Prácticas extracurriculares en CFZ Cobots

Ingeniería Robótica, Universidad de Alicante

Propuesta de un método alternativo de programación por guiado basado en visión artificial

---

Autor:
José Miguel Torres Cámara

Tutores:
Jorge Vergara Roa (empresa)
Fernando Torres Medina (UA)

Índice

1. Motivación, objetivos y estructura

1.1. Programación por guiado

Actualmente se distinguen dos principales maneras de programar un robot: programación textual y programación por guiado. Mientras la **textual** consiste en escribir código de forma similar a como se hace en lenguajes de programación estándar, en la **programación por guiado** se mueve el robot hacia las posiciones que debe de alcanzar en cada momento, bien sea de forma ***discreta***, es decir, moviendo al robot manualmente o desde la consola de programación a todos los puntos uno a uno e ir grabándolos independientemente o de forma ***contínua***, en la que directamente se le coloca la herramienta que usará en la aplicación y se realiza la tarea que deberá reproducir el robot con la herramienta ya colocada en el extremo del mismo. Este último método es muy común en aplicaciones donde se requiere cierta habilidad o *arte* en la tarea, donde es complicado reproducir los movimientos que seguiría el humano *artista* mediante los otros métodos tradicionales. Algunas de estas tareas serían operaciones de pintura, soldadura, etc. Dada la versatilidad que supone este método así como su gran cantidad de aplicaciones y facilidad de programación de cara al usuario, durante el proyecto me centraré en él.

A pesar de todas las ventajas que supone este tipo de programación por guiado, también conlleva **imprecisión** en los movimientos (ya que lo mueve un humano) así como cierta **incomodidad** ya que el artista no estará ejecutando la tarea con su herramienta *al aire*, si no que tiene que arrastrar el robot tirando de ella, lo que puede suponer **limitaciones espaciales** así como un evidente **esfuerzo extra**, desenvocando en que sea prácticamente imposible realizar la tarea igual que si la herramienta no estuviese unida al robot.

A causa de esto, pensé en formas de mejorar el resultado que proporcionaba este tipo de programación tratando de evitar costes excesivos y manteniendo su sencillez de uso de cara al usuario, llegando así a la conclusión de que las opciones con más posibilidades era lograr que **el robot pudiese almacenar las posiciones a seguir mientras el usuario realiza la tarea con la herramienta totalmente desacoplada del robot**. De esta forma se reducirían parcialmente todos los *contras* comentados en el párrafo anterior.

1.2. Objetivos generales

Como principal objetivo tengo que, mis prácticas sirvan como un **inicio de proyecto real**, ya que en un mes no puede desarrollarse una alternativa de programación realmente práctica por lo que el objetivo, evidentemente, **no es lograr una alternativa real a los métodos de programaciones actuales**.

Con el objetivo de desarrollar este *primer paso* hacia un nuevo método de programación, pretendo desarrollar un **prototipo mínimo** que demuestre que es posible implementarlo, que permita descubrir posibles mejoras y que cumpla con algunas de las características que considero más importantes para este propósito, las cuales son:

- **Robustez**: El prototipo debe de comportarse de forma similar en la mayor parte de los ambientes donde se ponga a prueba, tanto en un entorno controlado de laboratorio como en las muchas condiciones que puede suponer cualquier entorno industrial, ya que este último será su verdadero campo de aplicación
- **Efectividad**: Debo lograr un seguimiento lo más cercano a la realidad que sea posible ya que, de no ser así, no tendría sentido utilizar esta propuesta. Con ese propósito, los movimientos del robot deberán ser todo lo suaves y naturales que se pueda, dando por sentado que se debe seguir la trayectoria del identificador fielmente. Otro punto importante es la velocidad con la que el robot puede seguir al objetivo sin perderlo, ya sea porque no pueda frenar de lo rápido que va o porque el objetivo vaya demasiado rápido
- **Sencillez**: Como mencioné anteriormente, una de las principales ventajas de la programación por guiado es su carácter *user-friendly*, cosa que me gustaría conservar e incluso mejorar con esta aplicación. Quiero que el usuario final sólo tenga que montar el dispositivo en el robot e iniciar la ejecución del programa del robot
- **Comodidad**: Trataré de hacer un dispositivo embebido, lo más pequeño y manejable que me sea posible

1.3. Estructura del informe

Con el propósito de lograr la mayor claridad posible en este informe, comenzaré explicando **hasta donde llegué** en el proyecto así como los **recursos software y hardware** en los que me basé para después seguir comentando **otras opciones** que contemplé durante su desarrollo así como algunas de las **pruebas** que hice para cada una de las partes del mismo. Luego **analizaré los resultados** y seguiré mencionando algunos de los **trabajos futuros** que podrían realizarse para seguir avanzando. Por último, acabaré con las **conclusiones** que saqué tras realizar todo el proyecto.

2. Resultados

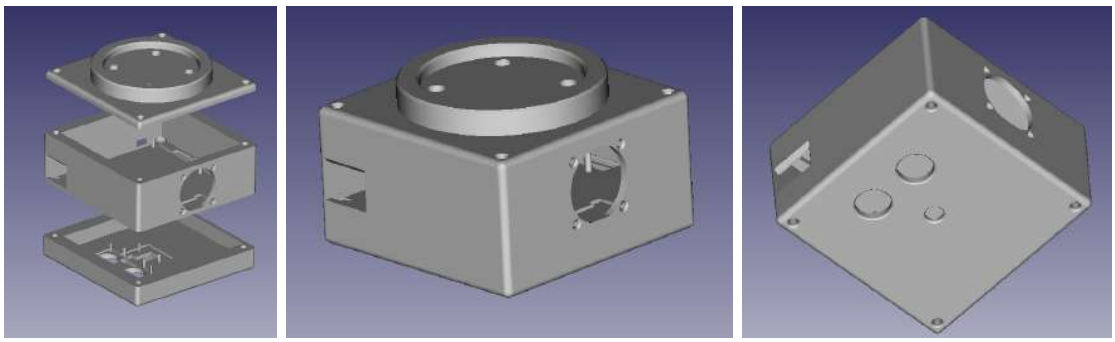
Tras finalizar el periodo de prácticas logré un prototipo mínimamente funcional que era capaz de hacer que el robot siguiese al identificador (un triángulo verde), almacenar los movimientos que se hacen relativos a su posición inicial y luego repetirlos desde cualquier posición desde la que se parta (a no ser que ésta provoque la existencia de singularidades, colisiones o salidas del espacio de trabajo en algún momento de la trayectoria, es decir, limitaciones propias del robot). Todo esto lo posibilita la existencia de cuatro partes principales: una **herramienta para el robot** que controlada por una

Raspberry Pi, un **programa del robot** y **dos programas en Python** ejecutados en la Raspberry. Cada una de estas cuatro partes es la mejor versión que he podido crear de las mismas y, a continuación, paso a explicar cada una de esas versiones definitivas.

