e inversa y obtener el espacio de trabajo del brazo robótico empleado se recurrirá a la **Robotics Toolbox** para **Matlab**, desarrollada por **Peter Corke**. Esta ofrece, de forma gratuita, una serie de herramientas y comandos que nos permitirán realizar los cálculos de una manera sencilla.

Esta herramienta de **Matlab** nos proveerá de funciones específicas para emplear vectores, transformadas homogéneas y ángulos de Euler, entre otros, para facilitarnos representar posiciones tridimensionales y orientación.

Para instalar esta toolbox primero necesitamos descargar un archivo .zip en el siguiente enlace, perteneciente a la página web de su desarrollador, **Peter Corke**:

http://www.petercorke.com/Robotics_Toolbox.html

La versión empleada es la octava, aunque actualmente está disponible la novena. Se ha utilizado con **Matlab R2011a**, con el que es plenamente compatible.

La carpeta contenida en el archivo .zip descargado se han extraído a la carpeta que contiene el resto de toolboxes de **Matlab** y posteriormente, para su utilización, se ha agregado al **MATLABPATH**. Finalmente, para activarla se introduce el comando **startup_rvc**.

La base de utilización de esta Toolbox es el modelado bajo convención **Denavit-Hartenberg (DH)**, que permitirá la construcción de matrices homogéneas a partir de cada renglón de parámetros (cada uno de los cuales representará un grado de libertad del robot). Su arquitectura utiliza la estructura de clases de **Matlab** para almacenar cada renglón de la tabla **DH**.

Para la simulación de un modelo **DH**, utilizaremos de base la clase "link", que nos permitirá almacenar los parámetros de cada renglón en la tabla **DH**. Un objeto derivado de la clase "**link**" podrá ser utilizado después como parámetro de entrada en la clase "**SerialLink**", que enlazará cada uno de los eslabones dentro de una única estructura.

Este objeto "**SerialLink**" va a contener la información cinemática y dinámica de un eslabón.

La utilización de la clase "**link**" para introducir los parámetros del brazo robot necesarios para crear la tabla **DH** y realizar posteriores cálculos será como sigue:

Eslabon(1) = Link ([θ d a α p])

Siendo **θ** el valor angular que representa la posición de la articulación, **d** la distancia de ajuste (offset) entre eslabones consecutivos, **a** longitud del eslabón desde el origen del sistema **n-1** hasta el origen del sistema **n** sobre la dirección establecida por el eje **x**, **α** será el giro necesario a aplicar al eje $\mathbf{z_{n-1}}$ para ajustar la desviación que tenga con respecto al eje $\mathbf{z_n}$ tomando como eje de rotación al eje $\mathbf{x_{n-1}}$. Finalmente, el parámetro p podrá tener sólo valores 1 o 0, siendo 1 en caso de representar un eslabón prismático o 0 en caso de ser un eslabón rotacional (por defecto).

Una vez definidos los eslabones utilizaremos el comando "**SerialLink**" para unir todos los eslabones en un objeto de dicha clase y formar una cadena cinemática única.

Robot= SerialLink(Eslabon)

Posteriormente podremos ponerle nombre al objeto "**SerialLink**" y representarlo con la función **plot** y mostrar la simulación con **teach**:

plot(Robot)
plot.teach

Al crear el robot, el programa nos devuelve un listado de datos en el que nos indica el número de articulaciones que conforman la cadena, RRR en nuestro caso al ser tres articulaciones rotacionales (en caso de haber prismáticas se hubieran indicado con la letra **P**), el nombre que le hayamos puesto, el vector de gravedad que se utilizará en la simulación y los valores de eslabones que le hayamos introducido previamente:

```
Manuel (3 axis, RRR, stdDH)
 Prototipo;
+---+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |
+---+-----------+-----------+-----------+-----------+
|  1|        q1|       0.4|         0|         0|
|  2|        q2|       0.5|         0|         0|
|  3|        q3|       0.3|         0|         0|
+---+-----------+-----------+-----------+-----------+
```

```
grav =    0  base = 1  0  0  0   tool =  1  0  0  0
          0           0  1  0  0          0  1  0  0
       9.81           0  0  1  0          0  0  1  0
                      0  0  0  1          0  0  0  1
```

Se puede apreciar también como indica que se trata de **DH** standard (**stdDH**).

Para calcular la cinemática directa de una cadena robótica como la que hemos construido, que será representada por la matriz homogénea **$^0T_e(q)$** cuyo contenido es la posición y orientación del actuador final en la cadena cinemática, el método que se utiliza es construir cada una de las matrices **T** representando a cada eslabón, a partir de su renglón respectivo en la tabla **DH** y su coordenada generalizada correspondiente, para después realizar la integración a través del producto de todas las matrices de transformación.

En **Robotics Toolbox**, sin embargo, disponemos de una función "**fkine**" que calcula la multiplicación de todas las matrices. La utilizaremos de la siguiente forma:

T = nombre_robot.fkine([vector_coordenadas q])
Los dos argumentos que necesitamos son el nombre del robot que hayamos creado a través de la tabla **DH** y **q**, que será el vector de valores para las coordenadas generalizadas:

$$q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -\pi/4 \\ -\boldsymbol{\pi}/\boldsymbol{4} \end{bmatrix}$$

Donde $\boldsymbol{\theta_1}$**,** $\boldsymbol{\theta_2}$ **y** $\boldsymbol{\theta_3}$ son los ángulos en cada uno de los eslabones, que van a definir la posición del robot.

Creamos este vector q de la siguiente forma:

q = [0 -pi/4 -pi/4]

y ahora podemos utilizar la función fkine para obtener la matriz homogénea:

T = robotlibro.fkine(q)

Y obtenemos el siguiente resultado:

T =

0.0000	**1.0000**	**0**	**0**
-1.0000	**0.0000**	**0**	**0**
0	**0**	**1.0000**	**1.2000**
0	**0**	**0**	**1.0000**

Para implementar este método en nuestro robot vamos a definir el origen del sistema en la base del robot, los ejes de rotación los consideraremos **z**, tal como aconseja la convención **DH**, los ejes **x** unen en línea recta orígenes de sistemas de coordenadas de cada articulación y el eje y se asumirá por la regla de la mano derecha.

Obtenemos las dimensiones que necesitamos para rellenar la tabla **DH** y las introducimos en sus respectivas casillas:

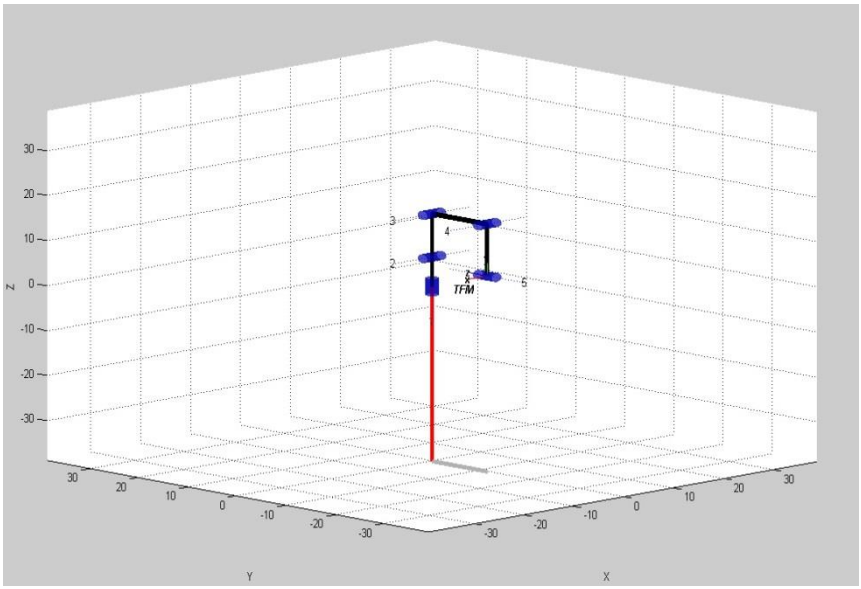
Eslabón	α	a	d	Variable
1	90	2,5	6,5	θ_1
2	0	9,6	0	θ_2
3	0	11,2	0	θ_3
4	-90	11,5	0	θ_4
5	0	0	0	θ_5

Se crean los eslabones ("**Eslabon**") y el objeto **SerialLink** ("**Brazo**") conteniéndolos a todos, tal como se ha explicado anteriormente y se pide una representación gráfica en una posición en la que todos los ángulos de las articulaciones **θ** sean **pi/2** y que también muestre el entorno gráfico "drivebot" para modificar los valores de las coordenadas generalizadas y cambiar de posición y orientación el actuador final.

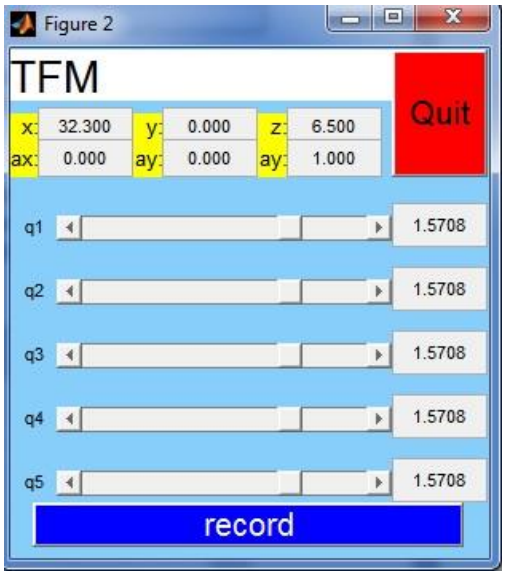
```
Brazo =


TFM (5 axis, RRRRR, stdDH)
 Prototipo;
+---+-----------+-----------+-----------+-----------+
| j |    theta |       d |       a |    alpha |
+---+-----------+-----------+-----------+-----------+
|  1|       q1|      6.5|        0|     1.571|
|  2|       q2|        0|      9.6|        0|
|  3|       q3|        0|     11.2|        0|
|  4|       q4|        0|     11.5|    -1.571|
|  5|       q5|        0|        0|        0|
+---+-----------+-----------+-----------+-----------+

grav =    0  base = 1  0  0  0   tool =  1  0  0  0
          0          0  1  0  0           0  1  0  0
          9.81       0  0  1  0           0  0  1  0
                     0  0  0  1           0  0  0  1
```

RRRRR = 5 Articulaciones rotacionales.
stdDH = Convención DH standard.

```
>> Brazo.plot([pi/2 pi/2 pi/2 pi/2 pi/2])
>> Brazo.teach
```



Representación en 3D del modelo del brazo robótico Lynxmotion AL5A



Interfaz Drivebot para modificar las entradas de coordenadas generalizadas

Aplicamos la función fkine para obtener la transformada homogénea de la cadena cinemática, de nuevo para los valores angulares **θ = pi/2**:

>> T = fkine(Brazo,[pi/2 pi/2 pi/2 pi/2 pi/2])

T =

-1.0000	**-0.0000**	**0.0000**	**-0.0000**
0.0000	**0.0000**	**1.0000**	**-11.2000**
-0.0000	**1.0000**	**-0.0000**	**4.6000**
0	**0**	**0**	**1.0000**

Para el caso de la cinemática inversa utilizaremos el método iterativo, que consiste en utilizar los principios de los métodos numéricos para minimizar la diferencia entre una configuración inicial del robot y la posición deseada sobre la cual se requiere determinar el vector de coordenadas generalizadas.

La configuración deseada se define a través de la matriz transformada objetivo, **T**, que determina el problema que se desea resolver. Esto quiere decir que la solución para el problema de cinemática inversa consiste en determinar de valores del vector de coordenadas generalizadas **q**, que se requiere para alcanzar la posición y orientación definidas por **T**.

Esta solución se obtendrá, por tanto, a través de un algoritmo de minimización para disminuir la diferencia entre la cinemática directa inicial y la matriz transformada objetivo.

Para plantear la solución hace falta plantear la utilización de la matriz Jacobiana **J** que contiene la contribución en velocidad de cada uno de los eslabones que participan en la cadena cinemática, donde cada columna refiere a cada una de las articulaciones. El cálculo de la velocidad integrada del actuador final se realiza a través de la multiplicación de la matriz Jacobiana **J** y el vector $\dot{\mathbf{q}}$, que representa la velocidad de cada una de las articulaciones que mueven los eslabones respectivos:

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$$

Donde $\dot{\mathbf{x}}$ es la velocidad del actuador final de la cadena cinemática.

El vector de coordenadas generalizadas **q** puede aproximarse mediante el método iterativo, paso a paso, al realizar la integración de los valores del vector de velocidad $\dot{\mathbf{q}}$ considerando la expresión de la ecuación antes enunciada de la siguiente forma:

$$\dot{\mathbf{q}} = \mathbf{J}^{-1}\dot{\mathbf{x}}$$

Para el cálculo del vector $\dot{\mathbf{x}}$ se puede considerar que será la velocidad requerida para que la cadena cinemática alcance la localización definida por la transformada objetivo **T** a partir de la posición actual, que se calcula con los valores iniciales del vector **q** . Con estos valores iniciales se calcula la matriz de cinemática directa **F(q)** como inicio del algoritmo. Teniendo en cuenta lo expuesto, el vector $\dot{\mathbf{x}}$ se podrá calcular de la siguiente forma:

$$\dot{\mathbf{x}} = \mathbf{F(q) - T}$$

Sustituyendo el valor de $\dot{\mathbf{x}}$ en la expresión dada para el cálculo de $\dot{\mathbf{q}}$:

$$\dot{\mathbf{q}} = \mathbf{J^{-1}(F(q) - T)}$$

Una vez expuesto esto, es necsario demostrar que la diferencia entre **F(q)** y **T** se minimiza en cada iteración, es decir, que la diferencia converge a cero a medida que se realizan los sucesivos procesos de integración.

Para evaluar esta solución se toma una función de costo cuadrática $\mathbf{G(\dot{q})}$. Esta función permite evaluar si una solución se acerca a cero a medida que se ejecuta cada iteración. Normalmente, estas ecuaciones de costo cuadráticas tienen la siguiente forma:

$$\mathbf{G(\dot{q})} = \dot{\mathbf{q}}^{\mathbf{T}}\mathbf{W}\dot{\mathbf{q}}$$

Donde **W** es una matriz cuadrada (nxn), siendo n el número de variables dentro del vector $\dot{\mathbf{q}}$. Esta matriz es positiva y simétrica.

Para demostrar, por tanto, la convergencia, hace falta encontrar el vector $\dot{\mathbf{q}}$ que se obtiene de la ecuación $\dot{\mathbf{q}} = \mathbf{J^{-1}\ (F(q) - T)}$ y que cumpla la minimización de costo definida en la expresión $\mathbf{G(\dot{q})} = \dot{\mathbf{q}}^{\mathbf{T}}\mathbf{W}\dot{\mathbf{q}}$.

Es necesario incluir una estimación ponderada de la calidad de la solución, a medida que avanza el algoritmo. Esta ponderación se realiza a través de la inclusión de multiplicadores de Lagrange sobre la expresión original que se va a minimizar, donde, a partir de $\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$, se tiene:

$$\mathbf{0 = J}\dot{\mathbf{q}} - \dot{\mathbf{x}}$$

Por lo que podemos plantear la nueva expresión para la función de costo como:

$$\mathbf{G}(\dot{q}, \lambda) = \dot{\mathbf{q}}^{\mathbf{T}}\mathbf{W}\dot{\mathbf{q}} - \lambda^{T}\ (\mathbf{J}\dot{\mathbf{q}} - \dot{\mathbf{x}}\ )$$

Según el cálculo integral, las condiciones necesarias para obtener una minimización pueden calcularse al igualar a cero la derivada con respecto a cada una de las variables de la función, por lo que se tiene:

$$\frac{\partial G}{\partial \dot{q}} = \mathbf{0} \qquad \text{es decir: } \mathbf{2W\dot{q} - J^T \lambda = 0}$$

$$\frac{\partial G}{\partial \dot{\lambda}} = \mathbf{0} \qquad \text{de donde : } \mathbf{J\dot{q} - \dot{x} = 0}$$

Considerando que **W** es positiva, por tanto invertible, se podrá expresar la primera condición como:

$$\dot{\mathbf{q}} = \frac{\mathbf{1}}{\mathbf{2}} \mathbf{W^{-1}\, J^T}\, \lambda$$

Sustituyendo esta expresión en la segunda condición, se tiene:

$$\mathbf{J}\left(\frac{1}{2} \mathbf{W^{-1}\, J^T}\, \lambda\right) - \dot{\mathbf{x}} = \mathbf{0}$$

O lo que es lo mismo:

$$\mathbf{J}\left(\mathbf{W^{-1}\, J^T}\right) = \mathbf{2}\, \dot{\mathbf{x}}$$

$$\mathbf{(J\, W^{-1}\, J^T)}\lambda = \mathbf{2}\, \dot{\mathbf{x}}$$

Despejando λ :

$$\lambda = \mathbf{2(J\, W^{-1}\, J^T)^{-1}}\, \dot{\mathbf{x}}$$

Y, si sustituimos este valor de lambda en la expresión que enunciamos para la primera condición:

$$\dot{\mathbf{q}} = \frac{\mathbf{1}}{\mathbf{2}} \mathbf{W^{-1}\, J^T}\, \mathbf{2(J\, W^{-1}\, J^T)^{-1}}\, \dot{\mathbf{x}}$$

Esto eliminaría los multiplicadores de Lagrange, quedando:

$$\dot{\mathbf{q}} = \mathbf{W^{-1}\, J^T\, (J\, W^{-1}\, J^T)^{-1}}\, \dot{\mathbf{x}}$$

De donde podemos corroborar que será posible minimizar la aproximación de $\dot{\mathbf{q}}$ conforme se realizan iteraciones sucesivas sobre el vector $\dot{\mathbf{x}}$ que contiene la diferencia entre **F(q)** y la matriz transformada objetivo **T**. Será además posible regresar a la expresión $\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$ si esta última ecuación se premultiplica por **J**. La minimización utiliza el factor:

$$\mathbf{J^T (J\ J^T)^{-1} \approx J^{-1}}$$

Este factor se conoce también como la seudoinversa de la matriz **J**. Si se trata de un robot con seis grados de libertad, la matriz **J** será cuadrada y podrá utilizarse la inversa de dicha matriz. En robots de menos grados de libertad, el algoritmo utiliza el cálculo de la seudoinversa para minimizar de forma óptima la diferencia **F(q) – T**.

Trasladando el problema a Matlab, el algoritmo empezará definiendo una postura inicial tomando como base los valores actuales del vector de coordenadas generalizadas $\mathbf{q_0}$, calculando la cinemática directa **F(q)**. Esta operación devolverá una matriz homogénea que describe la orientación y posición del actuador como un valor inicial para el algoritmo.

A partir de **F(q)**, es posible definir la diferencia **F(q) – T**, que se obtiene como un argumento de entrada al comando. De esta forma, es posible calcular la diferencia entre ambas transformadas, con respecto a la matriz de rotación y al vector de posición.

El método iterativo implementa la solución a la ecuación $\dot{\mathbf{q}} = \mathbf{J^{-1}(F(q) - T)}$ a través de la integración sucesiva del vector $\dot{\mathbf{q}}$. Dicha solución converge hasta minimizar la diferencia entre **F(q)** y **T** como se ha demostrado anteriormente.

Según las indicaciones dadas por **Peter Corke** en su manual para la **Robotics Toolbox** referentes al uso de la función ikine, que nos realizará los cálculos de la cinemática inversa, en el caso donde el manipulador tiene menos de 6 grados de libertad, se debe de emplear la expresión

ik = Brazo.ikine(T,q_0,m,options)

Donde **T** es la matriz transformada homogénea que representa la localización y orientación a la que deseamos llevar nuestro actuador final , $\mathbf{q_0}$ es un argumento opcional que define el vector de un punto de inicio conveniente para la búsqueda del q requerido que devolverá la función ikine, **m** será un vector máscara (1X6) que especifíca un grado de libertad cartesiano (en el eje de coordenadas de la muñeca) que será ignorado para alcanzar una solución. Este vector tiene seis elementos que corresponden a la traslación en **X,Y,Z** y rotación en **X,Y,Z** respectivamente. Los valores deben ser 0 (para ignorar) o 1. El número de elementos distintos de cero

debe de ser igual al número de grados de libertad del manipulador. En nuestro caso, con 5 grados de libertad este método es difícil de implementar porque la orientación específicada en **T** es en coordenadas cartesianas y las orientaciones posibles son en función de la posición de la pinza, o herramienta que constituya el actuador final. Teóricamente deberíamos utilizar un vector m con los siguientes valores: **[1 1 1 1 1 0]**, es decir, debemos de incluir en el vector un 1 por cada grado de libertad y dejar el restante valor a cero.

En cualquier caso, la función **ikine** devuelve las coordenadas de las articulaciones correspondientes a cada una de las transformadas en la cadena, como hemos comentado antes, será el vector q requerido para situar el brazo en la posición deseada.

El argumento **options** nos ofrece las siguientes posibilidades:

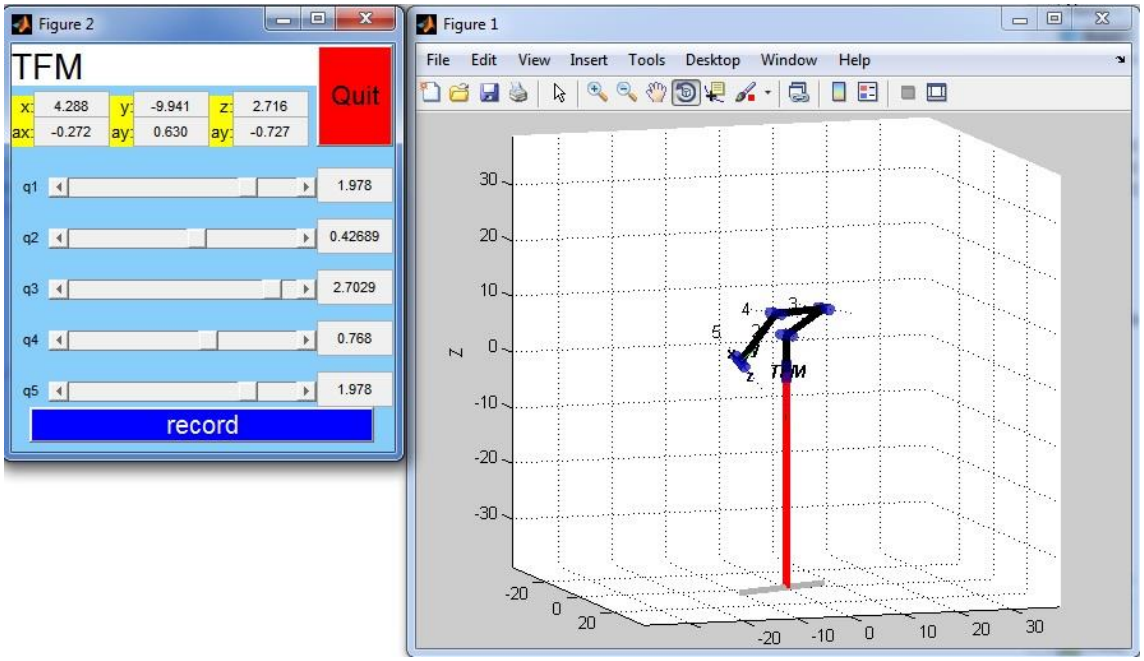
`'pinv'`	use pseudo-inverse instead of Jacobian transpose (default)
`'ilimit', L`	set the maximum iteration count (default 1000)
`'tol', T`	set the tolerance on error norm (default 1e-6)
`'alpha', A`	set step size gain (default 1)
`'varstep'`	enable variable step size if pinv is false
`'verbose'`	show number of iterations for each point
`'verbose=2'`	show state at each iteration
`'plot'`	plot iteration state versus time

Utilizaremos la opción '**pinv**', porque según indicaciones del propio Corke, es la opción que lleva a una convergencia de forma más rápida.

El uso de esta función nos plantea una serie de circunstancias que es necesario tener en cuenta:

- La solución es generalmente no única, como antes habíamos comentado, y es dependiente del valor $\mathbf{q_0}$ introducido, que por defecto es con todos los valores a 0.
- El valor defecto de $\mathbf{q_0}$ no es apropiado para la mayor parte de manipuladores.
- Los límites de las articulaciones no son tenidos en cuenta para esta solución.

Establecemos una posición deseada a la que queremos llevar el actuador final:



Los valores de **x, y, z** que vamos a introducir serán 4.288, -9.941 y 2.716. Creamos una matriz homogénea con la posición deseada:

deseada =

-1.0000	**0**	**0**	**4.2880**
0	**1.000**	**0**	**-9.9410**
0	**0**	**1.0000**	**2.7160**
0	**0**	**0**	**1.0000**

Como no es recomendable utilizar los valores de $\mathbf{q_0}$ por defecto y se recomienda utilizar los valores de coordenadas generalizadas cercano a los de la posición deseada, creamos un vector con los siguientes valores:

>> qi = [1.7 1 1.8 1 1.2]

qi =

1.7000 1.0000 1.8000 1.0000 1.2000

Y, finalmente, utilizamos el comando *ikine* con los valores adecuados:

>> ik = Brazo.ikine(deseada, qi, m, 'pinv')

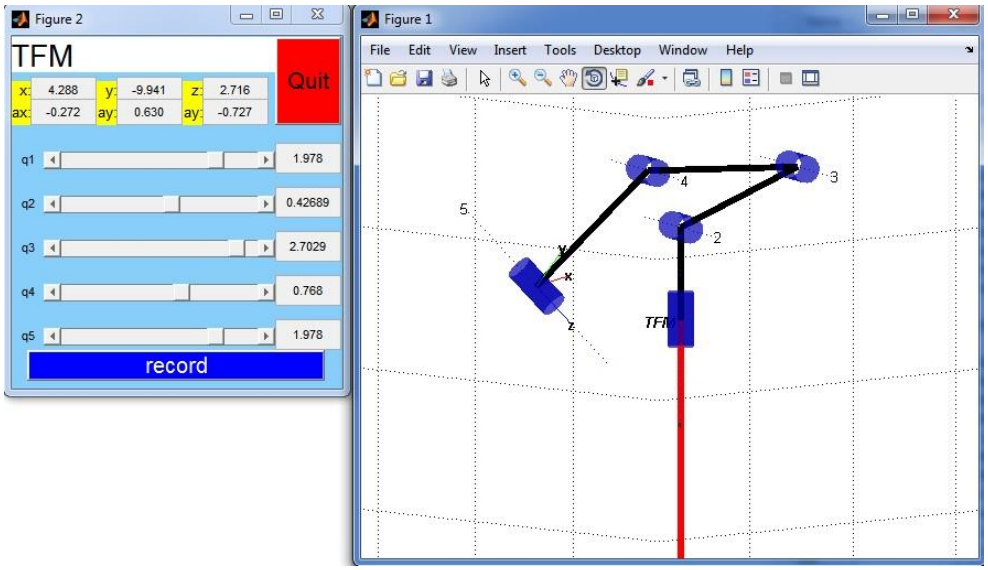
ik =

1.9780 0.4269 2.7029 0.7680 1.9780

Como podemos comprobar, comparando con los valores mostrados en el "**drivebot**", los valores obtenidos son aceptablemente similares. Creamos la gráfica para comprobar que se encuentra en la posición deseada:

>> Brazo.plot(ik)

>> Brazo.teach



Para representar el espacio de trabajo, que describirá todas las posiciones posibles que nuestro robot puede alcanzar con su actuador final

tomamos primero el rango en el que trabajan sus articulaciones. Estas son las siguientes:

θ_1 = 0 – 180º

θ_2 = 60 - 123º

θ_3 = 0 – 90º

θ_4 = 0 – 90º

θ_5 no afectará para considerar la posición del actuador final, así que no ha sido tenida en cuenta.

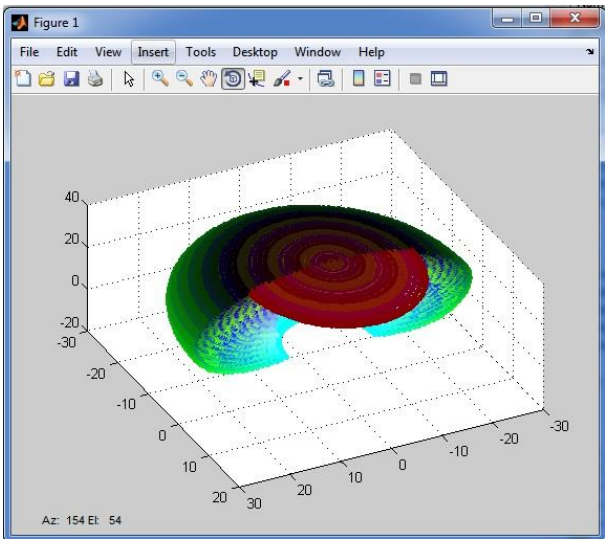
Para obtener el espacio de trabajo iteraremos cada articulación a través de su rango de movimiento. Haciendo este procedimiento con todas las articulaciones podremos obtener todas las posiciones posibles en las que podremos tener nuestro brazo. Utilizaremos la cinemática directa calculada anteriormente para tal efecto. El código empleado ha sido el siguiente:

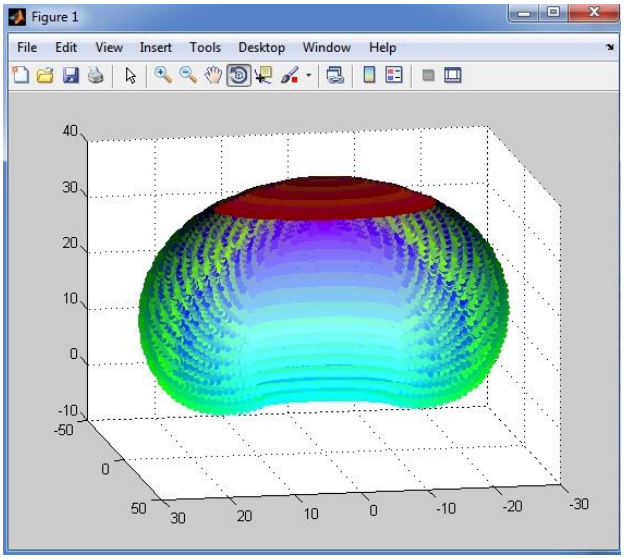
```
>> X = [];

>> Y = [];

>> Z = [];

>> C = [],[];


C =


    []

>> for i = 0:2:180,

fprintf ('en iteración: %d \n', i)

for j = 60:10:123,

for k = 0:10:90,

for l = 0:5:90,

theta1 = i   * pi/180;

theta2 = j   * pi/180;

theta3 = k   * pi/180;
```

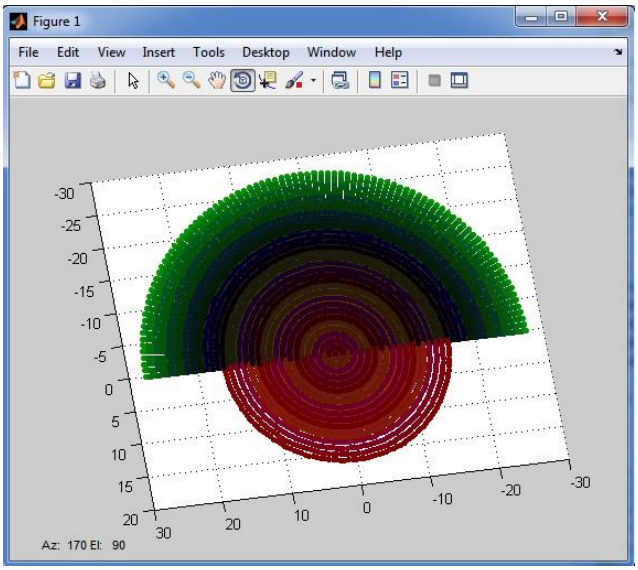
```
theta4 = l   * pi/180;

T = fkine(Brazo,[theta1 theta2 theta3 theta4 0]);

C(end +1,1) = 1 -(j/123);  // Para variar los colores en función del valor

C(end,2) = (k/90);

C(end,3) = (l/90);

X(end+1) = T(1,4);

Y(end+1) = T(2,4);

Z(end+1) = T(3,4);

end

end

end

end
```

```
>> scatter3(X,Y,Z,20,C,'filled');
```

El resultado:







7.2.4. CALIBRACIÓN DE SERVOS

Es necesario realizar una calibración de los servos para asegurar un desplazamiento preciso de forma que si se le entrega la orden a nuestro brazo para que se coloque en unas coordenadas (x,y,x) la pinza se coloque en esa posición con un margen de error aceptable.

El procedimiento que vamos a seguir para realizar esta calibración será el siguiente:

- Para la rotación de la base y para el movimiento de alzado y abatimiento del servo de la muñeca obtendremos varios ángulos de respuesta para valores de ancho de pulso introducidos y calcularemos

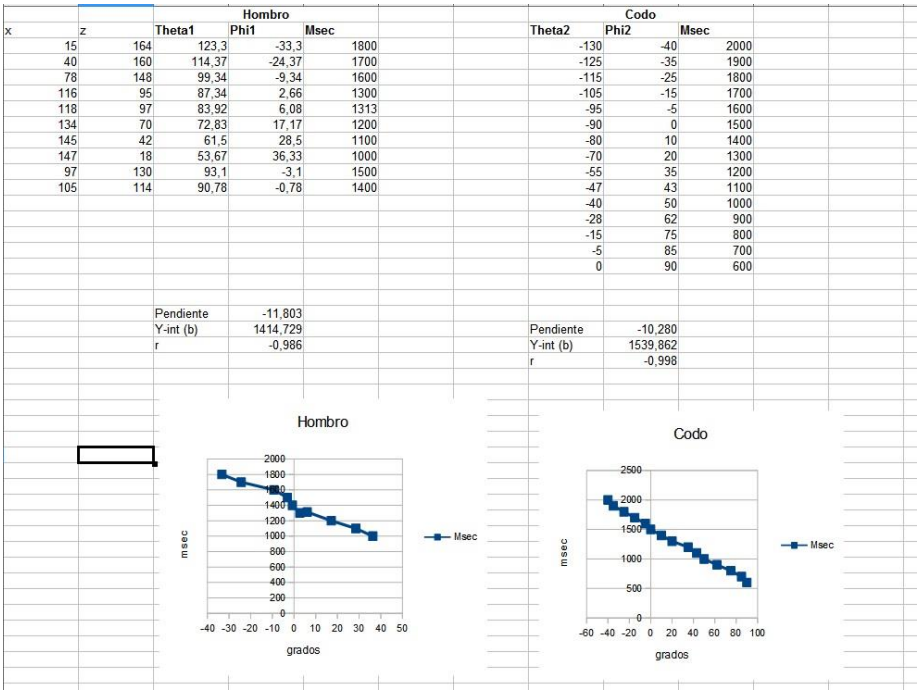
la pendiente de una línea de regresión representada con estos parámetros.

- Para la pinza aplicaremos varias aperturas y cierres según el ancho de pulso introducido y obtendremos una pendiente.
- Para el hombro y el codo moveremos el brazo a varias localizaciones (x,y) en la superficie de un papel que utilizaremos como plantilla y resolveremos las ecuaciones de cinemática inversa para esos valores obteniendo unos ángulos. En una hoja de cálculo introduciremos esos valores de ángulo y sus anchos de pulso correspondientes y calcularemos la pendiente.
- A continuación, cuando tengamos definidas todas las pendientes, que serán factores por los que tengamos que multiplicar los anchos de pulso podremos definir los ángulos de las posiciones definitivas siguiendo la siguiente expresión:

Ángulo = pendiente * AnchoPulso + Posición neutra

Donde el ángulo será el ángulo de la articulación en grados y el ancho de pulso será el que obtenemos después de resolver la cinemática inversa para generar el movimiento que lleve a las articulaciones a la posición deseada en microsegundos.

Para la obtención del valor de la pendiente se ha utilizado una hoja de cálculo modificada a partir de la utilizada en el proyecto tic-tac-toe.