2.1. Herramienta desarrollada

Como he mencionado, he desarrollado una herramienta que puede acoplarse al extremo de cualquier modelo de los brazos robots de Universal Robots. Esta herramienta cuenta de tres partes:

- **Placa superior**: Permite anclar la herramienta a la brida de herramienta de los brazos robóticos UR
- **Placa intermedia**: Contiene el controlador, es decir, la Raspberry Pi (en concreto una 3B), que es la que se encarga de procesar la información captada por los sistemas de percepción y corregir la pose del robot. En esta parte de la herramienta también hay alojado un pequeño ventilador para refrigerar el sistema ya que, al estar todo encapsulado, se acumula bastante calor. También incorpora un divisor de tensión que adapta la señal que proporciona el sensor de distancia (un HC-SR04, basado en tiempo de vuelo de ultrasonidos). Evidentemente, esta placa cuenta con accesos desde el exterior al ventilador, al puerto ethernet y al microUSB de alimentación
- **Placa inferior**: Esta última capa contiene tanto el módulo de cámara de la Raspberry (concretamente la revisión 2 del mismo) como el HC-SR04. La existencia de esta placa permite *esconder* tanto los cables como las PCBs del exterior, dejando únicamente visible las partes estrictamente esenciales. De este modo, logro una mayor protección para el sistema así como un acabado mucho más estético



(a) Desglose de placas (b) Vista superior (c) Vista inferior

Figura 1: Diseño de la versión definitiva del soporte/herramienta

2.2. *Script* de programación (en Python) para la Raspberry

Este programa es el responsable de configurar la Raspberry tal y como se precisa en cada ejecución del programa, leer datos de la cámara y del sensor de distancia y calcular las correcciones que deben hacerse en en las posiciones X e Y del extremo del robot para reducir el error y enviárselas al robot para que se mueva, siguiendo al triángulo verde utilizado como marcador. También almacena estas correcciones en un archivo para su posterior lectura y reproducción de la tarea.

2.2.1. Dependencias y configuraciones previas

Existe una gran cantidad de recursos *software* que la comunidad ha creado y que facilitan enormemente el desarrollo de innumerables *scripts* que utilizan alguna de esas funcionalidades. Con tal de aprovechar esas **bibliotecas**, he utilizado algunas de ellas, como son:

- **time**: Para contar el tiempo (función *time()*) que pasa entre que se emite la señal con el sensor de distancia hasta cuando se recibe y para hacer pausas (función *sleep()*) en varias partes del código
- **cv2** (OpenCV): Para capturar *frames* del módulo de cámara así como para realizar todo el procesado de imagen
- **pymodbus.client.sync**: Para escribir los cuatro registros Modbus del servidor que se ejecuta sobre el armario de control del robot. Utilizo esos cuatro registros para indicar las correcciones que deben hacerse en la posición del extremo del robot para que este siga estando sobre el triángulo verde
- **os.path**: La necesito para utilizar la función *isfile()*, que me permite comprobar si en el directorio actual ya existe un archivo con correcciones almacenadas de otro proceso de programación previo. En caso de detectar que ya existe un archivo, se le comunica al usuario y se da la opción de abortar la ejecución o de continuar y sobreescribirlo
- **RPi.GPIO**: Me permite utilizar los pines GPIO (*General Purpose Input/Output*) para tomar lecturas del sensor de distancia

Por lo referente a la **secuencia de inicialización** del programa, comienzo inicializando algunas **constantes de configuración**, como son la dirección IP del robot y el puerto donde este tiene abierto el servidor Modbus-TCP o como la distancia aproximada (en Z) entre el sensor de distancia y la cámara, que empíricamente determiné que rondaba los 6 milímetros. Otra constante de mucha importancia es la longitud que tendrá el filtro de media móvil (LONG_FILTRO) que utilizo. Por defecto está establecida a 4. Elevarla supondrá más estabilidad y robustez pero menos velocidad y precisión. También se establece un intervalo (AREA_LOW y AREA_MAX) en el que debe estar el area del triángulo verde (identificador) percibido por la cámara. Por último, también se

permite configurar el tamaño de la ZONA_TOLERANCIA. Esta constante es el número por el que se dividirá el tamaño del *frame* captado, creando un rectángulo en el centro del mismo con las dimensiones que resulten de esas divisiones. Cuando el centro del identificador esté en ese rectángulo, se considerará que el robot está sobre el triángulo a seguir, por lo que no se moverá. Esto quiere decir que disminuir el tamaño del rectángulo (es decir, aumentar el valor de esta constante) supondrá más precisión pero aumentarlo demasiado hará que el rectángulo sea muy pequeño, pudiendo hacer que no se logre posicionar el centro del identificador en el mismo, haciendo que el robot oscile en torno a la posición sobre la que debería mantenerse estático.

Tras definir algunas funciones auxiliares (que explico en la siguiente sub-sub-sección del informe), continúo con **otras inicializaciones** que NO deben ser modificadas. La primera de estas es la del **sistema de visión**, es decir, lograr acceder al módulo de la cámara y leer un *frame* con éxito. En caso de no conseguirlo, se le indicará al usuario y el programa finalizará automáticamente. Se continúa con el **sensor de distancia**, inicialización para la que deshabilito las advertencias de posibles fallos que se hayan cometido en otros programas que usasen los GPIO y no los reconfigurasen debidamente. También indico que utilizaré la numeración de la placa y especifico los pines a los que se conectarán los pines *Trig* (disparador) y *Echo* (receptor) del sensor, además de configurarlos como salida y como entrada, respectivamente.

Tras configurar los sistemas de percepción, continúo con las **comunicaciones**, es decir, abriendo el *socket* que me permite escribir los registros del servidor Modbus del robot. En caso de que sucediese algún error durante la conexión, se informará al usuario y se finalizará la ejecución de forma automática. Tras esto, abro el **archivo de escritura**, que estará en el mismo directorio que el programa y se llamará çorrecciones.txt". Comienzo comprobando si ya existe un archivo con ese nombre en el directorio. En caso positivo se informará al usuario y se dará la opción tanto de abortar la ejecución como de continuarla (y sobreescribir el archivo).

Continúo las inicializaciones declarando algunas variables para el **procesado de imagen**, tales como el intervalo en HSV que se considerará verde, que determiné empíricamente por mi cuenta. También leo las dimensiones de los *frames* y asigno valores iniciales a algunas variables por cuestiones de optimización de código como evitar declaraciones y destrucciones iterativas que consuman tiempo y coste computacional innecesariamente.

Por último, **adapto algunas variables del algoritmo al entorno** según la distancia que haya entre la superfície más próxima a la cámara y la cámara, ya que de esta magnitud dependerán el tamaño de las correcciones que se le ordenen al robot. Tras inicializar algunas variables por los mismos motivos que en la inicialización del procesado de imagen, tomo varias medidas del sensor de distancia (el doble de elementos contenidos en el filtro de media móvil), de las cuales hago la media, considerando al resultado de la misma la distancia cámara-superficie anteriormente mencionada tras añadirle la corrección de la constante de configuración. Por último, calculo los desplazamientos máximos que debería hacer el robot en X e Y para seguir al identificador. Estas correcciones será la distancia real que hay entre lo que en el *frame* aparece en el centro del mismo y en la mitad de los bordes derecho o izquierdo (corrección en X) y en la mitad de los bordes

superior o inferior (corrección en Y). La relación entre la altura y esas distancias las obtuve con cálculos trigonométricos sencillos que no incluí en el códiogo por aligerarlo. Estos cálculos utilizan los ángulos de visión de la cámara y la ley del seno.