Muestra de la hoja de cálculo empleada para el cálculo de la pendiente

7.2.5. COMUNICACIÓN

Es posible utilizar la interfaz de monitor serie incorporada en el entorno de programación de A**rduino** para realizar pruebas de funcionamiento y comprobar que el código creado en la **IDE** responde de la manera que deseamos al recibir datos en serie. Pero, si necesitamos comunicar el programa que hemos estado desarrollando en C++ en **Visual Studio** y **Qt**, necesitaremos emplear una de las librerías disponibles para tal efecto. En este caso, para el programa desarrollado en **Visual Studio** (Al portar el código desarrollado a **Qt** hemos tenido que utilizar la librería incoporada en el entorno de dicho programa, aunque esto será comentado más adelante), tras haber hecho pruebas con una serie de librerías, tales como la "**Tserial.h**", se han obtenido resultados satisfactorios con la librería "**SerialClass.h**", la cual se puede descargar de forma gratuita en un repositorio de GitHub, alojado en esta dirección:

https://github.com/Gmatarrubia/LibreriasTutoriales

Para poder servirnos de sus funcionalidades debemos primero agregarla a nuestro proyecto, tanto el archivo de cabecera "**SerialClass.h**", como el archivo de código fuente de C++ "**Serial.cpp**".

Para poder construir el objeto de comunicación, que llamaremos Arduino:

```
Serial* Arduino
```

Necesitamos conocer el puerto serial al que está conectado nuestra placa Arduino. En este caso podemos comprobar este dato en la IDE de Arduino. Para eso entraremos en el menú Herramientas y accederemos al apartado "Puerto serial". En nuestro caso, al ser el puerto indicado el "**COM6**", así lo especificaremos en nuestro código:

```
Serial* Arduino = new Serial("COM6");
```

Comprobaremos que estamos conectados mediante la función **Arduino->isConnected()**, que devolverá true en caso afirmativo.

El uso de las funciones de lectura y escritura de datos es muy sencillo. Tan sólo debemos crear un buffer de entrada y uno de salida y utilizar respectivamente las órdenes **Arduino->ReadData**(buffer de entrada,longitud de este) y **Arduino->WriteData**(buffer de salida, longitud de este).

Para abrir la comunicación en nuestro código de **Arduino** daremos la orden **Serial.begin(9600)**, con el valor entre paréntesis representando la velocidad de transmisión de datos. Después crearemos un condicional que asegure que mientras exista comunicación se realicen las operaciones que consideremos necesarias. En nuestro caso estamos interesados en que se

realice una única transmisión de datos, usaremos while Serial == true y, cuando terminemos de realizar el intercambio de datos entre **Arduino** y **Visual Studio**, cambiaremos el estado de serial a false, deteniendo así la comunicación. Para asegurarnos que sólo se va a ejecutar este código una vez en este bucle utilizaremos la siguiente construcción:

```
static uint64_tc; // Se ejecutará siempre

if (c++ = = 0)

{

Código que se ejecutará una única ocasión

}
```

resto de código que se ejecutará con normalidad en cada iteración.

Con **static** podemos declarar la variable dentro del bucle. El código encapsulado en los corchetes posteriores al **if** se ejecutará sólo una vez, ya que cuando incremente el valor de c no cumplirá la condición. Cuando terminemos el proceso de comunicación indicaremos que dejamos de utilizar el objeto serial creado con **delete** + nombre del objeto: **delete Arduino**, en nuestro caso.

Para poder recibir los valores de las coordenadas **x** e **y** y los ángulos, tras haber realizado la detección, crearemos un buffer de entrada, como se ha comentado previamente:

En Visual Studio:

palabra[ ]

y lo iremos llenando con los valores que vamos recibiendo:

```
int cx;

cx = sprintf(palabra,"%d",punto.x);

int cx2;

cx2 = sprintf(palabra+cx,",%d",punto.y);

int cx3;

cx3 = sprintf(palabra+cx+cx2,",%d",.....
```

y así sucesivamente. Nótese que se van dejando comas entre los valores, que nos serán útiles para poder asignar en el código de Arduino dichos

datos a las variables que necesitamos para realizar los movimientos del brazo. Este procedimiento lo podremos realizar gracias a la función **parseFloat()**;

En Arduino:

```
while (Serial.available () >0) //Mientras haya datos en el puerto serial

{

float coordx = Serial.parseFloat();

float coordy = Serial.parseFloat();

if (Serial.read() == '/n')

{

pos_brazo(coordx,coordy,z....

....
```

con **parseFloat()** lo que haremos será extraer los valores numéricos que hay entre otros caracteres, que en este caso serán las comas que habíamos introducido anteriormente.

Cuando se reciba el carácter **/n**, se procederá a ejecutar la función creada para mover el brazo robótico (**pos_brazo**).

8. MÓDULO 3: DESARROLLO DE INTERFAZ GRÁFICA DE USUARIO (GUI)

8.1. DESCRIPCIÓN GENERAL. FUNCIONAMIENTO

Tras haber desarrollado en **Visual Studio** los códigos que nos posibilitarán realizar todas las operaciones deseadas con nuestro brazo robótico, se plantea ahora el diseño y construcción de una interfaz gráfica de usuario (**GUI**) que facilite al usuario la interacción con dicho código de una manera cómoda e intuitiva. Esta supondrá una mediación entre el usuario y el código realizado para el proyecto que permita su manejo mediante widgets interactivos, textos e imágenes.

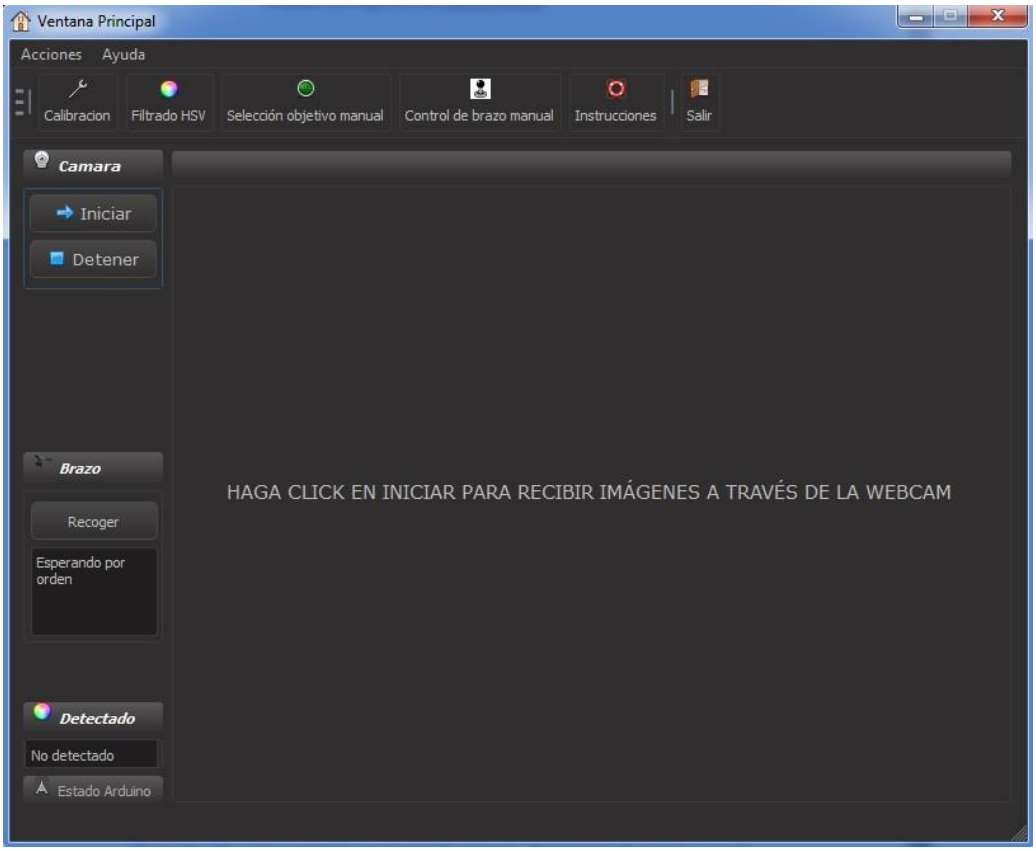
Se han propuesto los siguientes objetivos para la realización de dicha **GUI** :

- Facilidad de navegación.
- Uso intuitivo de las operaciones disponibles. Presencia de descripciones e instrucciones para poder realizar estas y evitar confusiones.
- Ofrecer al usuario la mayor cantidad de información útil posible y que esta se encuentre presente en pantalla o que sea accesible de una manera simple y correctamente indicada.
- Presencia de una ventana principal desde la que se pueda acceder a otras ventanas en las que se desarrollan unas operaciones específicas y a la que se regrese al cerrar estas últimas. La función de esta ventana principal, aparte de menú de inicio, será la de manejo / monitorización de la operación principal de recogida realizada por nuestro robot.
- Buen diseño general y uso de gráficos e iconos para crear una estética agradable y, en el caso de los iconos, para ofrecer información visual añadida.

El esquema de funcionamiento de la aplicación diseñada será como se indica en el siguiente gráfico:

Se accederá a la **GUI** a través de una ventana de bienvenida desde la que se redireccionará a la ventana-menú principal. Desde esta ventana se podrán realizar todas las operaciones disponibles. Presentes en la ventana principal estarán un menú desplegable en la parte superior y una barra de herramientas que se podrá desplazar si se desea. El widget central de esta ventana será un campo en el que se visualizarán las imágenes recibidas por la cámara web instalada en nuestro prototipo. También se dispondrán botones que permitan el inicio de la recepción de dichas imágenes y su detención. Una ventana de información en la esquina inferior izquierda nos indicará si se ha realizado una detección y si existe una conexión abierta con la tarjeta controladora del brazo robótico. También se informará del estado del brazo.

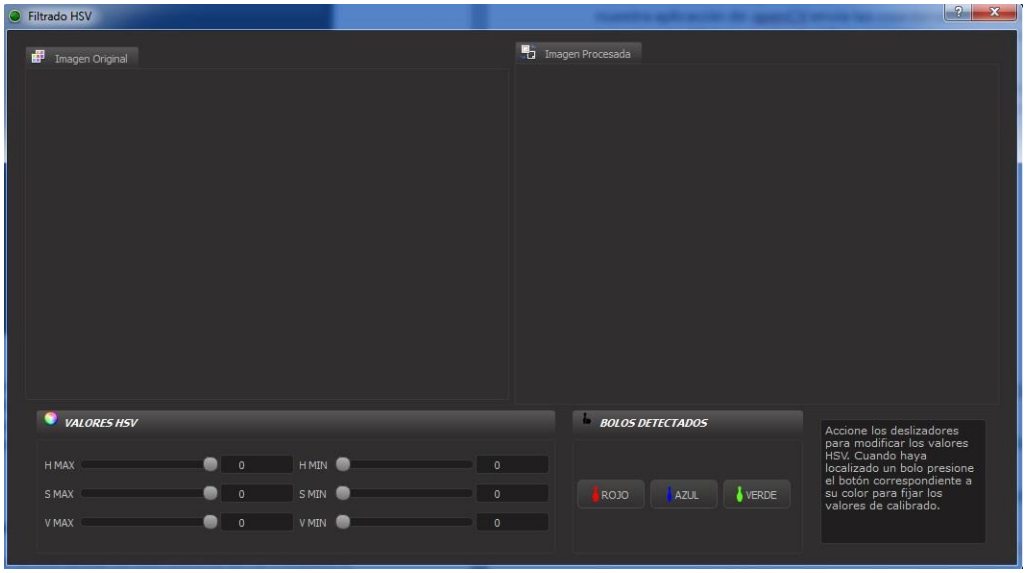
Ventana de bienvenida



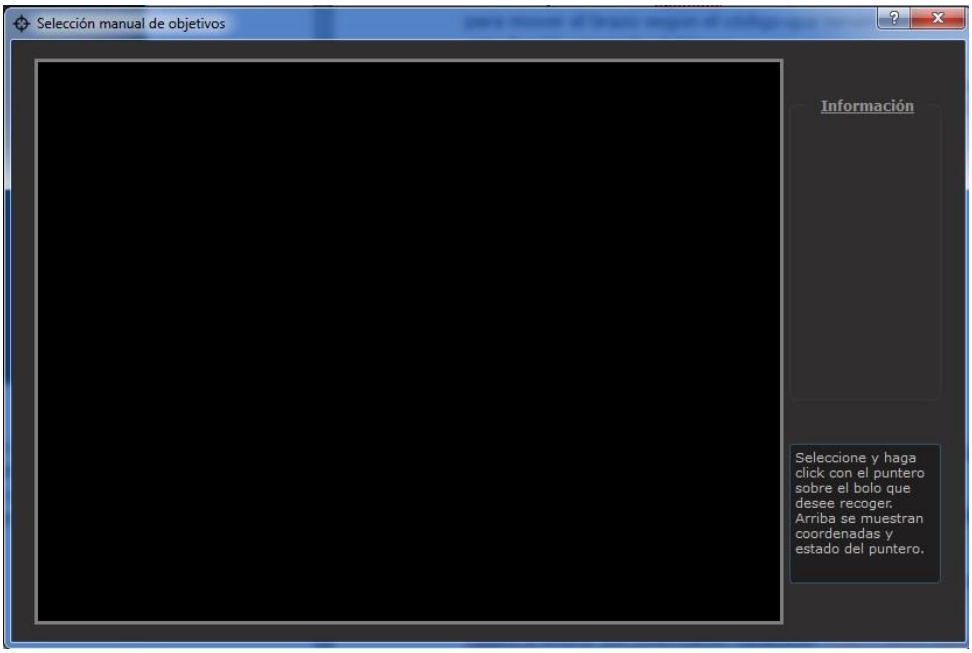
Ventana principal

Las posibilidades que ofrece nuestra aplicación serán la de realizar el calibrado de la cámara, selección de los valores de filtrado **HSV** que serán

empleados para la detección de los objetivos a recoger, control manual del brazo robótico mediante deslizadores y selección de objetivos a recoger mediante el ratón. Las opciones de calibrado y de filtrado de **HSV** devolverán valores a la ventana principal que serán utilizados para la configuración general de la aplicación, mientras que las opciones de control manual del brazo y de selección manual de objetivos funcionarán como aplicaciones independientes.



Ventana de ajuste de filtrado HSV



Ventana de selección manual de objetivos

En la ventana de ajuste de filtrado se mostrará a la izquierda la imagen recibida por la webcam y a la derecha la imagen tras el procesamiento de

filtrado. Se disponen seis controles deslizadores en la parte inferior mediante los cuales se introducen los valores de **HSV** deseados. Cuando hayamos realizado una detección adecuada se pulsará uno de los tres botones dispuestos en función del color del bolo detectado. Esta acción nos llevará a la ventana principal, donde se actualizará el campo habilitado para mostrar si se ha producido una detección y se realizará el seguimiento (tracking) del objeto detectado. En la esquina inferior derecha se dispone de una guía de utilización de esta rutina.

Para la selección manual de objetivos mediante el puntero se dispone de un widget central en el que se mostrará la recepción de imágenes de la webcam y donde se podrá hacer clic para seleccionar el objetivo a recoger. A la derecha se mostrará, en el cuadro marcado como "Información" , las coordenadas del puntero y el estado de los botones del ratón (pulsado, navegando o fuera de ventana). Al igual que en la ventana de ajuste de filtrado, se dispone de un pequeño cuadro informativo con las instrucciones para realizar esta operación.

Ventana para control manual de articulaciones del robot

8.2. DESARROLLO DE LA INTERFAZ

8.2.1. INTRODUCCIÓN

El desarrollo de la **GUI** se ha realizado con **Qt**. Esta es una plataforma de código abierto para desarrollo que contiene una serie de librerías gráficas, basadas en lenguaje de programación C++ de forma nativa, aunque ofrece la posibilidad de ser utilizado con **Pyton** y otros a través de bindings. Es ampliamente utilizada para desarrollo de interfaces gráficas, así como en campos como automoción, aeronavegación y aparatos domésticos.

Inicialmente fue desarrollado por la empresa noruega **Trolltech**, después fue adquirido por **Nokia** y finalmente sus librerías fueron licenciadas bajo **GPL**.

Se ha escogido esta plataforma por ofrecer las siguientes ventajas:

- Es gratuito
- Facilidad de uso
- Al ser de código abierto y de uso muy extendido dispone de una comunidad de usuarios que contribuyen a su mejora continua.
- El uso de un editor visual nos permite además de realizar el diseño de las UI de una forma más cómoda y rápida, sin tener que escribir líneas de código y realizar ajustes constantes.
- Es multi-plataforma, lo que nos permitirá desarrollar aplicaciones compatibles con Mac, Windows, Symbian, IOS o Android.
- Utiliza lenguaje de programación C++ y aporta además una serie de funciones y comandos que facilitan mucho la creación de código.
- Se dispone de una gran cantidad de documentación en su sitio web, así como de una considerable cantidad de publicaciones y libros de referencia.

8.2.2. INSTALACIÓN DE Qt Y OPENCV

Para su instalación acudimos a la sección de descargas de su sitio web y descargamos el instalador para la versión **5.1.0** (actualmente hay nuevas versiones disponibles)

Para su instalación acudimos a la sección de descargas de su sitio web y descargamos el instalador para la versión **5.1.0** (actualmente hay nuevas versiones disponibles)

http://www.qt.io/download/

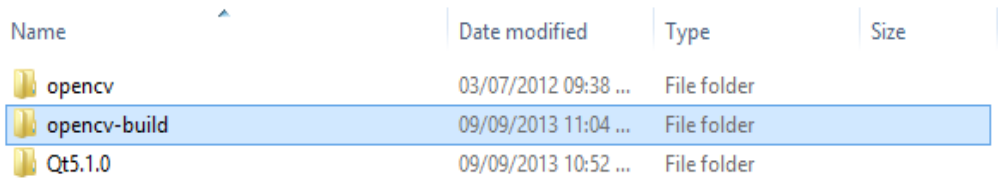
Descomprimimos los archivos en el directorio deseado para la instalación.

Como pretendemos utilizar la librería **OpenCV** debemos preparar **Qt** para poder acceder a sus contenidos. Para eso debemos de realizar una instalación con **Cmake**.

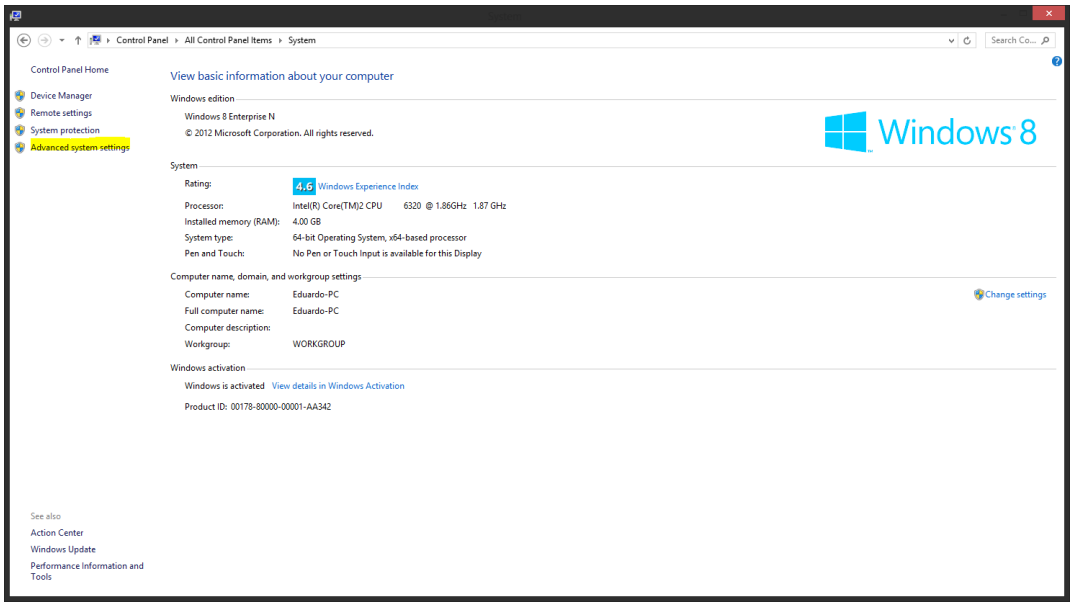
Obviamente debemos disponer de **Cmake** en nuestro equipo para realizar la instalación. De no ser así, podemos descargar la versión **2.8** en su sitio web: http://www.cmake.org/

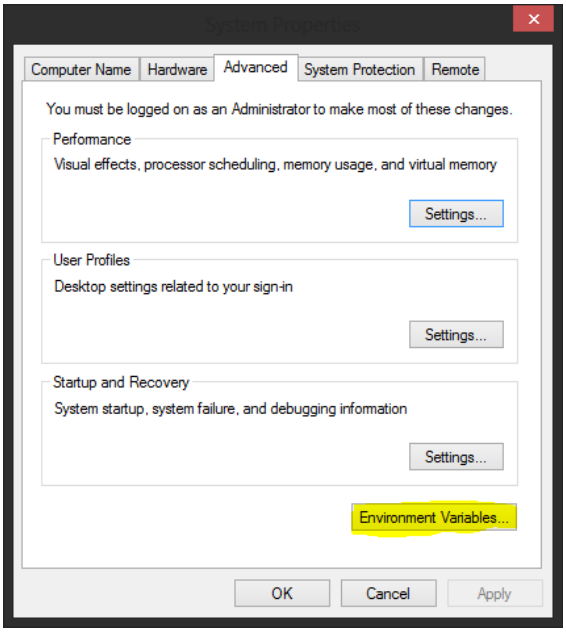
Para instalar **OpenCV**, accedemos a su sitio http://opencv.org/ y descargamos el archivo comprimido con la versión más actual.

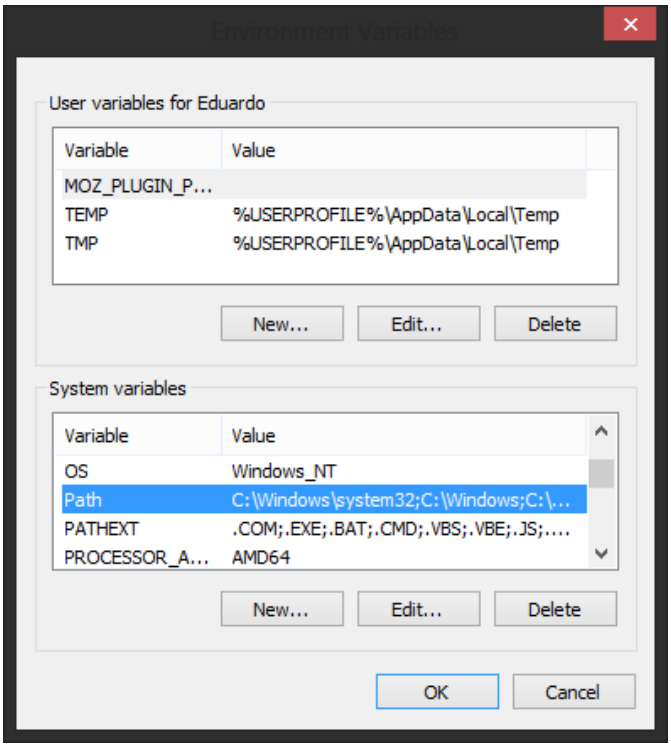
Seleccionaremos la carpeta de **Qt** instalada previamente y descomprimiremos el contenido del archivo .rar en su interior. Crearemos una carpeta nueva en Qt que llamaremos opencv-build .



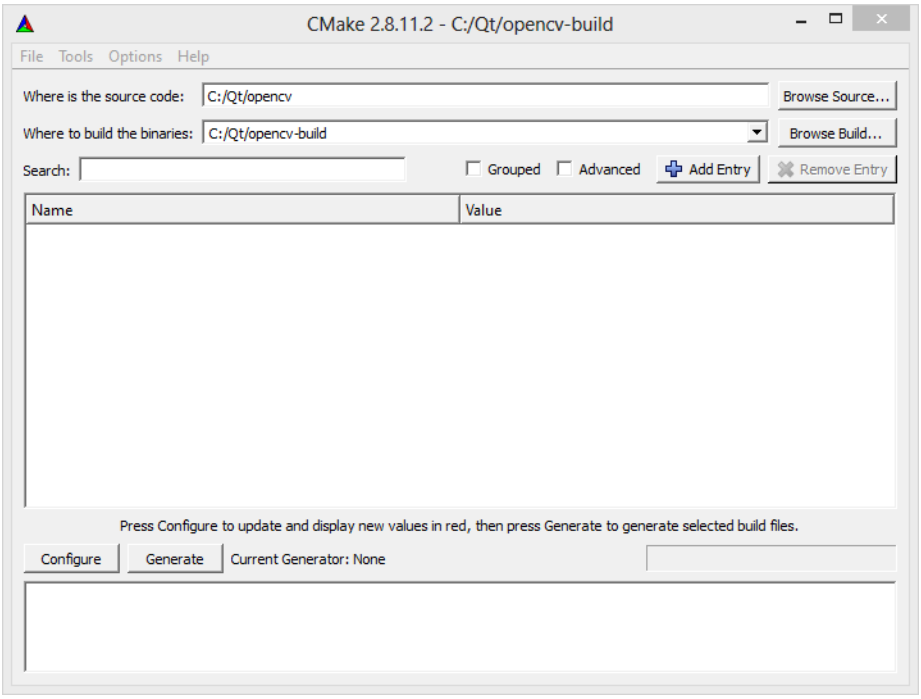
El siguiente paso sería añadir **OpenCV** a la dirección **PATH** del entorno. Para ello acudiremos al panel de control de nuestro equipo y una vez dentro accederemos a información de sistema. Dentro de sistema abrimos las opciones de configuración avanzada y a continuación la pestaña de variables de entorno. Una vez allí encontramos en **PATH** las direcciones de **Cmake**, **OpenCV** y **Qt** que haya escritas y las modificamos para que estén de acuerdo con su ubicación actual.



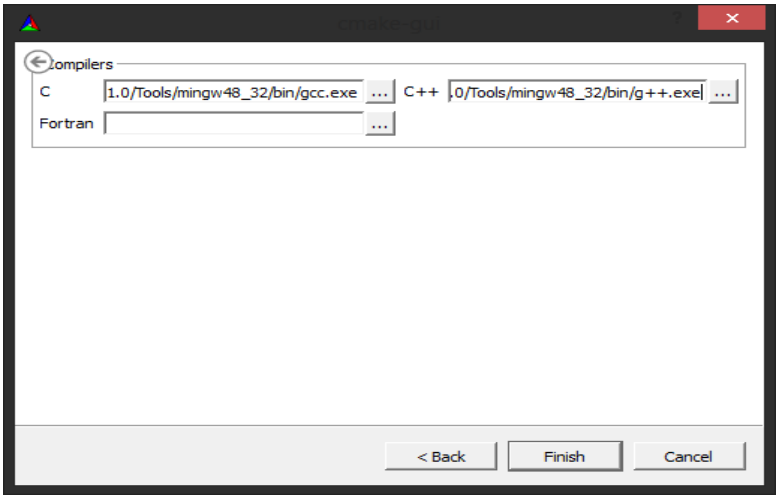




Ejecutamos a continuación **CMake** y seleccionamos como raiz la carpeta de **OpenCV** extraída en **Qt** y como objetivo la nueva carpeta opencv-build que hemos creado previamente.

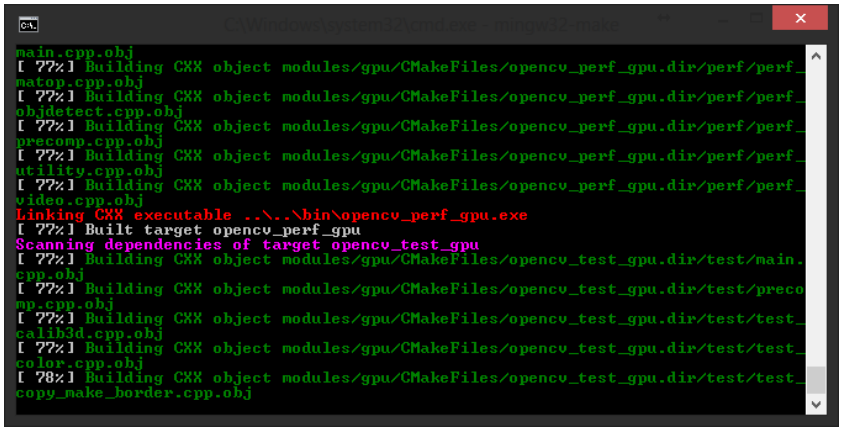


Se pulsa en Configurar y seleccionaremos **MinGW** como generador. Se selecciona también **MinGW** (se encuentra en la carpeta Tools de la instalación de **Qt**. Es un archivo **gcc.exe** para compilar **C** y otro **g++.exe** para compilar **C++**) como compilador. Una vez terminado el proceso se selecciona build_docs y with **Qt** y volvemos a pulsar Configurar.

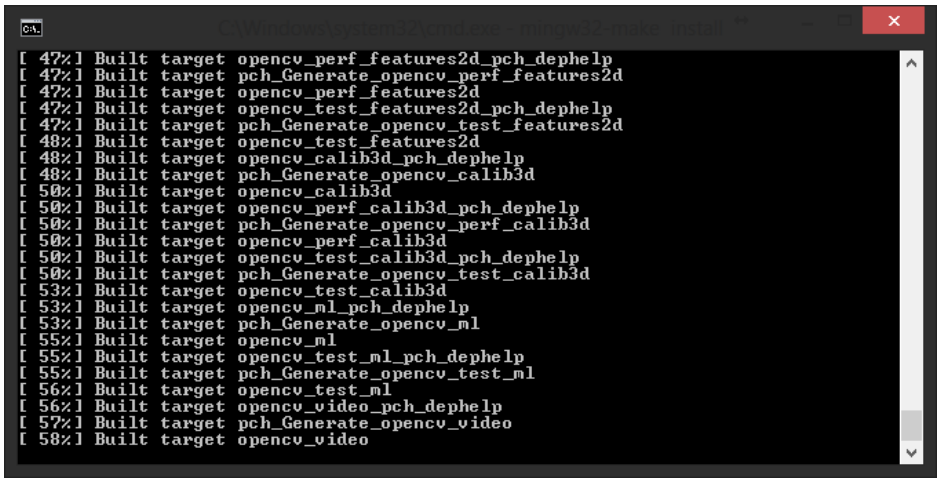


Si durante este segundo proceso se pide introducir la dirección de qmake.exe, se debe de indicar su localización, que será dentro de la carpeta mingw48_32\bin del directorio de **Qt**.

En una consola de comandos (accederemos tecleando cmd en búsqueda en el menú de inicio de Windows) se debe de entrar en la carpeta **opencv-build** e introducir el siguiente comando: **ming32-make**.



En cuanto se termine de realizar las operaciones de crear los instaladores introduciremos el siguiente comando: **ming32-make install**.



Al término de este proceso, ya deberíamos tener instalada la librería **OpenCV** en el entorno de **Qt**. Aún así, debemos de añadir en nuestro archivo de proyecto las siguientes líneas:

```
INCLUDEPATH += C:\Qt\opencv-build\install\include

LIBS += C:\Qt\opencv-build\bin\libopencv_core242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_contrib242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_highgui242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_imgproc242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_calib3d242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_features2d242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_flann242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_legacy242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_ml242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_objdetect242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_video242.dll
LIBS += C:\Qt\opencv-build\bin\libopencv_gpu242.dll
```

8.2.3. ENTORNO DE DESARROLLO DE Qt

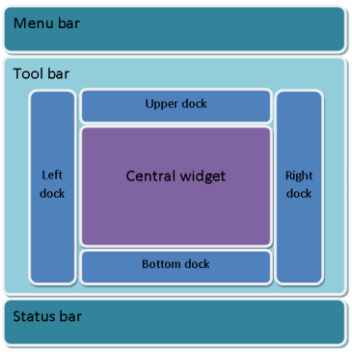
La herramienta utilizada para el el desarrollo de la interfaz es el **Qt Creator**, que viene incluído en el paquete de descarga de **Qt**.

Esta **IDE** provee una serie de características que facilitan mucho las tareas de desarrollo:

- Editor de código con soporte de **C++**, **QML** y **ECMA**.
- Herramientas de navegación de acceso rápido.
- Función de autocompletar código de forma predictiva y resalte de sintaxis con diferentes colores que facilitan la comprensión de este y las correcciones.
- Asistente para cierre de paréntesis y símbolos.
- Debugger con inclusión de interrupciones, puntos de ruptura y depurado línea a línea.
-

Cuando se crea un proyecto, en nuestro caso una aplicación **GUI**, se nos pide escoger una ubicación para los archivos y para la carpeta que contendrá los archivos de depuración y el ejecutable creado, también escoger el compilador que deseemos, la clase de ventana que vamos a crear, que puede ser de las siguientes bases:

- **QMainWindow** : Será la ventana principal para un proyecto. Dispone de menús desplegables, barras de herramientas y un widget central que será el contenedor del principal elemento de la aplicación. No será posible eliminar ni deshabilitar este widget central. Las librerías que gestionan los elementos de esta ventana principal son **QToolBar**, **QMenu**, **QMenuBar**, **QWidget** y **QObject**.



Disposición de elementos de una ventana principal

- **QWidget**: La clase **QWidget** es la base de todos los objetos de interfaz de usuario. Es la parte principal que asegura la interacción con el código: Recibirá eventos a través de cualquiera de los posibles

dispositivos de entrada (ratón, teclado…) y tendrá una representación visual en pantalla. Cuando un widget no está embebido en una ventana o widget padre se le considera ventana de tipo **QWidget**.

- **QDialog**: Es la clase básica de ventanas de diálogo, que será designada a tareas cortas y secundarias. Pueden proveer un valor de retorno (en nuestro caso así será y se comentará en un apartado posterior)

Una vez configuradas estas opciones se crean cuatro archivos: Uno de proyecto **(.pro)**, uno de cabecera **(.h)**, uno de fuente **(.cpp)** y una forma **(.ui)**

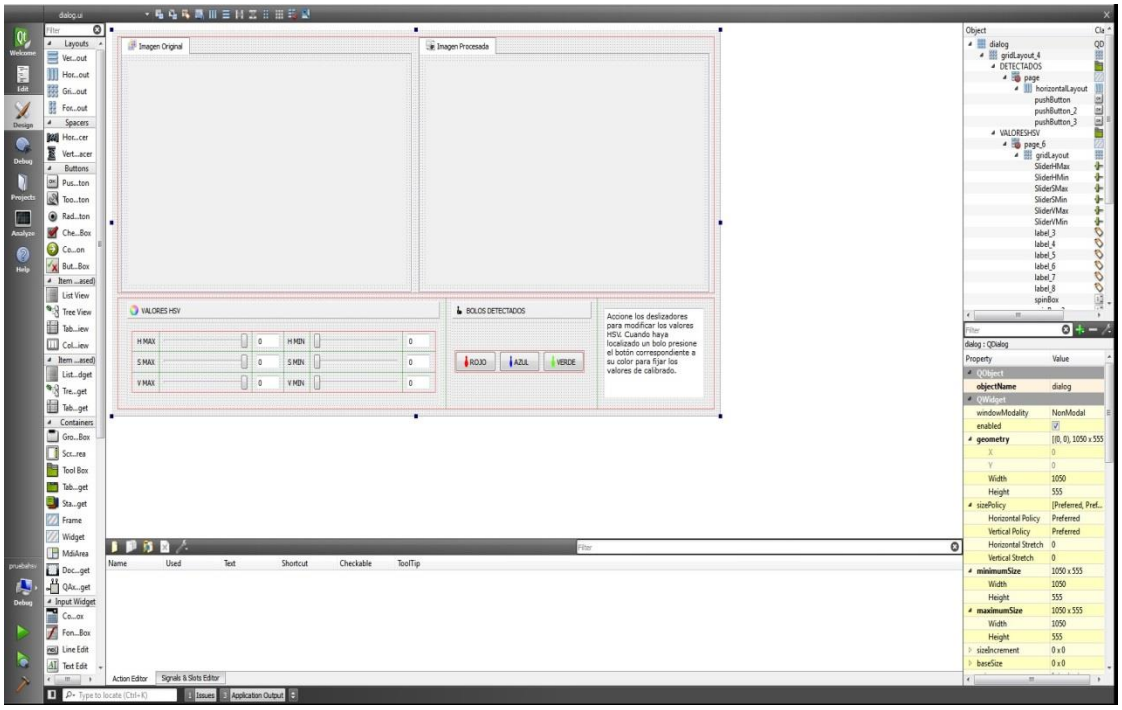
El archivo de proyecto **(.pro)** contiene toda la información que requiere **qmake** (herramienta que genera las Makefiles) para construir la aplicación, librería o plugin que estemos desarrollando.