Sabiendo que, **según [1], el ángulo de visión de la cámara es de 62.2° en X y de 48.8° en Y**, es posible calcular las correcciones de la siguiente manera:

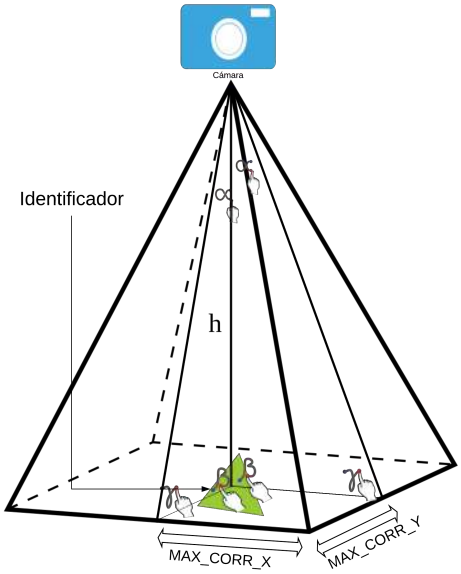


Figura 2: Representación del área de visión de la cámara

Cálculo de la corrección máxima en X

Ángulo de visión en X = 62.2°

$\alpha = \frac{62,2}{2} = \mathbf{31.1}°$

h = lectura del sensor de distancia

$\beta = 90°$

$\gamma = 180 - 90 - 31,1 = 180 - 121,1 = \mathbf{58.9}°$

Teorema del seno:

$$\frac{CORR\_MAX\_X}{sen(\alpha)} = \frac{h}{sen(\gamma)}$$

$$\mathbf{CORR\_MAX\_X} = \frac{h \cdot sen(\alpha)}{sen(\gamma)} =$$

$$= \frac{h \cdot sen(31,1)}{sen(58,9)} = \mathbf{0.6032386 \cdot h}$$

Cálculo de la corrección máxima en Y

Ángulo de visión en Y = 48.8°

$\alpha = \frac{48,8}{2} = \mathbf{24.4}°$

h = lectura del sensor de distancia

$\beta = 90°$

$\gamma = 180 - 90 - 24,4 = 180 - 114,4 = \mathbf{65.6}°$

Teorema del seno:

$$\frac{CORR\_MAX\_Y}{sen(\alpha)} = \frac{h}{sen(\gamma)}$$

$$\mathbf{CORR\_MAX\_Y} = \frac{h \cdot sen(\alpha)}{sen(\gamma)} =$$

$$= \frac{h \cdot sen(24,4)}{sen(65,6)} = \mathbf{0.4507694 \cdot h}$$

2.2.2. Funciones auxiliares

En pos de un código lo más limpio y claro posible, he definido varias funciones auxiliares que me permiten evitar la inclusión en el *cuerpo principal* del programa de código que dificultarían su lectura. Estas funciones son:

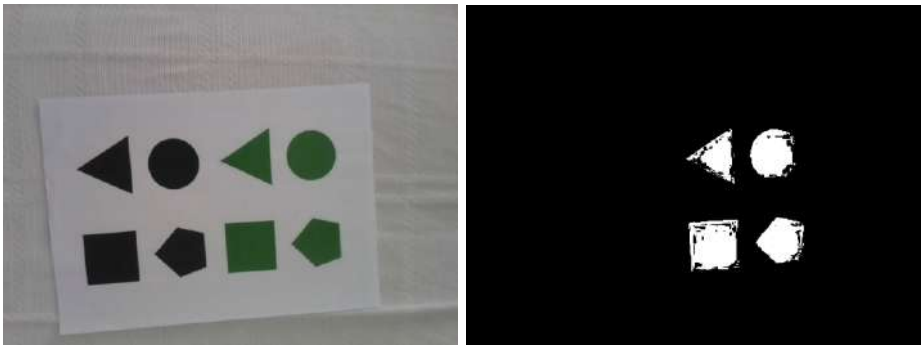
- **media_movil()**: Me permite **añadir robustez y estabilidad a las lecturas de posiciones de la característica en el *frame*** actual. Recibe como argumentos de entrada dos vectores que almacenan el historial de las últimas posiciones (en píxeles) del identificador. Uno de los historiales es sólo de las coordenadas X y, el otro, de las Y. Los historiales albergarán tantas medidas como indique la constante de configuración LONG_FILTRO. Cuanto más alto sea su valor, más robusto será el proceso a lecturas erróneas (ruido puntual) pero se seguirá a la característica con menor precisión y velocidad. Un valor aconsejable es 4. **A partir de los dos historiales, calcula la media** y devuelve los resultados
- **detectarTriangulo()**: **Evalúa si un contorno es o no un triángulo**. Basada en el **algoritmo de Ramer-Douglas-Peucker**, que simplifica un segmento curvo aproximando varios segmentos lineales. Esto permite saber cuántos vértices *principales* se tienen en un contorno determinado. El contorno a analizar es su único argumento de entrada. Este contorno estará formado por una serie de puntos obtenidos tras el procesado de la imagen. Simplemente realizo la aproximación por segmentos lineales y, en caso de haberse aproximado con tres segmentos, confirmo que la forma es un triángulo devolviendo un valor *True*. En caso de que la aproximación no resultase en tres contornos, se devuelve *False*
- **medirDistancia()**: Hace uso de los pines GPIO para **medir la distancia desde el sensor de distancia (y de la cámara) hasta el objeto más cercano** (superfície donde esté el identificador). Pone a nivel bajo el pin que ordena que se emitan señales desde el sensor y se hace una pequeña pausa (2 microsegundos) para que de tiempo a que se haya apagado. Después, se enciende dicho pin, ordenando una emisión de ultrasonidos para continuar con otra pausa (de 10 μs) para asegurar la emisión antes de volverlo a apagar. Después se comienza a contar tiempo cuando se recibe un valor alto a través del pin que indica si el receptor recibe o no. La cuenta se detiene cuando para de recibir. Por último, se calcula el tiempo transcurrido entre estos dos eventos y se utiliza ese dato para calcular la distancia hasta el objeto más próximo en metros. El resultado de esta conversión será el valor que devuelva la función

2.2.3. Bucle principal: Procesado de imagen

El **objetivo** del procesado de imagen que realiza mi programa es **obtener el centroide del contorno (sucesión de puntos) del triángulo verde más grande** que aparezca en el *frame*. Con este propósito, desarrollé un procesado de imágen divisible en dos partes: discriminación por color y por forma y tamaño.

La **discriminación por color** fue por la que empecé, detectando el **verde**, ya que tras numerosas pruebas, determiné que era el color que se reconocía con más robustez antes cambios de iluminación y para áreas de distintos tamaños y formas. Para esta parte, comienzo **pasando de RGB a HSV**, ya que en este espacio de color se separa el tono de la luminosidad, logrando muchísima más robustez frente a variaciones en la luminosidad. Además, dada su distribución cilíndrica de los colores, es más sencillo establecer un único intervalo en el que se localizan los colores deseados. Esta transformación la realizo con la función *cvtColor()* de OpenCV[3].

Una vez convertida la imagen a HSV, creo una **imagen binaria en la que los píxeles cuyo valor entren en el intervalo establecido en las constantes de configuración aparecerán en blano y, el resto, en negro**. La ubralización la permite el método *inRange()*.

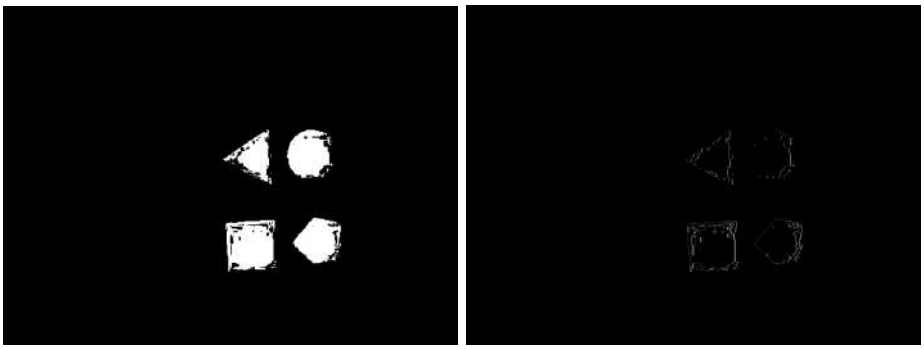


(a) *Frame* original (b) *Frame* binarizado

Figura 3: Umbralización por color

Sobre esta imagen binaria, con *findContours()*, también de OpenCV[3], **obtengo todos los contornos que envuelven a las figuras** que en ella aparecen, que incluirán tanto las áreas verdes que aparezcan como los ruidos que figuren en la imagen por cualquier causa.