El archivo de cabecera **(.h)** será como el utilizado normalmente en programación **C++**. Contendrá declaraciones de clases, subrutinas y variables. El código fuente **(.cpp)** es un archivo de rutinas, funciones y operaciones de programación en lenguaje **C++**.

La forma será una disposición de objetos con representación visual en la pantalla, que asegurarán la interacción del usuario con el programa y facilitarán el uso de las rutinas programadas.

Una de las mejores herramientas que se pueden utilizar durante el desarrollo mediante **Qt** es el diseñador de disposición (layout) de la **GUI**. Cuando se crea una aplicación de **Qt** se ofrece la posibilidad de crear una forma **.ui**, aunque también se pueden incluir a nuestro proyecto en cualquier momento.

Con este diseñador podremos crear widgets y dialogs utilizando una de las muchas plantillas de las que se disponen y hacerlas plenamente funcionales asignándoles slots para crear interacción con nuestro código.



En la imagen superior se puede apreciar una captura del diseñador de **GUI** mostrando la forma creada para la ventana de diálogo de filtrado de **HSV**. A la izquierda está ubicada el menú de widgets que podemos introducir en nuestro diseño, en el centro está la ventana en la que configuraremos la disposición de todos los componentes que vayamos aplicando. Posteriormente podremos aplicar relaciones entre ellos para asegurar que se presentan de una manera ordenada y coherente en pantalla. En nuestro caso hemos ido creando relaciones de coincidencias verticales, horizontales y posteriormente en toda la ventana se ha incluido una disposición de cuadrícula. Y se han fijado los tamaños de todos los widgets para que no haya problemas de visualización.

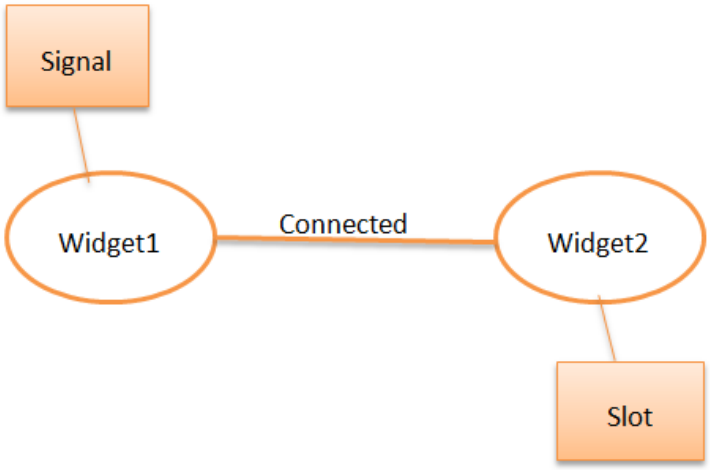
Bajo el marco principal se coloca el editor de acciones, con el que podremos comprobar y asignar acciones a los botones, deslizadores y demás widgets que se vayan colocando en el diseño .

A la derecha tendremos dispuesto el inspector de objetos, que no es más de una lista de todos los componentes del diseño y sus relaciones con respecto a otros objetos. Finalmente, bajo este encontramos el inspector de propiedades, en la que podemos modificar una serie de parámetros de los objetos y de la ventana en general, como pueden ser nombre, dimensiones, iconos, plantilla de estilo…

Por último es necesario señalar una característica muy importante de Qt y esta es el mecanismo de **Signals & Slots** . Este sistema se utiliza para establecer comunicación entre objetos.

Cuando interactuamos con un widget, a menudo deseamos que otro widget sea notificado. De esta forma podemos abrir o cerrar una ventana o llamar a una función determinada con un clic en un botón de la interfaz, por ejemplo. En otros frameworks se utilizan normalmente callbacks, que son punteros a una función, pero estos son un tipo de comunicación menos segura.

Signal es una señal, que será emitida cuando suceda un evento concreto y **Slot** es una función que será llamada en respuesta a dicha señal.



8.2.4. ADAPTACIÓN DEL CÓDIGO EXISTENTE

Para utilizar el código que se había desarrollado previamente en el entorno de **Visual Studio** ha sido necesario realizar una serie de modificaciones para poder utilizar librerías de **Qt** que nos permitiesen realizar ciertas tareas que en la interfaz creada.

Para mostrar, por ejemplo, las imágenes recogidas por la webcam es necesario crear una función (en nuestro caso *ProcesaryActualizarGUI*) en la que se inicia un objeto **QTimer** (temporizador) que, conectado a la función creada para la captura de vídeo nos permite actualizar los frames de entrada de video:

```
tmrTimer2 = new QTimer(this);
connect(tmrTimer2, SIGNAL(timeout()), this, SLOT(processFrameAndUpdateGUI2()));
tmrTimer2->start(20);
```

También es necesaria para poder mostrar dichas imágenes en un widget de tipo label la creación de un objeto tipo **QImage** en el que copiar los datos de la matriz (Mat) que contiene la captura de video. Después hace

falta asignar a este widget perteneciente a la ui (user interface) un mapa de píxeles **QPixmap** que señale a este objeto **QImag**

```
QImage qimgOriginal2((uchar*)matOriginal2.data, matOriginal2.cols,
matOriginal2.rows, matOriginal2.step,QImage::Format_RGB888);

ui->mainCapture->setPixmap(QPixmap::fromImage(qimgOriginal2));
```

Para poder lanzar las ventanas adicionales (todas las que no son la ventana principal) es necesario crear un objeto perteneciente a la clase que da nombre a la ventana (de tipo **QLabel** la que controla los eventos del ratón y **QDialog** el resto) y pedir su ejecución (**exec**()). Esto se ha asignado en todos los casos a la interacción con algún widget de la ventana principal o una acción de las disponibles en el menú desplegable o en la barra de herramientas, como en este caso, por ejemplo:

```
    connect(ui-
>actionSelecci_n_objetivo_manual,SIGNAL(triggered()),this,SLOT(openPuntero()));

    ……

void VentanaPrincipal::openPuntero()
{

    puntero punt;
    punt.exec();

}
```

En el caso de heredar valores de una de estas ventanas de diálogo, como es el caso por ejemplo del filtrado de **HSV**, se necesita crear variables a las que asignar los valores obtenidos en dicha ventana y aplicarlos a los setters (funciones para asignar valores a variables *private* de la clase de la ventana principal), de la siguiente forma:

```
void VentanaPrincipal::openDialog()
{

    dialog dlg;
    dlg.exec();

    int hueMax = dlg.GetHMax();
    int saturationMax = dlg.GetSMax();
    int valueMax = dlg.GetVMax();
    int hueMin = dlg.GetHMin();
    int saturationMin = dlg.GetSMin();
    int valueMin = dlg.GetVMin();

    SetHMax(hueMax);
    SetSMax(saturationMax);
    SetVMax(valueMax);
    SetHMin(hueMin);
```

```
    SetSMin(saturationMin);
    SetVMin(valueMin);

    QString estado = dlg.GetString();
    ui->infoColor->setText(estado)
    }
```

8.2.5. COMUNICACIÓN CON ARDUINO

Como comentábamos con anterioridad en el apartado de comunicación con **Arduino**, durante el proceso de adaptación del código creado en Visual Studio para el desarrollo de la aplicación en **Qt** hemos encontrado con la necesidad de implementar la utilización del módulo **QSerialPort** por problemas de compatibilidad con la clase **SerialClass.h** que habíamos utilizado previamente.

Este módulo hereda del **QIODevice** y nos ofrece una interfaz tanto para hardware como para puertos serial virtuales.

Para su uso en **Qt5** es necesario incluir la siguiente línea en el archivo de proyecto:

```
QT += serialport
```

Y, posteriormente, habrá que incluir los siguientes archivos de cabecera en donde sea necesaria su utilización:

```
#include <QtSerialPort/QSerialPort>
```

Para realizar la comunicación con **Arduino** procederemos de la siguiente forma:

- Se crea un objeto de la clase **QSerialPort** que llamaremos serial:

```
serial=new QSerialPort(this);
```

- Creamos un objeto de la clase **QTimer** que asociaremos a nuestra función de establecimiento de comunicación:

```
tmrTimer3 = new QTimer(this);

tmrTimer3->setInterval(4000);


connect(tmrTimer3,SIGNAL(timeout()),SLOT(close_serial()));
```

- Configuraremos la conexión fijando una serie de parámetros:

```
serial->setPortName("COM6");
```

```
if(serial->open(QIODevice::ReadWrite))
        {
           tmrTimer3->start();
          serial->setBaudRate(QSerialPort::Baud9600);
          serial->setDataBits(QSerialPort::Data8);
          serial->setParity(QSerialPort::NoParity);
          serial->setStopBits(QSerialPort::OneStop);
          serial->setFlowControl(QSerialPort::NoFlowControl);
```

- Creamos un array que actuará como buffer de salida en el que introduciremos los valores de las coordenadas x() e y(), que serán convertidas a cadena de caracteres **QString** (tiene una función "number" que nos permite desechar el uso de la función que habíamos creado en **Visual Studio** para conversión de valores de tipo númerico a cadenas de caracteres para su envío por puerto serial) y enviadas con la función **serial-> write()**. Incluiremos un carácter "**a**" al final del buffer que **Arduino** interpretará como fin de recepción de caracteres:

```
QByteArray output;

unsigned int var = GetCoordX();
unsigned int var2 = GetCoordY();
unsigned int var3 = GetAngulo();
QString str = QString::number(var);
QString str2 = "," + QString::number(var2);
QString str3 = "," + QString::number(var3);
QString fin = "a";

output.append(str);
output.append(str2);
output.append(str3);
output.append(fin);
serial->write(output);
```

- Esperamos a que se envíen todos los datos, esperamos a que el puerto esté disponible para lectura. Se llama a la función de lectura de datos.

```
serial->flush();
serial->waitForBytesWritten(1000);
serial->waitForReadyRead(1000);

connect(serial,SIGNAL(readyRead()),this,SLOT(read()));
```

- La función creada para lectura de datos incluye la creación de un array que actuará de buffer que recogerá los datos que se envíen a

través de **Arduino** y se utilizarán para rellenar el cuadro de texto de la ventana principal en el que se indica el estado del Robot.

```
void VentanaPrincipal::read()
{
  QByteArray data = serial->readAll();

  ui->plainTextEdit->insertPlainText(data);
 }
```

- Finalmente se limpiará el buffer de salida y el temporizador llamará a la función que cierra la comunicación cuando llegue al timeout.

```
output.clear();

void VentanaPrincipal::close_serial()
{
 serial->close();
 ui->infoConexion->setText("Desconectado");
}
```

Para comprobar el funcionamiento de la comunicación se puso en práctica la misma prueba que se realizó cuando se realizó el programa en Visual Studio y los resultados fueron satisfactorios.

8.2.6. APLICACIÓN DE EVENTOS CON PUNTERO

Se ha desarrollado una aplicación para la opción ofrecida que permite seleccionar con el puntero los objetivos a recoger por el brazo robótico. Esta aplicación hará uso de los módulos **QEvent** y **QMouseEvent** de **Qt**. Para ello se crean dos clases con sus correspondientes archivos **.cpp**: Una **Dialog** (puntero) y otra **Widget** (my_qlabel).

En la clase puntero se incluirá un marco "**label**" en el que se reproducirá la imagen obtenida por webcam y se crearán funciones para gestionar los eventos generados por interacción con el ratón.

Estas funciones son las siguientes:

```
void Mouse_current_pos();
void Mouse_Pressed();
void Mouse_left();
```

Mouse_current_pos() fijará el texto que contiene los valores de las coordenadas de **X** e **Y** en las que se encuentra el puntero.

Mouse_Pressed() cambiará el estado del puntero cuando el botón del ratón sea pulsado

Mouse_left() escribirá el texto correspondiente cuando el puntero se encuentre fuera del marco en el que se encuentra la imagen.

La clase **my_qlabel** será del tipo **QLabel** y así se especificára durante su creación:

```
class my_qlabel : public QLabel
```

En esta clase se utilizarán tres funciones pertenecientes al módulo **QMouseEvent**:

```
void mouseMoveEvent(QMouseEvent *ev);
void mousePressEvent(QMouseEvent *ev);
void leaveEvent(QEvent *);
```

Se crearán tres señales que serán emitidas desde cada función de las creadas:

```
void Mouse_Pressed();
void Mouse_Pos();
void Mouse_Left();
```

En el archivo **.cpp** se configura el funcionamiento que tendrán estos tres eventos creados.

MouseMoveEvent asignará los valores de las coordenadas en pixeles x e y en los que navega actualmente el puntero, los asignará a unas variables del mismo nombre y emitirá la señal **Mouse_Pos()**.

MousePressEvent emitirá la señal **Mouse_Pressed()** cuando el botón del ratón sea pulsado.

LeaveEvent emitirá la señal **Mouse_Left()** cuando el puntero esté fuera del marco de la imagen.

Finalmente, para establecer la relación entre las dos clases, en el archivo **puntero.cpp** se incluye en el **setup** de la **ui** las siguientes conexiones:

```
connect(ui>label,SIGNAL(Mouse_Pos()),this,SLOT(Mouse_current_pos()));

connect(ui>label,SIGNAL(Mouse_Pressed()),this,SLOT(Mouse_Pressed()));

connect(ui->label,SIGNAL(Mouse_Left()),this,SLOT(Mouse_left()));
```

9. PROTOCOLOS DE PRUEBAS

En este apartado se recogen los protocolos de pruebas implantados en los distintos módulos que componen el trabajo realizado con el fin de estudiar el grado en el que se han satisfecho las especificaciones de diseño planteadas inicialmente. Se añade un sub-apartado en el que se recogen los resultados en tablas para su consulta rápida.

9.1. MÓDULO 1 : DETECCIÓN

Para este módulo se han realizado una serie de pruebas con el fin de comprobar si se realiza una detección adecuada de los objetos que tendremos que manipular, atendiendo a los requisitos que se han señalado en el correspondiente apartado de esta memoria.

- Se han realizado varias operaciones de **calibrado de cámara**, utilizando el programa que ofrece **OpenCV** y modificando los parámetros para que se ajusten a nuestras necesidades, con el fin de obtener las matrices de cámara y de coeficientes de distorsión, que posteriormente aplicaremos en nuestro programa. Las pruebas se han realizado según recomendaciones indicadas en la documentación online de **OpenCV**, utilizando un tablero de ajedrez para obtener detecciones de las esquinas de sus casillas. Los resultados han sido moderadamente satisfactorios, ya que en ocasiones, tras realizar la operación de calibrado y mostrar la imagen con las correcciones correspondientes ya aplicadas, se aprecian distorsiones similares al efecto fotográfico conocido como ojo de pez.

Proceso de calibración: Obtención de frames

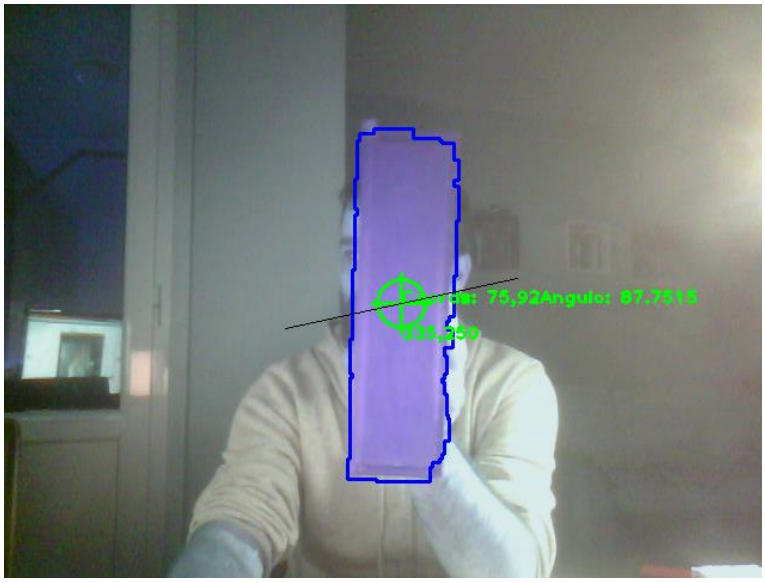
Ejemplo de mala calibración

Ejemplo de buena calibración

- Se ha comprobado que la aplicación desarrollada tiene una sintaxis correcta y un **funcionamiento sin interrupciones** y que no se producen salidas inesperadas del programa.
- Se han realizado pruebas en cada una de las **operaciones del procesamiento de imágenes** para comprobar su correcta implementación y, tal como se ha mostrado en imágenes aportadas en los apartados correspondientes en la presente memoria, se han obtenido buenos resultados en todas ellas.
- Se han realizado una serie de intentos de **detección correcta de objetos** tras la segmentación y los resultados han sido

moderadamente convincentes, debido a que en condiciones de escasa iluminación se han encontrado dificultades.

- En lo referente a la **orientación del objeto** en el espacio se han encontrado bastantes problemas y, en ocasiones, no se han obtenido resultados correctos. Para comprobar estos resultados se ha utilizado la misma plantilla que se empleó durante el proceso de calibración de servos y los ángulos obtenidos en el programa, en contraste con los medidos sobre la plantilla, han tenido ciertas desviaciones, del orden del 5 al 10%.
- Se han realizado pruebas para comprobar la buena **detección y representación de contornos**. Esta operación es relevante, porque sobre la detección de contornos se realiza el cálculo de momentos que nos permite hallar la posición del centro de masas/centroide del objeto detectado. Según las condiciones de iluminación de nuevo, el proceso de filtrado de la imagen, durante el procesamiento de esta, dificulta posteriormente una buena detección de los contornos. A pesar de no ser en principio necesario, se ha decidido incluir la representación gráfica de los contornos en la interfaz gráfica del programa, ya que, en caso de apreciarse una deficiente representación, el usuario puede entender necesaria la realización de un nuevo proceso de filtrado para mejorar los resultados.



Detección de contornos deficiente

- Por último se han realizado pruebas para comprobar el grado de exactitud de la **transformación homográfica** aplicada a las coordenadas de la imagen para convertirlas a coordenadas del mundo real.

Como se ha expuesto en el apartado **6.4.8** de esta memoria, se ha realizado durante el desarrollo del programa una comprobación consistente en aplicar la transformación sirviéndonos de la matriz de homografía obtenida sobre las coordenadas en píxeles del origen de coordenadas seleccionado para obtener las coordenadas reales en milímetros. El resultado debería de ser lo más cercano a **(0,0)** posible, y en nuestro caso hemos obtenido el siguiente resultado:

[2.455201, 1.732573]

No obstante, se han realizado tests adicionales: en el programa se ha dispuesto que las coordenadas convertidas sean mostradas en pantalla al realizarse una detección junto con las coordenadas en píxeles y se ha procedido a realizar comprobaciones sobre una plantilla para constatar su correspondencia respeto al origen de coordenadas que hemos seleccionado. Los resultados han sido de nuevo variados: En ocasiones hemos encontrado discrepancias de hasta 5 milímetros, pero teniendo en cuenta que se trabaja con una cámara web de una calidad no profesional, en la que, a pesar de haber realizado correcciones mediante el procedimiento de calibrado de cámara, seguirá existiendo un considerable efecto de distorsión, se consideran estos resultados aceptables.

Es importante reseñar que el programa diseñado para el cálculo de la matriz homográfica es bastante inestable y se producen en ocasiones interrupciones indeseadas y crashes.

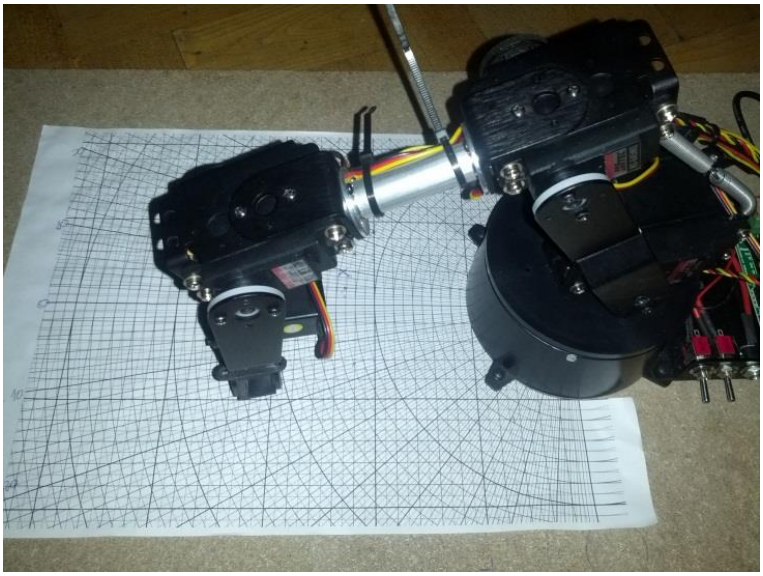
9.2. MÓDULO 2 : BRAZO ROBÓTICO

- Se comprueba la concordancia de los resultados obtenidos con los cálculos obtenidos por **Matlab** para la cinemática inversa y los que se obtienen en el desarrollo geométrico.
- Se hacen tests para confirmar el movimiento correcto de los servos tras la aplicación de la librería que modera la velocidad de estos, con el fin de evitar movimientos bruscos. Se han introducido una serie de órdenes de desplazamiento para el brazo y se ha comprobado que se realizaban de forma suave y fluida.
- Se comprueba la correcta calibración de servos introduciendo órdenes de desplazamiento para cada uno en milisegundos y comprobando que se mantienen dentro de los rangos expuestos y devuelven los valores angulares que se esperan.
- La manipulación de los bolos resulta en ocasiones problemática, debido al perfil curvo de su superficie, que produce en ocasiones

resbalamiento sobre la pinza del actuador final del brazo robótico. Sería recomendable implementar algún tipo de mejora para solucionar este problema encontrado.

- Se comprueba la precisión en el posicionamiento del extremo final del brazo situando una plantilla sobre la superficie del tablero en la que se encuentra fijado y se envían comandos para situarlos en posiciones determinadas del plano **(x,y,z)**. Se ha observado que, debido a rozamientos producidos por un defectuoso estado de los rodamientos de la base, se producen en ocasiones errores considerables de posicionamiento en el eje **x**. Se reproducen en la siguiente tabla algunas de las posiciones obtenidas contrastadas con las deseadas:

• DESEADAS (x,y,z)	OBTENIDAS (x,y,z)
200, 150,100	**193, 150,100**
-130,175,50	**-133,177,50**
0,200,0	**3,205,0**
100,180,40	**100,185,45**
-200,200,0	**-185,201,0**

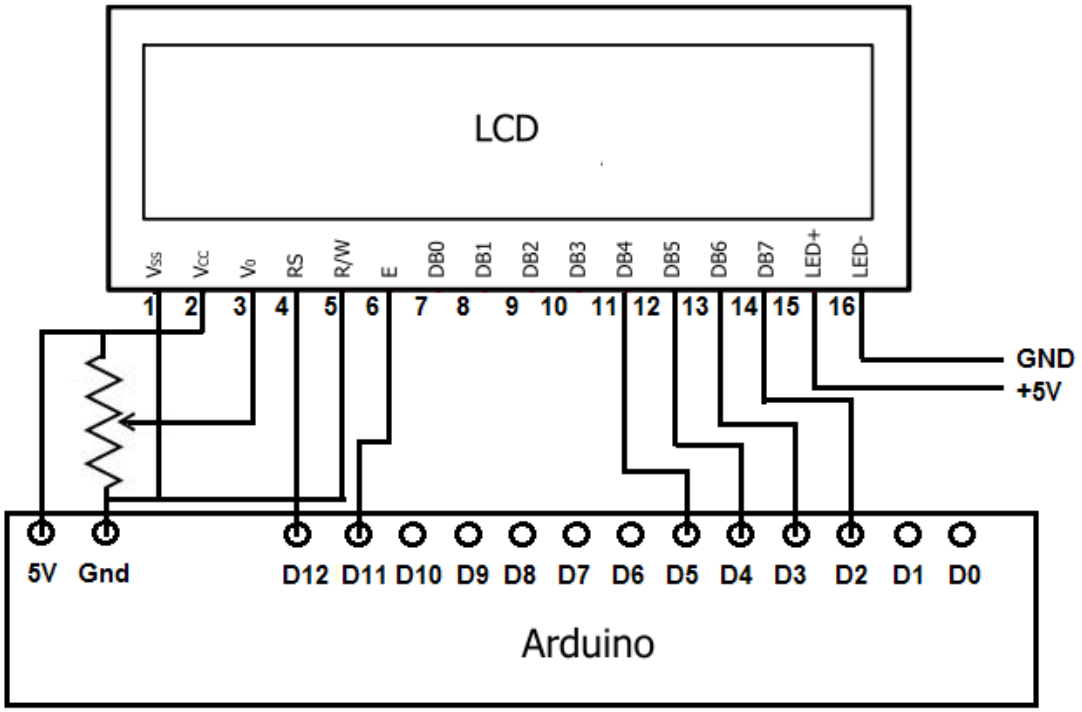


Plantilla utilizada para comprobaciones y medidas

- Un apartado fundamental para el correcto funcionamiento del sistema es el referente a la comunicación entre el programa desarrollado para la detección de objetos y el control del brazo robótico. Para

comprobar el funcionamiento de dicha comunicación se ha utilizado un display **LCD** en el que mostrar los caracteres recibidos por el puerto serial.

Utilizamos un **LCD Hitachi HD44780**, que conectaremos a nuestra placa Arduino siguiendo el esquema de la siguiente página:

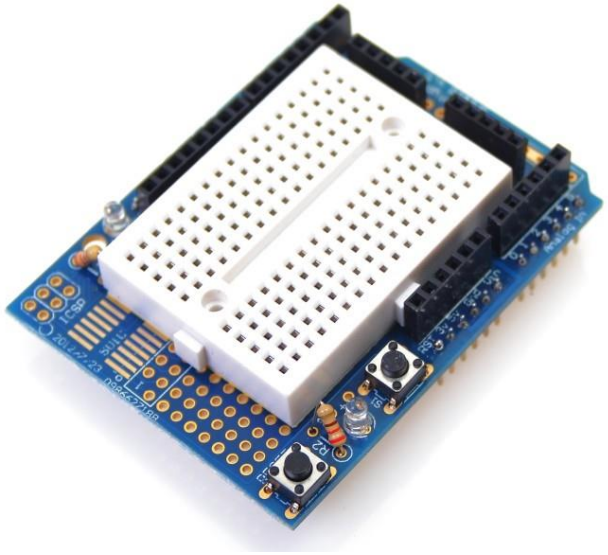


Esquemático de conexiones de LCD a tarjeta Arduino

Por tanto los pines que se conectarán serán:

1. A la pata conectada a tierra de un potenciómetro de 10K.
2. A la pata conectada a 5V del mismo potenciómetro.
3. Será el ajuste de contraste y lo conectaremos a la pata central del potenciómetro.
4. Selector de registro (0 orden, 1 datos), conectado al pin 12.
5. Read/Write (0 Escribir, 1 leer). Nos interesa sólo escribir en él, por tanto lo tendremos conectado a tierra, en la pata del potenciómetro que comentábamos anteriormente.
6. Activar clock que iniciará el ciclo de escritura o lectura.
7. Los pines 11,12,13 y 14 son DB (buses de datos).
8. Pines 15 y 16 los utilizaremos para activar la retroiluminación. El pin 15 se conectará a la alimentación, mientras que el 16 lo hará a tierra.

Para realizar el montaje hemos utilizado una proto-shield **Keyes 5.1** instalada sobre una tarjeta **Arduino uno**.



Protoshield utilizada para el montaje

Posteriormente, se ha escrito un código en **Visual Studio** en el que creamos un objeto de la clase **SerialClass**, establecemos la comunicación con nuestra tarjeta **Arduino** a través del puerto **COM5** y le enviamos a ésta el contenido del buffer de salida, cuyo contenido es la siguiente cadena de caracteres:

char palabra[ ] = "150,100,90";

En **Arduino** crearemos a su vez un código en el que nos aseguraremos primero de incluir la librería **LiquidCrystal.h**, para poder utilizar nuestro display **LCD**. Indicaremos los pines I/O de nuestra tarjeta que están conectados al display (11,12,5,4,3,2) en nuestro caso.

Creamos un buffer de entrada, al igual que en **Visual Studio** y posteriormente, en **setup()**, inicializamos el LCD con la orden **lcd.begin**(número de columnas, número de filas) y empezaremos imprimiendo algo en la pantalla ("Recibiendo datos").

Se pondrá el cursor en la primera columna de la segunda fila y se irán obteniendo los valores recibidos del buffer de salida del programa de Visual Studio extrayendo sus valores númericos y obviando las comas separadoras con la función **parseInt()** y asignamos estos valores a unas variables "**x**:", "**y**:" y "**deg**" para mostrarlas en la pantalla.

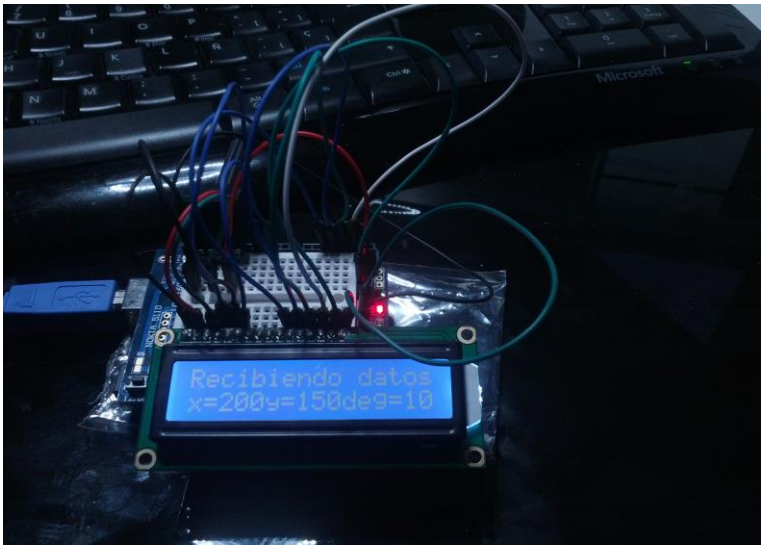
Al terminar de recibir datos y estar los campos cubiertos inicializaremos el buffer de entrada de nuestro programa de Arduino:

```
#include <LiquidCrystal.h>
```

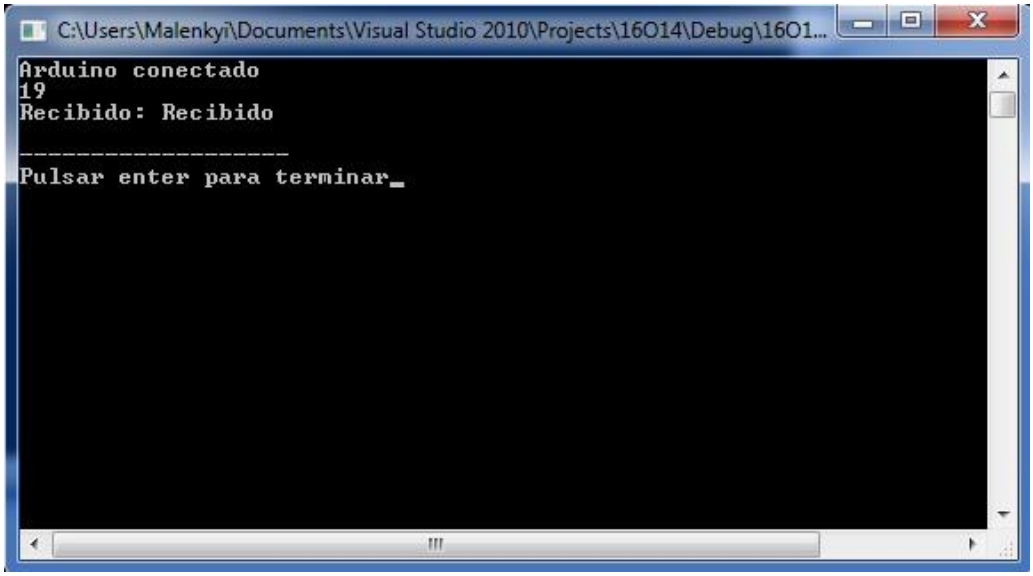
```
const int NumRows = 2;

const int NumCols = 16;



LiquidCrystal lcd(12,11,5,4,3,2);



const int NUMBER_OF_FIELDS = 3;

int fieldIndex = 0;

int values [NUMBER_OF_FIELDS];



void setup()

{

Serial.begin(9600);

lcd.begin(NumCols,NumRows);

lcd.print("Recibiendo datos");

}

void loop()

{

lcd.setCursor(0, 1);

if(Serial.available())

{

for(fieldIndex = 0; fieldIndex < 10; fieldIndex ++)

{

values[fieldIndex] = Serial.parseInt();

}

print("x=");

lcd.print(values[0]);

lcd.print("y=");
```

```
lcd.print(values[1]);

lcd.print("deg=");

lcd.print(values[2]);

fieldIndex = 0;

}

}
```

El resultado se muestra en las siguientes imágenes:



Datos obtenidos mostrados en la pantalla LCD



Confirmación de recepción de datos en consola de comandos

con lo que podemos concluir que la comunicación entre nuestro código en C++ y nuestra tarjeta **Arduino** se está llevando a cabo correctamente.

Es necesario indicar ahora que al realizar el porteado del código que hemos desarrollado en **Visual Studio**, para adaptarlo al contenido de la interfaz gráfica construida con el programa **Qt Creator** y **Designer**, hemos tenido que utilizar otra clase para asegurar la comunicación con nuestra tarjeta. Esta clase a la que nos referimos es una de las incluidas en la librería propia de **Qt** y se llama **QserialPort**. Se describirán los detalles más en profundidad en los capítulos correspondientes al desarrollo de la **GUI** en **Qt** para nuestra aplicación.

9.3. MÓDULO 3 : INTERFAZ DE USUARIO

Las pruebas realizadas sobre este módulo han sido básicamente consistentes en las necesarias para asegurar su funcionalidad, su correcta adaptación del código inicialmente desarrollado en **Visual Studio** y su adecuado diseño y accesibilidad.

- Se ha comprobado que la navegación entre los distintos documentos que integran la aplicación desarrollada se realiza de manera satisfactoria y que dicha navegación resulta intuitiva, facilitando al usuario acceder de forma inequívoca a las secciones que desee.
- Se ha comprobado que todos los widgets presentes y disponibles en la aplicación realizan las funciones para las que han sido diseñados de manera adecuada.
- Se han realizado pruebas sobre las funciones adicionales incorporadas para realizar una recogida seleccionando objetivos mediante el uso

del puntero y la posibilidad de controlar las articulaciones del brazo robótico mediante el accionamiento de unos diales.

En ambos casos los resultados han sido moderadamente satisfactorios, ya que en el caso de la selección de objetos con puntero se han encontrado problemas para realizar una correcta transformación de coordenadas, a pesar de utilizar el mismo procedimiento que el utilizado en el programa de detección con buenos resultados.

En la aplicación para el control de las articulaciones del brazo robótico de forma manual se han encontrado bugs que hacen desplazarse en ocasiones a más de una articulación al accionar un dial.

- Se comprueba la correcta comunicación con Arduino. Esto ha sido necesario, porque se han tenido que realizar ciertas modificaciones para adaptar el código de Visual Studio a Qt. El procedimiento para realizar estas pruebas de comunicación ha sido análogo al empleado en el caso expuesto en el apartado **9.2.** y los resultados han sido moderadamente satisfactorios: Mientras que el protocolo para comunicar la tarjeta con la aplicación con el programa sencillo de prueba se ha realizado sin problemas, al implantarlo en el proyecto definitivo han existido bastantes errores, produciéndose pérdidas de datos o problemas como el comentado en la aplicación para control manual de las articulaciones del robot, que requerirían de un depurado del código.

9.4. RECOPILACIÓN DE RESULTADOS

Como se ha comentado en apartados anteriores, se han realizado diferentes pruebas en cada uno de los módulos que componen este proyecto para comprobar su funcionalidad. En esta sección se expondrán recopilados esquemáticamente los resultados de dichas pruebas, el grado de satisfacción obtenido en estas (Alto, bueno, moderado y aceptable) y su repetibilidad, donde sea aplicable (Si, No, Bueno o no aplicable N/A).