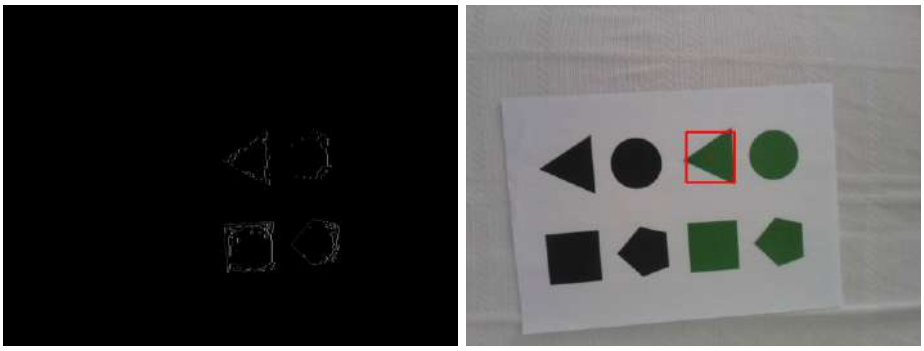
Una vez obtenidos todos los contornos, paso a **discriminarlos por tamaño y forma**. Se recorre el vector de contornos obtenido (la variable *cnt*) para evaluar el área que encierra cada uno de ellos, siendo esta la primera "barrera"que deben pasar para ser considerados el identificador. Si el contorno que se esté evaluando es el mayor hasta el momento, **se comprueba si además es un triángulo** (con mi función *detectarTriangulo()*). En caso de cumplir ambas condiciones, se **almacena tanto el contorno como el área** que encierra para poder disponer de los mismos después.



(a) *Frame* binarizado (b) Contornos extraidos

Figura 4: Extracción de contornos

Tras analizar todos los contornos detectados, se dispondrá del que corresponda al triángulo verde más grande de la imagen, habiendo discriminado tanto triángulos de otros colores como ruidos pequeños, posibles reflejos, otras formas verdes, etc.



(a) Contornos extraidos (b) Detección definitiva

Figura 5: Discriminación por forma

El hecho de **haber escogido al triángulo como forma discriminadora vino dado tras un análisis con varias formas** (triángulos, cuadrados, rectángulos, pentágonos y círculos) **de varios colores y tamaños**. Traté de detectarlos desde distintas distancias y orientaciones y el que se comportaba mejor casi siempre era el triángulo. Las figuras con más aristas se confundían cuando las enfocaba con inclinaciones elevadas o desde distancias relativamente largas.

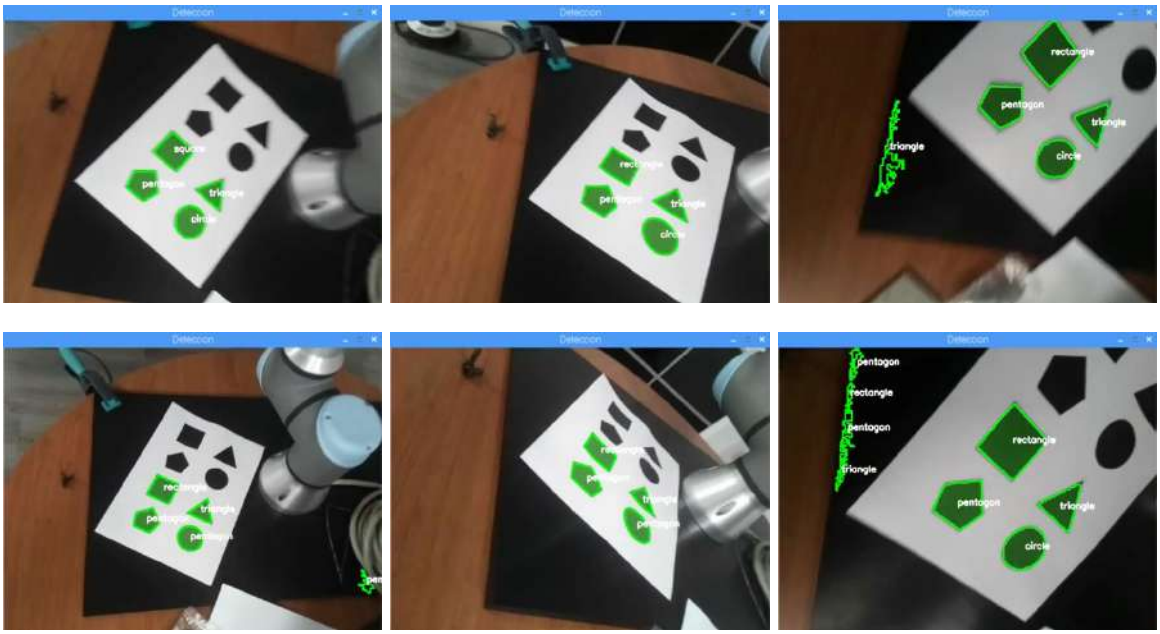


Figura 6: Detección y clasificación de varias formas con varios ángulos y tamaños

Tal y como se aprecia en las imágenes (extraídas de un vídeo en el que detectaba y clasificaba formas *on-line*), **el círculo es el que más se confunde**, normalmente con figuras de muchos lados (considero que es un círculo cuando se detectan 6 o más vértices). Sin embargo, en ningún momento se confunde al círculo con un triángulo. Dado que, como he mencionado, las imágenes son de un vídeo, algunos frames eran muy borrosos, por lo que no he incluido uno de ellos en el que el pentágono se confunde con un círculo. Estos dos hechos dejan como **mejores opciones** a los **cuadrados/rectángulos y a los triángulos**. Dado que, como he mencionado, **el círculo nunca se ha confundido con un triángulo** pero sí con un cuadrado, opté por descartar a los cuadriláteros, ya que podría darse el caso en el que hubiese un círculo verde en la escena y este deba ser discriminado, cosa que se logra utilizando triángulos en lugar de cuadrados o rectángulos. También se observa que los ruidos son formas pequeñas, por lo que se discriminarán, así como que otras formas de otros colores, como negras, no son tenidas en cuenta.

La última discriminación consiste en comprobar si el **área del contorno resultante está en el intervalo definido** en las constantes de configuración(AREA_TOP y AREA_LOW). Esto permite detectar cuándo ningún contorno cumplía con todos los requisitos (area_max seguirá siendo 0, estando por debajo de AREA_LOW) así como detectar posibles errores en los que se considere un contorno exageradamente grande. Si el contorno está en el intervalo definido, **calculo su centroide** (*[cx, cy]*) utilizando los momentos del mismo.

En caso de que no haya ningún contorno triangular que encierre un área de un tamaño aceptable, se notifica y se centra toda la potencia de procesado en encontrarlo, haciendo un *continúe*, obligando a reiniciar el bucle (que vuelve a

tomar otro frame y a procesarlo) en lugar de seguir con el cálculo y el procesado de las correcciones.

2.2.4. Bucle principal: Cálculo de correcciones

Tras el procesado, he obtenido los valores de *cx* y de *cy*, por lo que ya se puede continuar con el siguiente paso: **transformar la posición del identificador en la imagen en correcciones para la posición del extremo del robot**.

Antes de calcular nada, se **guardan las coordenadas en píxeles del identificador en los historiales** *historial_x* e *historial_y* y se comprueba su longitud. En caso de ser menor que la configurada en la constante LONG_FILTRO, se vuelve a iniciar todo el procesado de imagen para tomar otra medida de la posición del identificador, de forma que hasta no haber llenado un "*buffer*"del filtro, no se inicia el funcionamiento normal del bucle.