En el programa de detección de objetos y extracción de características se han llevado a cabo las comprobaciones enunciadas en la siguiente tabla:

PRUEBA REALIZADA	GRADO DE SATISFACCIÓN	REPETIBILIDAD
Calibrado realizado satisfactoriamente	Bueno	Si
Funcionamiento de la aplicación sin interrupciones	Alto	Si
Operaciones de procesamiento de imagen correctas	Alto	Si
Detección correcta de objetos	Alto	Si
Obtención y representación de contornos	Bueno	Bueno
Obtención de características del objeto detectado	Alto	Si
Orientación del objeto en el espacio	Deficiente	No
Correcta detección en cualquier condición de iluminación	Moderado	No
Cálculo correcto de proyección homográfica	Bueno	Aceptable

Para el brazo robótico se han puesto en práctica las siguientes pruebas:

PRUEBA REALIZADA	GRADO DE SATISFACCIÓN	REPETIBILIDAD
Cálculo de cinemática y modelo Matlab	Alto	N/A
No se alcanzan posiciones que produzcan colisiones	Alto	Si

Respuesta correcta a instrucciones de programa desarrollado	Alto	Si
No se producen movimientos bruscos del brazo	Alto	Si
Funcionamiento correcto de servos	Moderado	Moderado
Precisión al alcanzar posiciones con manipulador	Adecuado	Moderado
Manipulación de objetos a trasladar	Aceptable	Aceptable
Comunicación de placa con Visual Studio	Moderado	Moderado

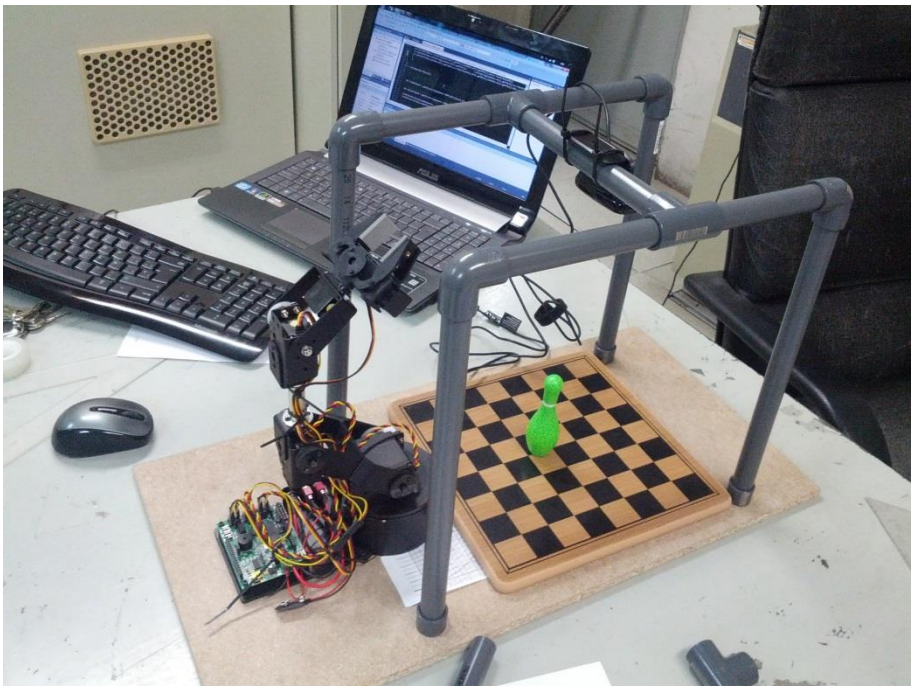
En la interfaz gráfica de usuario:

PRUEBA REALIZADA	GRADO DE SATISFACCIÓN	REPETIBILIDAD
Conexión entre distintos documentos	Alto	Si
Integración de código de detección	Alto	Si
Funcionamiento correcto de widgets	Alto	Si
Funcionamiento correcto de eventos de puntero	Moderado	Moderado
Comunicación con Arduino	Moderado	Moderado

9.5. MONTAJE DE PROTOTIPO PARA PRUEBAS

Para la realización de pruebas del funcionamiento del sistema ensamblado se ha procedido al montaje de un prototipo. Para ello se ha instalado el brazo robótico sobre una base que a su vez servirá de zona de trabajo y la cámara web se ha dispuesto sobre unos soportes de pvc, que la mantienen sobre dicha zona.

En la fecha en la que se finaliza la redacción de esta memoria, todavía no se dispone de un prototipo final y se ha trabajado sobre uno provisional. Se espera realizar la presentación y defensa de este trabajo con la versión definitiva de este.



Montaje de prototipo provisional

En la documentación aportada junto a este trabajo se aporta un video en el que se muestran pruebas de funcionamiento del sistema ensamblado realizadas con este prototipo.

Resultados obtenidos tras la realización de pruebas con este prototipo nos han llevado a la conclusión de que resulta imposible ofrecer una solución para la recogida y depósito de los bolos que se encuentren en posiciones tal que su cabeza se encuentre apuntando en la dirección del brazo al no disponer de un giro de muñeca adecuado. Con los 180º

disponibles tan sólo se podría solucionar esto realizando una operación adicional, y no se considera una solución óptima en ningún caso.

Las limitaciones impuestas por las dimensiones del brazo reducen también el campo de actuación sobre el que podemos realizar ensayos, lo cual ha conducido a prácticamente centrar estos ensayos en la localización, manipulación y colocación de un bolo, a pesar de contar con un programa de detección plenamente funcional.

A pesar de la relativa calidad del brazo robótico del que se ha dispuesto, el problema del rozamiento producido en la base resulta una influencia considerable en los malos resultados de posicionamiento que se producen con cierta frecuencia.

10. PRESUPUESTO

CONCEPTOS	UDS	FABRICANTE	IMPORTE (€)
Brazo robótico Lynxmotion AL5A	1	Lynxmotion	251,04
Kit rotación muñeca Lynxmotion	1	Lynxmotion	35,56
Webcam Logitech c525	1	Logitech	41,79
Material para montaje de prototipo		Varios	20
Arduino UNO	1	Arduino	20
Display LCD 16x2	1	Toshiba	9,90
Shield prototipo con minibreadboard Arduino	1	Keyes	5,30
Componentes electrónicos (cables,etc...)		Varios	10
Licencias de software		Varios	Gratuitas
OpenCV 2 Programming cookbook	1	Packt publishing	35,66
Arduino Cookbook	1	O'Reilly	35,57
Fundamentos de robótica y mecatrónica con Matlab	1	Ra-Ma	34,90
Mano de obra-ingeniería	250	N/A	2500
TOTAL			2999,72

CÓDIGO DE COLORES	
	ROBOT
	ELECTRÓNICA
	DOCUMENTACIÓN
	OTROS

11. CONCLUSIONES

En este apartado recogeremos el alcance y limitaciones del trabajo desarrollado, tras la observación de los resultados y el grado en el que se han cumplido los requisitos de diseño, y para finalizar se hará un balance y una valoración final en forma de impresiones personales.

11.1. ALCANCE Y LIMITACIONES

Indicaremos a continuación cual es el alcance y limitaciones de este trabajo. En ellas se han tenido en cuenta las posibilidades reales de desarrollo de un sistema efectivo y que se encontrará al término de éste proyecto aún en estado de prototipo inicial, que podría servir para analizar resultados y sobre el que incluir una serie de mejoras que también serán indicadas.

11.1.1. ALCANCE

- Se ha conseguido desarrollar un sistema con cierto grado de autonomía.

- El sistema puede adaptarse a las características de juego de bolos asturianos debido a su capacidad para distinguir bolos de distintos tipos y tamaños.

- Se ha conseguido integrar satisfactoriamente el funcionamiento del elemento electromecánico (robot) con el sistema de control informático.

- Se ha conseguido un grado de comunicación aceptable entre todos los elementos del sistema.

- El programa para la detección de los bolos realiza correctamente su función y se consigue una localización de estos en el espacio dentro de los límites aceptables.

- El usuario dispone de una interfaz gráfica desde la que se pueden realizar todas las operaciones disponibles.

- Se ha montado un prototipo en el que se han realizado pruebas de funcionamiento del conjunto.

11.1.2. LIMITACIONES

- Por las características del prototipo que se desarrollará, el resultado obtenido no significará una solución al problema planteado, sino un acercamiento a este y una vía abierta al posterior desarrollo sobre lo expuesto.

- No ha sido posible realizar una prueba convincente para mostrar el funcionamiento de la recogida de bolos en cualquier posición en la que se encuentren debido a la falta de un sistema para poder desplazar el robot sobre el terreno de juego.

- No se ha conseguido poder realizar pruebas con todos los bolos debido a las limitaciones dimensionales del robot, que reducen el campo de actuación de este y por tanto del terreno disponible para la realización de pruebas adecuadas.

- El material empleado ha supuesto un alto condicionamiento para la obtención de resultados satisfactorios y del grado de precisión logrado.

- Se han encontrado fallos en algunas funciones del software que requerirían de una depuración del código empleado.

- Al haberse realizado el montaje del prototipo contemplando su posible desmontaje para realización de pruebas y traslados, es frecuente la necesidad de realizar calibrados continuos.

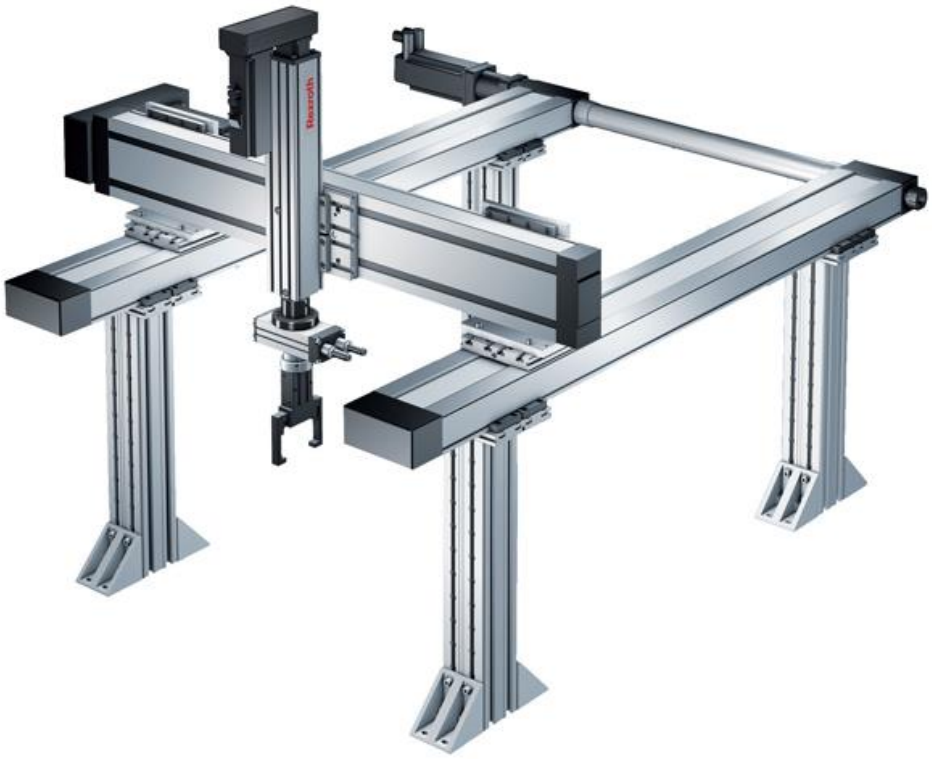
11.2. PROPUESTAS DE LÍNEAS FUTURAS

Teniendo en cuenta los resultados obtenidos tras el desarrollo de este trabajo, se han considerado una serie de propuestas que se podrían seguir en líneas futuras para conseguir un grado de desarrollo del producto que se adapte mejor a los requisitos de diseño.

Considerando los problemas que han surgido debido a las dimensiones del brazo robótico, quizá sería recomendable volver sobre la idea inicial propuesta por el profesor Jose Manuel Sierra Velasco de utilizar el brazo robótico **Scorbot ER-V Plus** de que dispone la Universidad. La implementación del código desarrollado para el **Lynxmotion AL5A** sería perfectamente compatible y tan sólo habría que tener en cuenta la utilización de lenguaje ACL de dicho brazo para el diseño de un emulador de

terminal con el que establecer una comunicación serial para poder operar con el robot.

Tras resolver que la forma más adecuada de realizar el acercamiento del robot al bolo sería desde arriba, otra posibilidad sería la de desarrollar un sistema de traslado del brazo robótico sobre el área de juego. Un sistema similar al de una mesa XY sería lo más adecuado para asegurar una buena precisión en el posicionamiento del robot sobre los bolos derribados y facilitar el acceso a estos sin colisiones con otros elementos que se encuentren sobre el terreno.



Concepto propuesto

12. BIBLIOGRAFÍA

- **[1] Learning OpenCV: Computer vision in C++ with the OpenCV library**- Gary Bradski- *2008,O'Reilly.*
- **[2] OpenCV 2 Computer Vision Application Programming Cookbook**- Robert Laganiere-*2011, Packt Publishing*.
- **[3] Homography estimation**- Elan Dubrofsky- *2009 University of British Columbia.*
- **[4] OpenCV online documentation database**: http://docs.opencv.org/index.html
- **[5] Getting started with Arduino**- Massimo Banzi- *2011, O'Reilly.*
- **[6] Making things talk**- Tom Igoe- *2009, O'Reilly.*
- **[7] Arduino Cookbook**- Mark Margolis- *2012, O'Reilly.*
- **[8] Arduino, referencia, y documentación en línea**: http://www.arduino.cc/es/
- **[9] Foundations of Qt Development** – Johan Thelin – *Apress 2007.*
- **[10] Qt-project documentation** : http://doc.qt.io/
- **[11] Fundamentos de robótica y mecatrónica con Matlab y Simulink** – Marco A. Pérez Cisneros- *2014, Ra-Ma.*
- **[12] Instrumentación electrónica** – Miguel A. Pérez García, Juan C. Álvarez Antón, Juan C. Campo Rodriguez, Fco. Javier Ferrero Martín, Gustavo J. Grillo Ortega- *2008, Paraninfo*.

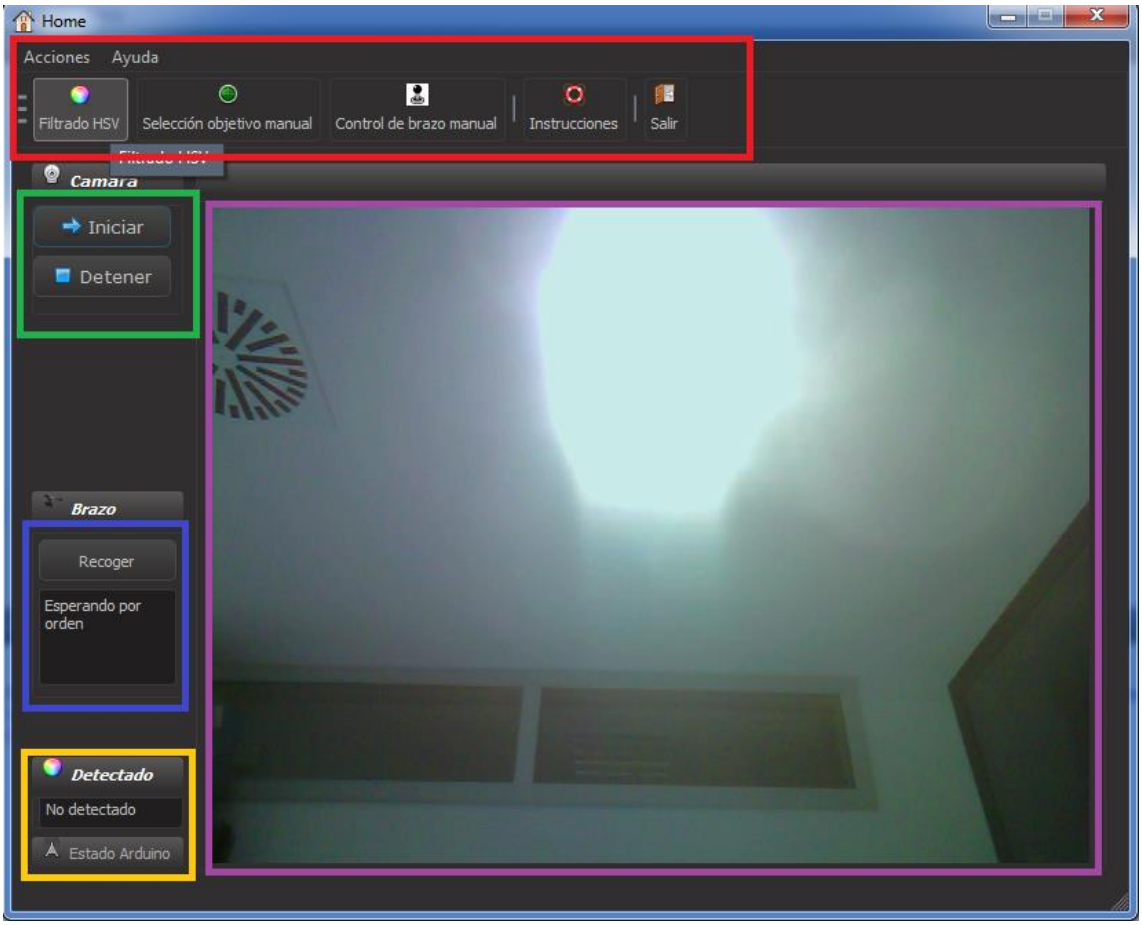
13. ANEXO I : MANUAL DE USUARIO DE INTERFAZ GRÁFICA

Para comenzar a utilizar el programa haga clic sobre el icono de la aplicación Recogida de Bolos.

Una vez se encuentre en la página de entrada/Bienvenida, haga clic en el botón inferior para acceder a la ventana principal (también llamada Home ) de la aplicación:

VENTANA PRINCIPAL

En la ventana principal dispone de una serie de opciones que se indican en el gráfico:



Ventana principal - Home

Marcadas en el recuadro rojo se encuentran las opciones/ acciones principales. Pueden utilizarse en el menú de barra de herramientas o bien pueden desplegarse en el menú de acciones. Será posible arrastrar la barra de herramientas y disponerla en la colocación que se desee.