En caso de que se disponga de suficientes medidas, se elimina la más antigua (el elemento 0 de los historiales) y **se calcula la media**, considerando el resultado de la misma la posición actual del identificador, siendo esta la que se utiliza para calcular las correcciones.

Las correcciones en cada eje (X e Y) se calcularán sólo si en ese eje el identificador está fuera de la zona de tolerancia (rectángulo verde de la figura 6), es decir, fuera del rectángulo que se establece en el centro del *frame* en el que se considera que, si el centro del identificador (punto rojo de la figura 6) está dentro, el robot está sobre el mismo, caso en el que a la corrección en ese eje tomará un valor de 0. El uso de esta zona viene dado porque, **si no la utilizaba o si era demasiado pequeña, el robot oscilaba** sobre la posición objetivo. En la siguiente imagen es una catorceava parte del temaño total del *frame*:

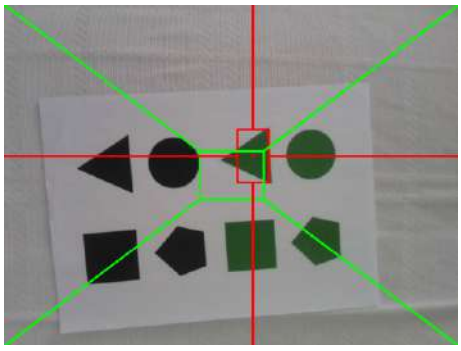


Figura 7: Zona de tolerancia

En caso de ser necesaria corrección en alguno de los ejes, **calculo y normalizo (entre 0 y 1) el error de la posición del centro del identificador**, asumiendo como posición deseada el píxel central del *frame*. **A ese valor entre 0 y 1 lo multiplico por la corrección máxima calculada** en el eje que corresponda (DESPLAZAMIENTO_MAX_X para el eje X y DESPLAZAMIENTO_MAX_Y para el eje Y), **obteniendo**

el desplazamiento que el robot deberá hacer para posicionar la cámara (que gracias al diseño de mi herramienta, tiene el eje Z alineado con la muñeca del robot) sobre el identificador.

2.2.5. Bucle principal: Procesado de correcciones

Tras haber calculado las correcciones a hacer (*correccionX* y *correccionY*), paso a procesarlas. Este procesado consiste únicamente en **comunicárselas al robot por Modbus-TCP y en anotarlas en el archivo** *correcciones.txt*.

Para **comunicarle los resultados al robot** utilizo cuatro registros Modbus. El uso de cuatro registros en lugar de dos es porque estos sólo pueden contener valores positivos, así que **escribo las correcciones a hacer en X y en Y en los registros 128 y 129 y sus signos en los 130 y 131**, escribiendo "1"si es positivo y "0"si es negativo. Por lo referente a los registros 128 y 129, escribo el valor absoluto del resultado de dividir las correcciones multiplicadas por 1000 y divididas por 2. Las divido entre 2 para evitar sobreoscilación durante el seguimiento, sacrificando velocidad. El hecho de multiplicar las correcciones por 1000 es debido a que los registros sólo pueden contener números enteros, por lo que lo escribo en milímetros en lugar de metros cosa que luego, en el programa del robot, tengo en cuenta.

Por último, también **escribo las correcciones en el archivo**. Para no perder tiempo, no vuelvo a hacer las operaciones de multiplicar por 1000 y dividir entre 2, cosa que SÍ que hago en el programa de reproducción, ya que este conlleva una carga computacional mucho menor.

2.2.6. Proceso de finalización

Con tal de lograr una finalización lo más limpia posible, he tratado de contemplar todas las posibles excepciones, tales como errores en las diferentes inicializaciones, peticiones del usuario o una interrupción de teclado (Ctrl + C). Si ocurre cualquiera de estas cosas, **libero la cámara** (liberando el archivo de captura así como la memoria reservada para su uso), **cierro el *socket*** que utilizo para Modbus/TCP (sólo si se ha conseguido abrir durante las inicializaciones), **libero el archivo de escritura** y **reseteo los GPIO** utilizados durante el programa (11 y 13 según la numeración de la placa para el sensor de distancia).

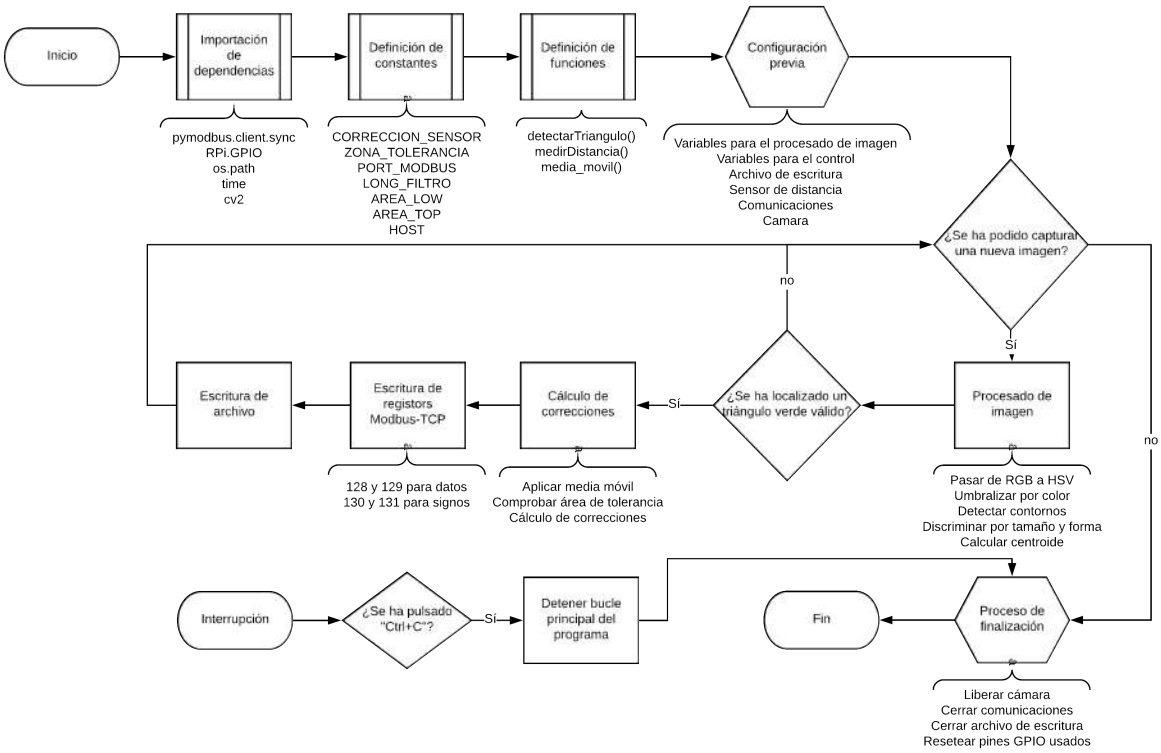


Figura 8: Diagrama de flujo de *programar.py*

2.3. *Script* de reproducción (en Python) para la Raspberry

Este programa se ejecutará una vez se haya *programado* al robot mediante una ejecución completa, exitosa y sin errores del anterior programa (caso normal), de forma que se haya creado el archivo. Este *script* **permite que se reproduzcan los movimientos realizados durante la programación** sin perder el tiempo que se perdió con el procesado de imagen y cálculo de correcciones, además de consumir menos recursos, ya que exige mucha menos memoria y potencia de cálculo para su ejecución.

Dado este método de reproducción, **no hace falta cargar un programa distinto en el robot**, ya que basa las comunicaciones en el mismo funcionamiento del programa anterior, escribiendo los mismos 4 registros Modbus.