Las acciones serán las siguientes:

-  **Filtrado de HSV**: Nos llevará a la ventana en la que podremos introducir los valores HSV para realizar el filtrado de la imagen y comprobar que se realizan detecciones correctas de los objetos que deseamos recoger.
-  **Selección de objetivo manual**: Desplegará la ventana correspondiente a esta opción, donde se nos permitirá seleccionar con el puntero el objeto a recoger por el brazo.
-  **Control de brazo manual**: Desplegará el widget correspondiente, en el que se dispone de 6 diales con los que se controlarán las articulaciones del brazo.
-  **Instrucciones**: Se ofrecerá aquí acceso al manual de instrucciones para realizar consultas.
-  **Salir** del programa y volver a Windows.

En el recuadro verde se localizan los controles de la webcam.  :

-  **Iniciar**: Da paso a la captura de imágenes con la webcam.
-  **Detener**: Se detiene la captura.

Enmarcados en el recuadro verde se encuentran los controles para enviar datos al brazo robótico.  :

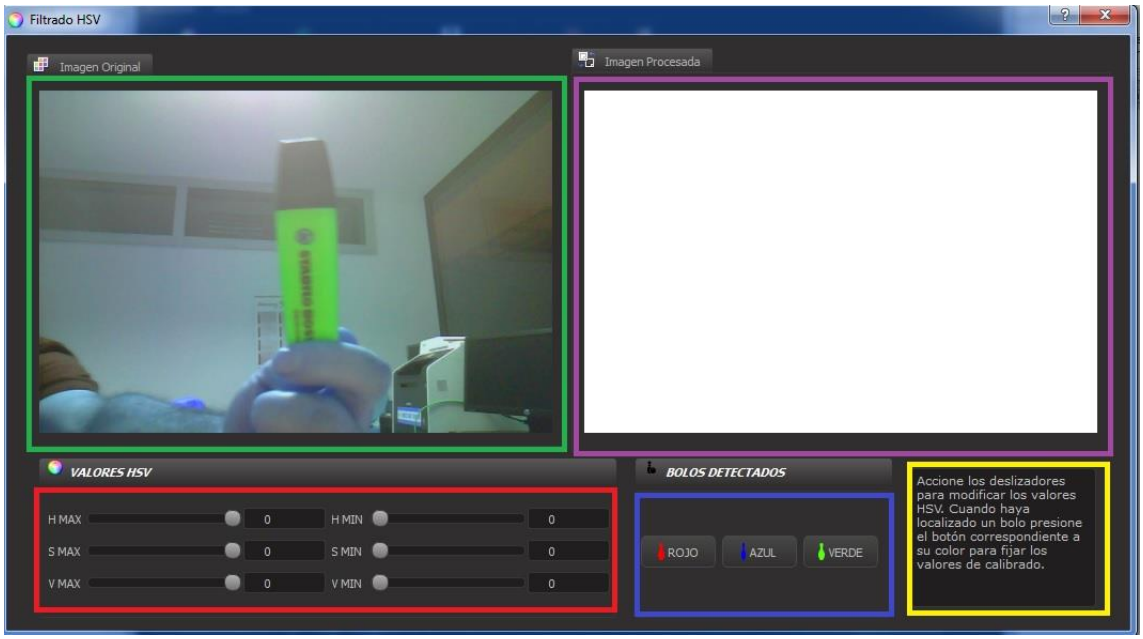
- **Recoger**: Al pulsar en este botón se enviarán las coordenadas del objeto detectado y localizado al brazo robótico para proceder a su recogida. Bajo este botón se encontrará información sobre el proceso y cuando se puede iniciar otra operación.

En la caja amarilla figuran dos pestañas a través de las cuales podemos recibir información sobre el estado del proceso:

-  **Estado Arduino**: Sus valores serán: Conectado si actualmente existe una conexión entre la tarjeta y la aplicación y Desconectado si no hay conexión con dicho dispositivo.
-  **Color detectado**: Informa del color del bolo detectado: Puede ser ROJO, VERDE o AZUL.

FILTRADO HSV

En la imagen se muestra la ventana , en la que se realiza la detección de los bolos para su recogida:



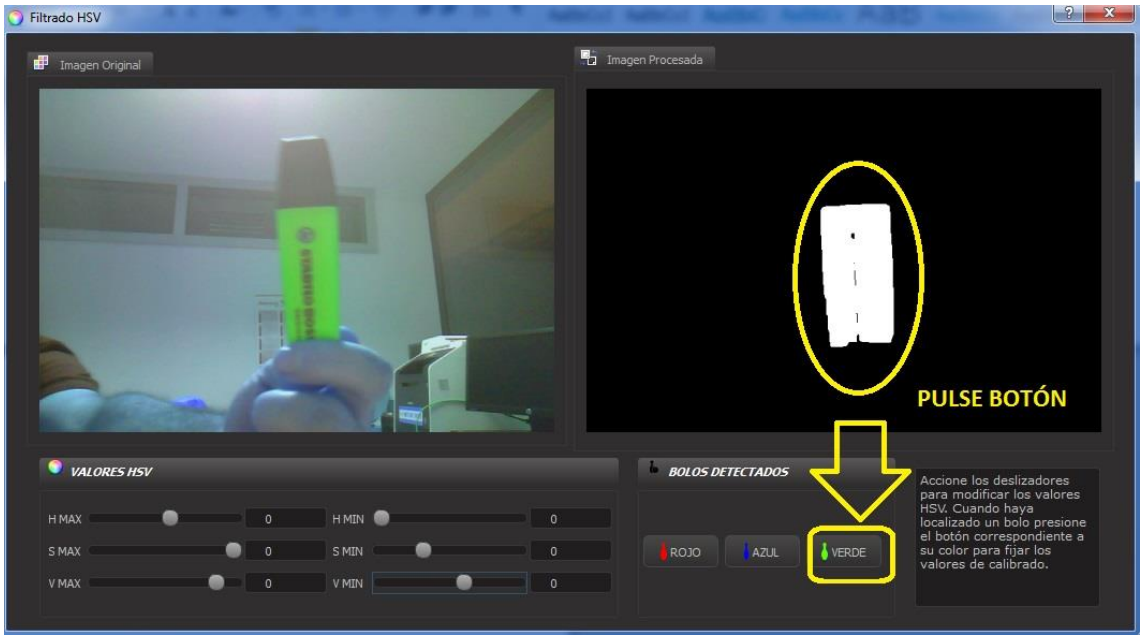
Ventana de detección – filtrado HSV

Los elementos presentes en esta ventana han sido marcados una vez más de la siguiente forma:

- **Imagen Original**: (Caja verde). Aquí se mostrará la imagen tal como ha sido capturada con la webcam, sin procesamiento.
- **Imagen Procesada**: (Caja lila). Se mostrará la imagen resultante, tras el procesamiento de imagen.
- **Filtrado HSV**: (Caja roja). Se disponen unos deslizadores con los que introducir valores de HSV min/max con los que realizar el filtrado de la imagen. A la derecha de cada deslizador se dispone de un display en el que se muestran los valores seleccionados.
- **Bolos detectados**: (Caja azul). Cuando se haya realizado una detección correcta de un bolo. Se dispone de estos botones para hacer clic en el color correspondiente: Si, por ejemplo, el bolo detectado es rojo, presione el botón ROJO.

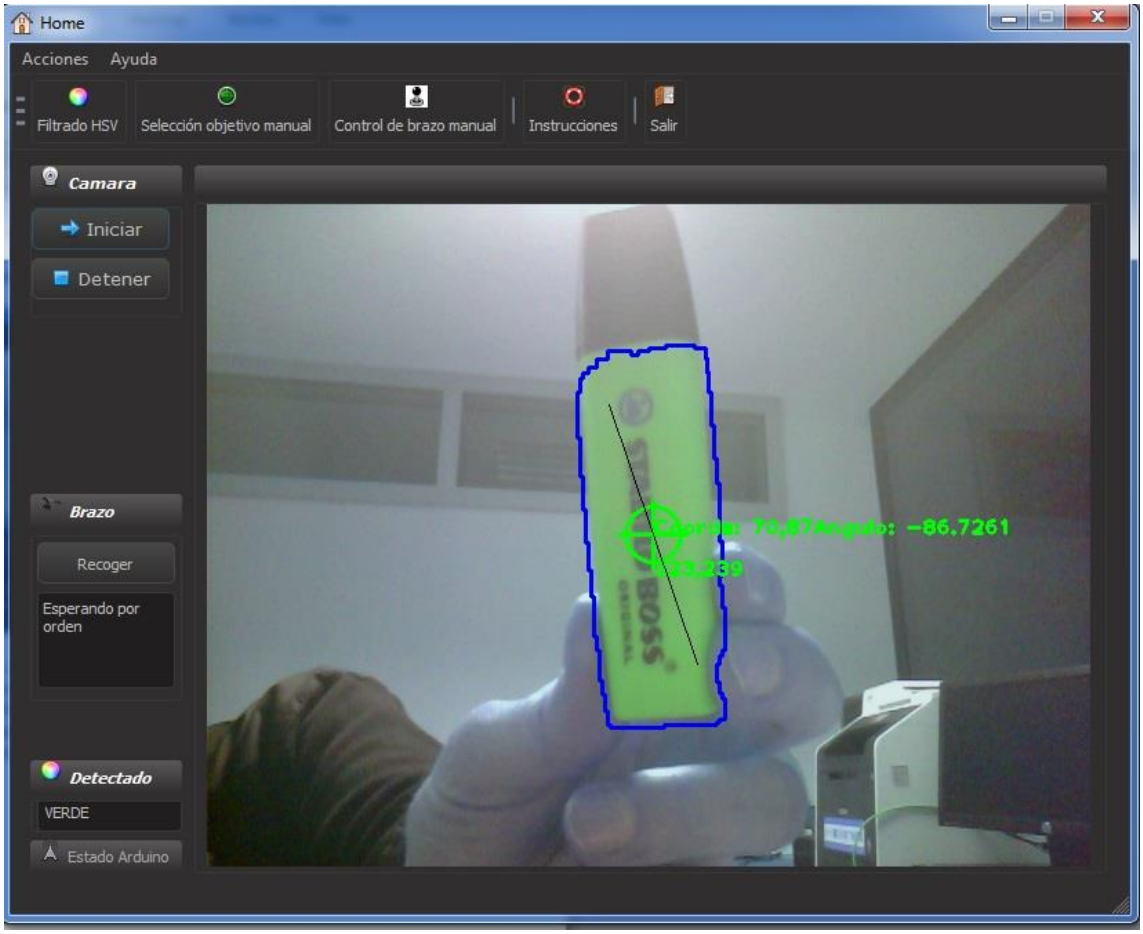
- La caja amarilla es un recuadro de información, en el que se indica el procedimiento a seguir para realizar la detección.

Tras la introducción de valores de HSV se conseguirá, y cuando se haya conseguido una detección correcta se obtendrá un resultado similar al mostrado en la imagen inferior:



Tras la introducción de valores para filtrado, validamos la detección

Aquí podemos ver cómo, tras introducir unos valores con los deslizadores, hemos conseguido una detección clara del objeto de la captura. El siguiente paso sería pulsar el botón del color correspondiente, tras lo cual la ventana se cerrará y se desplegará de nuevo la ventana principal, con el objeto ya detectado y en seguimiento. En la pestaña informativa **DETECTADO** se muestra como el objeto en seguimiento es del valor **VERDE** y ahora, en caso de desear ordenar su recogida, sólo restaría pulsar el botón "**Recoger**" en el cuadro **BRAZO**.

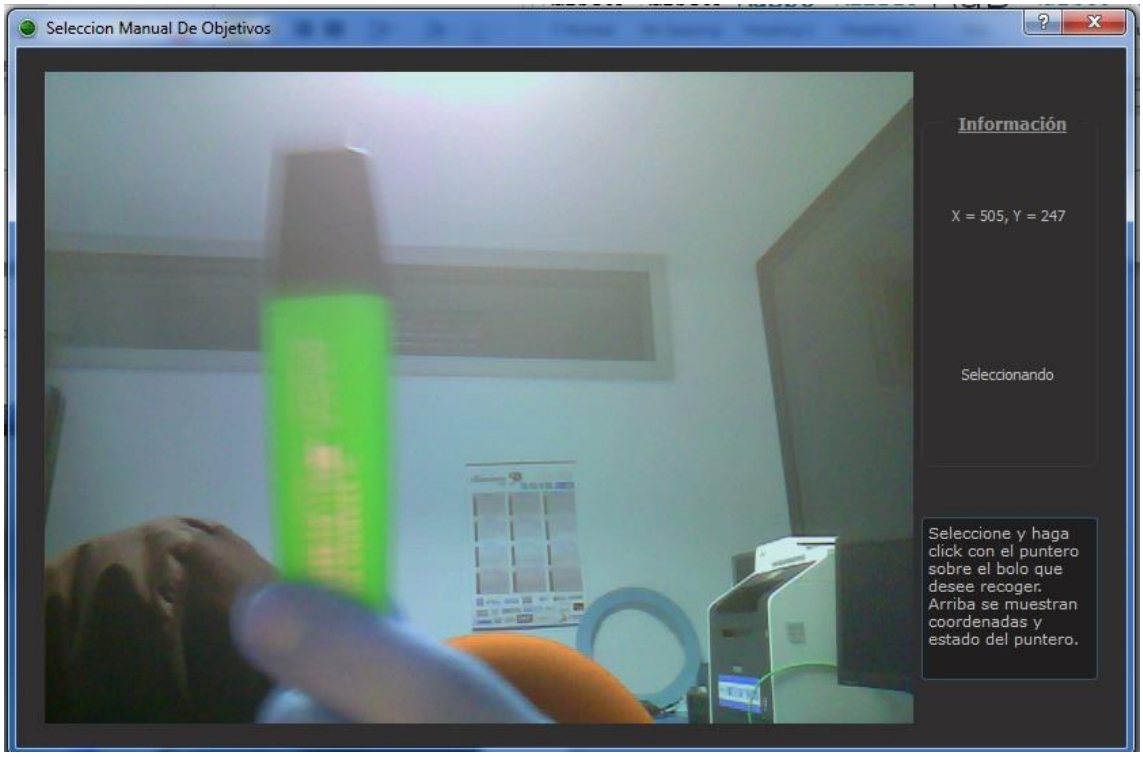


Seguimiento de objeto detectado en ventana principal

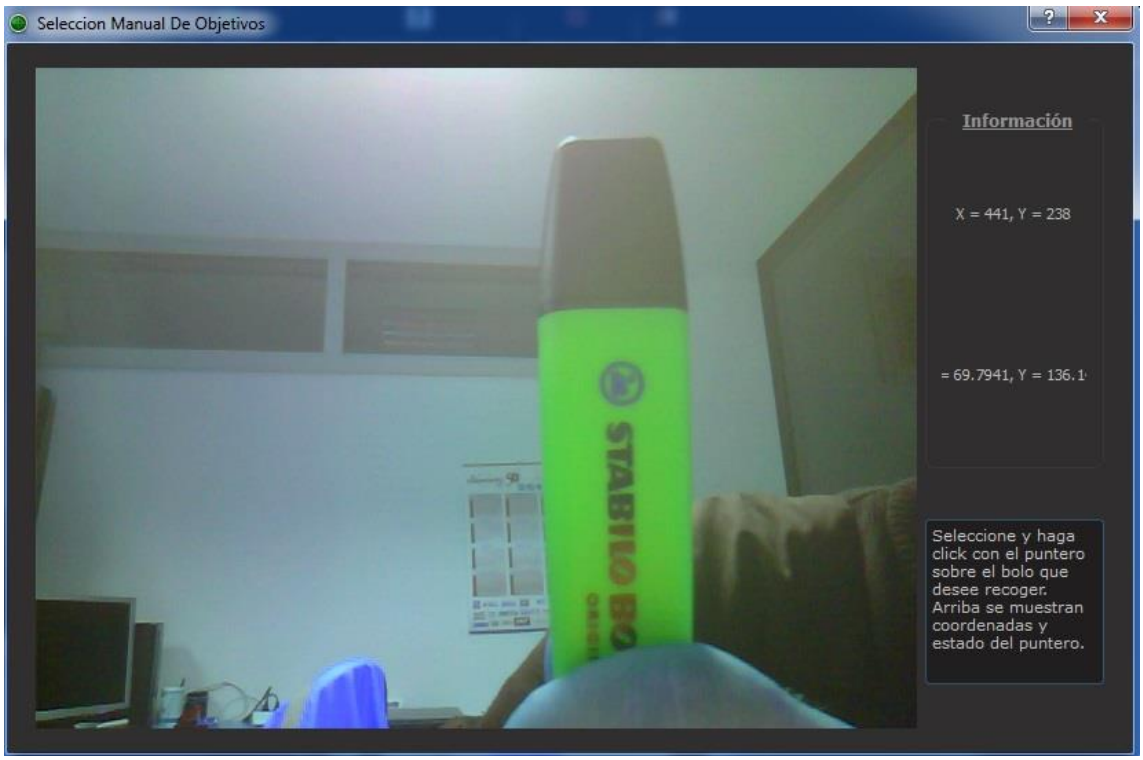
SELECCIÓN DE OBJETIVO MANUAL

Si se selecciona esta acción, se desplegará una ventana nueva en la que se dispondrá de un marco en el que se reproduce la captura de imágenes de la webcam en tiempo real y un puntero con el que podemos seleccionar manualmente el objeto a recoger. En el cuadro informativo de la derecha se recogen las coordenadas en píxeles en las que se encuentra el puntero en tiempo real.

En las imágenes que se incluyen a continuación primero se recoge un momento en el cual el usuario se encuentra navegando con el puntero por la imagen, sin ninguna selección realizada. En la segunda imagen se mostrará el resultado de seleccionar una imagen: En el cuadro informativo de la derecha se mostrarán las coordenadas reales del objeto y se enviarán estas al brazo robótico para proceder a su recogida:



Navegando por el marco en el que se muestra la imagen. Sin selección realizada



Tras realizar una selección se muestran las coordenadas (x,y,) reales del objeto detectado

CONTROL MANUAL DE BRAZO ROBÓTICO

Al seleccionar esta acción del menú desplegable en la ventana principal , o bien en el menú de la barra de herramientas, se abrirá una ventana con seis controladores en forma de dial con los que podremos mover cada uno de los seis servos de los que dispone el brazo. Al igual que la selección manual de objetivos, esta acción trabaja de forma independiente:

Diales controladores para las articulaciones del brazo robótico