2.3.1. Dependencias y configuraciones previas

Al igual que en el caso del programa anterior, utilizo algunas librerías ya desarrolladas por la comunidad. Aunque menos que en el caso anterior, todas también se usan en el

otro programa. Las dependencias que importo son **os**, **time** y **pymodbus.Client.sync**. Les doy los mismos usos que en el *script* para programar al robot.

Además de incluir estas dependencias, antes de iniciar el funcionamiento normal del programa, realizo una serie de **configuraciones previas**, como son la declaración de las **constantes de configuración**, como la **IP del robot** (HOST), **el puerto donde** el mismo **tiene el servidor Modbus** (PORT) y el **tiempo** que se deja **entre cada escritura de registros** (DELAY_ITERACIONES). Este tiempo está configurado a 0.048496 s porque es el tiempo medio que se tardaba en una iteración completa del bucle del *script* de programación. De este modo, el comportamiento durante al reproducción será lo más similar posible al que tuvo lugar durante la programación.

A continuación, al igual que hacía en el otro programa, **abro el archivo de lectura** donde se almacenaron las correcciones realizadas durante la programación y **abro un *socket*** para comunicarme por Modbus-TCP con el robot. Tras este último paso, **escribo un 0 en los registros de correcciones** del robot para que, en caso de que por cualquier motivo, tuviesen otro valor, el robot no se moviese cuando no debe hacerlo. También tengo en cuenta posibles errores, como que el archivo no exista o que haya algún fallo durante la conexión con el servidor, en cuyo caso, se le comunicaría al usuario y se abortaría la ejecución del programa.

2.3.2. Bucle principal

Este programa únicamente **lee las correcciones almacenadas en el archivo, las transforma de tipo y las escribe en los registros Modbus** por Modbus-TCP que usaba la otra aplicación, permitiendo que el robot las lea y corrija su posición. Entre escritura y escritura se realiza un pequeño *delay* que simula el tiempo que se tardó en todo el procesado de imagen y cálculo de correcciones en el otro programa, reproduciendo todo lo fielmente posible los movimientos realizados.

El usuario **podrá regular la velocidad de reproducción de movimientos desde la barra de velocidad de la consola de programación** y, como opción excepcional, reduciendo el tiempo de *delay*, aunque la diferencia será mínima.

2.3.3. Proceso de finalización

Para finalizar este programa, repito una parte de la secuencia de finalización del otro programa. Concretamente, la de **cerrar el *socket*** y **liberar el fichero** desde el que se leen las correcciones almacenadas.

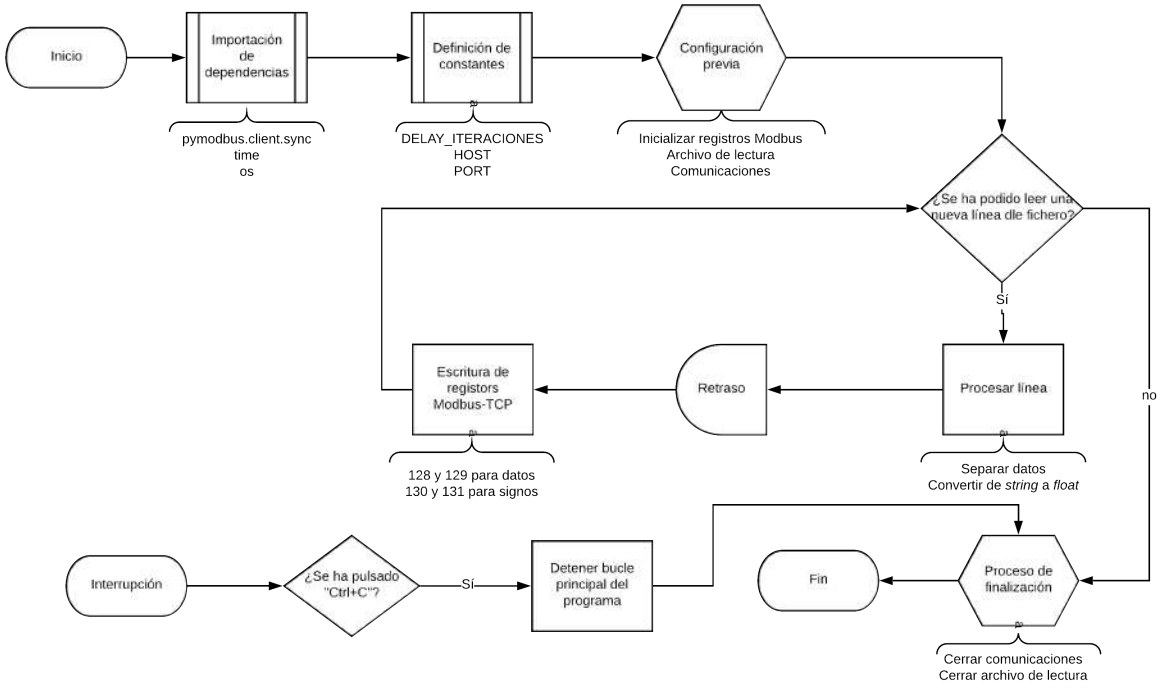


Figura 9: Diagrama de flujo de *reproducir.py*

2.4. Programa del robot

Para llevar a cabo las correcciones calculadas por la Raspberry, desarrollé un pequeño programa para el robot que procesaba los datos enviados a través de Modbus-TCP al robot y hacía uso de algunas de las funciones que los UR ponen a disposición de los desarrolladores, como *get_actual_tcp_pose()*, *read_port_register()*, *pose_trans()* o *MoveL()*.

Este programa es tremendamente sencillo pero **permitía ahorrarle a la Raspberry una considerable carga computacional** y simplificar muchísimo los códigos, cosa que era uno de mis principales objetivos, ya que el cuello de botella reside ahí, ya que se tarda sobre 50ms en llevar a cabo una iteración del bucle de control mientras que el robot es capaz de actuar recibiendo órdenes a 125Hz (una cada 8ms). Es debido a este ahorro temporal por lo que decidí utilizar este programa, ya que también **disponía de otras versiones en las que no era necesario ejecutar absolutamente nada desde el robot**. Además, al obligar a que cada programa se ejecute con una configuración de instalación concreta, **permite reducir la posibilidad de que el usuario final tenga problemas** que le puedan resultar complicados de solucionar, como algunos relacionados con la configuración de la red local RPi - robot.

El programa simplemente **lee los registros Modbus** de correcciones escritos por el programa de la Raspberry y, en caso de ser distintos de 0 (para no perder tiempo

inútilmente), **les asigna el signo que les corresponda** leyendo el otro par de registros Modbus, **transforma la posición actual del TCP en la deseada y ordena un movimiento** lineal hasta la misma, reduciendo el error en la medida que la Raspberry lo ordene.

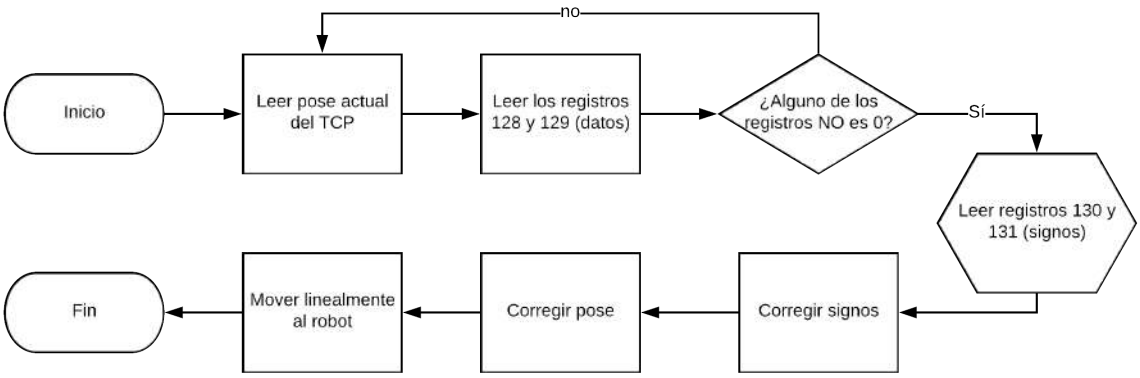


Figura 10: Diagrama de flujo del programa del robot

3. Recursos utilizados

3.1. Dispositivos involucrados

Como ya he mencionado, el elemento central de todas las pruebas que he realizado, ha sido una **Raspberry Pi** (modelo 3B concretamente). Este ordenador lo manejaba a través de una conexión VNC desde mi portátil. El otro factor constante en mis experimentos era un robot **UR3** de la serie CB de Universal Robts.

Por otro lado, he ido utilizando varios *accesorios* para las pruebas, como la versión 2.1 de la **cámara de la Raspberry**, un módulo **HC-SR04** para medir distancias, una IMU **MPU6050** y, para algunas pruebas, un **Arduino UNO**.

3.2. Recursos software

Dejando de lado el **Polyscope ejecutado sobre el Debian del UR3**, he estado utilizando **Raspbian** corriendo sobre la RPi. Así mismo, para realizar pruebas de cada programa utilizaba **Python** y luego transcribía el código definitivo a **C++**, para poder compilarlo y obtener un binario que pudiese ejecutarse más rápido de lo que un script interpretado por el intérprete de Python podría. Sin embargo, dejé las versiones finales

en Python porque consideré que se alcanzaban velocidades aceptables, además de la inmensa diferencia a la hora de leer y depurar el código, siendo en el caso de Python mucho más sencillo.

Respecto a recursos más relacionados con mi aplicación y pruebas, he utilizado la librería **Armadillo** para C++ para implementar parte de los códigos con una **carga matemática** más compleja (como cálculo pseudoinversas, etc.). También usé **OpenCV** para llevar toda la parte de **visión**, tanto en C++ como en Python. Por lo referente a las **comunicaciones**, usé los paquetes *socket* y *pymodbus* en Python (para TCP y Modbus respectivamente) así como algunos otros de uso común como ***time*, *numpy*, *struct* , *os* o *RPI.GPIO***.

Por otra parte, para **testear las comunicaciones** utilicé la versión 3 de ***socketTest***, un programa que permitía a la máquina que lo ejecutase ser cliente o servidor TCP en la IP y puerto que se le configurase. Para Modbus hice uso de ***diagslave***, que actúa como servidor. Ambos están disponibles tanto para Windows como para GNU/Linux.

4. Otras versiones de la herramienta

Para realizar pruebas era **necesario tener la cámara en el extremo del robot de forma que la cámara viajase con el mismo**, ahorrando el máximo de transformaciones entre sistemas de coordenadas o conversiones de desplazamientos innecesarias. En un inicio utilicé simplemente **cinta adhesiva**, aunque para las primeras pruebas fijé la Raspberry al robot con **bridas** y limité el uso de la cinta para sujetar la cámara al extremo del robot:

(a) Sujeción con cinta adhesiva (b) Sujeción con bridas

Figura 11: Sujeciones provisionales

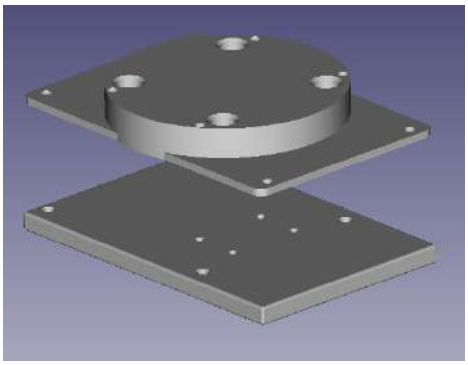
Sin embargo, como es evidente, esta no fue la sujeción final. Tras numerosos bocetos, **diseñé varias posibles herramientas para el robot que permitiesen la máxima**

limpieza y practicidad posibles. Para diseñar las piezas de las herramientas utilicé **FreeCAD[6]** y para las impresiones 3D, una **RAISE 3D con ABS** gris.

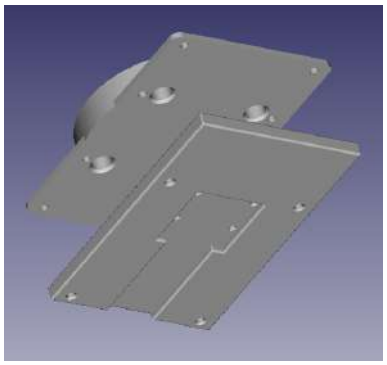
4.1. Primera versión de la herramienta

El principal problema que tenía es que, cuando fallaba el control, **dadas las limitaciones físicas que suponía la longitud del cable de la cámara, muchas veces se desconectaba** y en ocasiones me tocaba reinicar la Raspberry porque no la detectaba tras reconectarla, etc. Esto suponía una gran pérdida de tiempo además de ser intolerable para cualquier aplicación real. Por esto, decidí optar por un **soporte que uniese la Raspberry y la cámara haciendo que ambos dispositivos se muevan con la tercera muñeca** del robot.

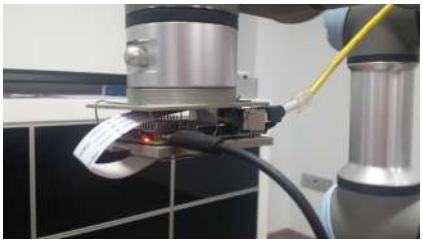
Varios de mis diseños establecían el **riesgo de que la Raspberry se chocase con alguno de los últimos eslabones** con relativa facilidad ante movimientos inesperados del robot. Sin embargo, acabé optando por, a pesar de que el resultado sería más voluminoso, **dejar paralelas tanto la Raspberry como la cámara al muñón** del robot utilizando dos placas, una de las cuales iría enganchada al extremo del robot por encima y a la Raspberry y a la segunda placa por debajo. La cámara se atornilla por la parte inferior de la segunda placa.



(a) Vista superior ideal



(b) Vista inferior ideal



(c) Vista superior real

(d) Vista inferior real

Figura 12: Diseño de la primera versión del soporte/herramienta

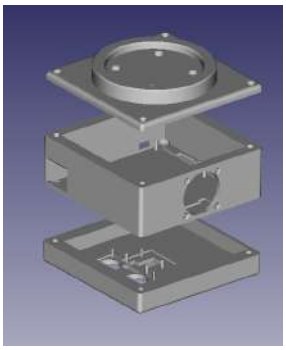
En este soporte **el principal problema que tuve fue encontrar los tornillos**

y pilaretes necesarios como para que quedase bien anclado todo, ya que debían ser relativamente largos dada la magnitud de los agujeros, que estaba limitada por los anclajes de la Raspberry y de la cámara. Así mismo, **el *hardware* quedaba bastante desprotegido y no es estético ver todos los cables y componentes** al aire.

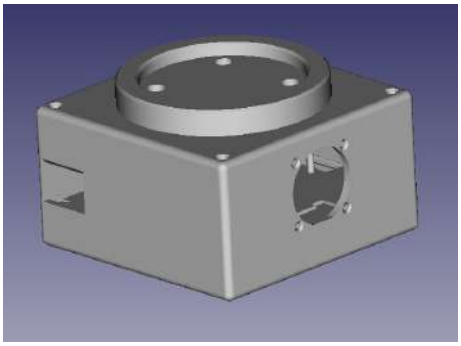
4.2. Segunda versión de la herramienta

En una segunda versión, a pesar de que la anterior era funcional, me enfoqué en hacer un soporte **más estético, además de añadir ciertas mejoras, como la presencia de un hueco para un pequeño ventilador que refrigere el sistema y otro para añadir el sensor HC-SR04 para medir distancias**. Además, todo quedaba **completamente cubierto**, pasando a **mejorar los apartados estéticos y de protección** respecto a la versión anterior.

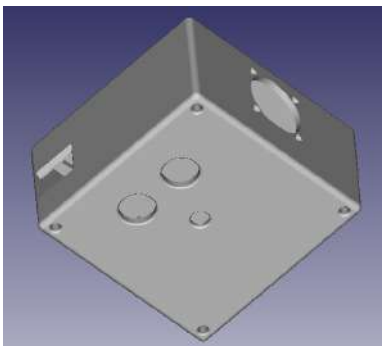
Esta segunda versión **sólo deja al descubierto** las partes esenciales, tales como los **agujeros para anclar la herramienta** al robot, los agujeros para los **tornillos que unen** cada una de las tres **placas** y los agujeros para poder acceder al **puerto ethernet, al microUSB de alimentación, al objetivo de la cámara y al emisor y el receptor del sensor** de distancia.



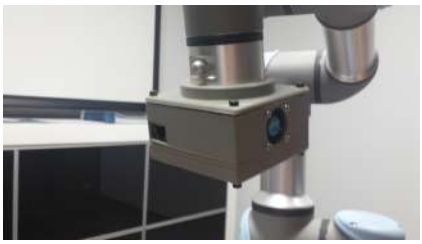
(a) Desglose de placas



(b) Vista superior ideal



(c) Vista inferior ideal



(d) Vista superior real

(e) Vista inferior real

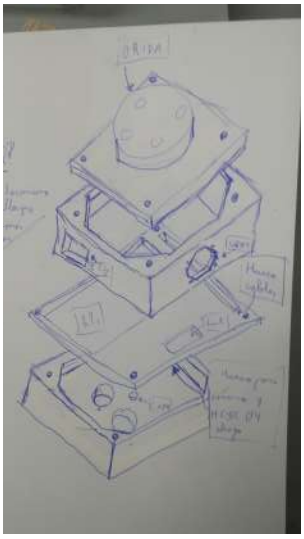
Figura 13: Diseño de la segunda versión (y definitiva) del soporte/herramienta

La placa inferior tiene particular complejidad debido a los surcos que permiten colocar la cámara y el sensor de forma que queden bien sujetos aprovechando las

características físicas de los mismos. Todo esto se aprecia en más detalle en las siguientes imágenes:

Figura 14: Vistas detallada de la placa inferior

A continuación muestro **parte del proceso de diseño** de esta herramienta, como el primer boceto a mano alzada sobre el papel y algunos recortes que realicé para pulir detalles y comprobar que las medidas eran las adecuadas:



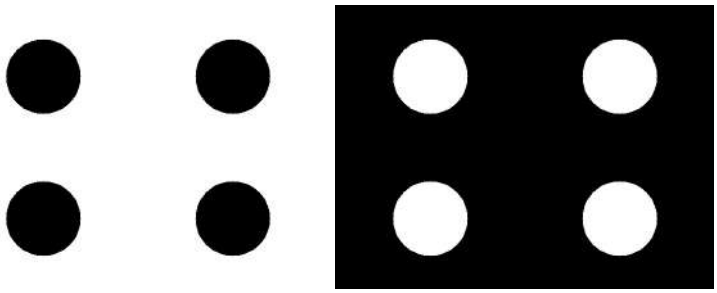
(a) Boceto inicial

Figura 15: Boceto y construcciones en papel de la segunda versión

5. Otras opciones para el identificador

Enfocando el proyecto desde la opción del control visual, todas las opciones para detectar la herramienta que mejor resultado me dieron (midiendo la calidad según la velocidad y robustez con las que obtenía los resultados) se basaban en **segmentación** y, las finales, también en detección por formas (más robustez a cambios de iluminación, reflejos y ruido aunque con más carga computacional). Hasta que llegué al triángulo verde, realicé pruebas con varios patrones:

- **Cuatro puntos negros sobre fondo blanco** (y puntos blancos sobre fondo negro). Logrando un alto contraste que favorecía la **sencillez de la detección y** permite una **segmentación en intensidad RGB**, disminuyendo bastante la carga computacional pero arriesgando la robustez, ya que pueden detectarse **falsos positivos** con gran facilidad en entornos no controlados



(a) Puntos negros en fondo blanco (b) Puntos blancos en fondo negro

Figura 16: Identificador con cuatro puntos de alto contraste

- **Cuatro puntos verdes**. De este modo, a pesar de exigir **segmentación multicanal en HSV** para lograr una buena robustez, se logra una **detección mucho mejor** que en el caso anterior, ya que en un entorno industrial, normalmente, no habrán objetos de estos colores y, si los hubiese, bastaría con modificar los umbrales e imprimir un identificador de otro color

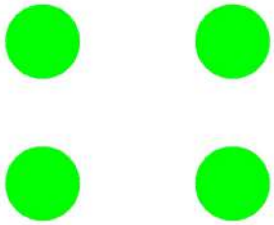


Figura 17: Identificador con cuatro puntos verdes

- **Cuatro puntos de un color distinto cada uno**. De este modo consigo **codificar** todos los posibles estados de **rotación en Z** del identificador. También permite implementar métodos para **incrementar muchísimo la robustez (a costa de consumo de recursos computacionales y tiempo)**. Por contra, supone cuatro segmentaciones y, por tanto, requiere de un **procesado de imagen mucho más pesado**, incluso optimizándolo todo lo posible

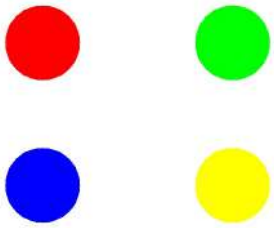


Figura 18: Identificador con cuatro puntos de cuatro colores

- **Un único punto amarillo**. Así se **reduce la aplicación a planos o superfícies con pocas variaciones**, aunque también implica una gran **disminución de la carga** computacional exigida
- **Un triángulo verde**. Tras realizar algunas pruebas, concluí que **la detección de verde era más robusta** a través del mismo procedimiento que la detección de amarillo. El utilizar un triángulo en lugar de un círculo vino dado por confusiones puntuales durante el seguimiento, en el que reflejos u otros elementos ocasionalmente hacían que el objetivo no estuviese en el identificador. Dada esta situación, decidí **endurecer la localización del identificador utilizando el algoritmo de Ramer-Douglas-Peucker**. Este algoritmo simplifica un contorno componiéndolo con varios segmentos lineales, lo que permite contar cuántos puntos (vértices) formas ese contorno, permitiendo **reconocer formas**. Tras varias pruebas con distintas formas (triángulos, cuadrados, rectángulos, pentágonos y círculos) y varias iluminaciones y puntos de vista, concluí que **el triángulo era la que aportaba más robustez**

6. Otras opciones para el procesado de imagen

Para procesar la imagen, tal y como mencioné anteriormente, me baso en una **segmentación** que dejen sólo las zonas verdes en blanco y lo demás negro. De esta forma, de esa nueva imagen, puedo **buscar contornos, discriminarlos por forma y tamaño** y, finalmente **calcular el centroide del triángulo verde más grande**, obteniendo el centro aproximado del identificador.

En ocasiones hay algo de ruido, ya que por reflejos o simplemente porque hay colores cercanos al buscado, aparecen pequeñas zonas blancas no deseadas. En el caso del **identificador de 4 puntos con 4 colores**, al haber tantos colores, había más ruido, por lo que me veía obligado a aplicar una **erosión para suprimir ese ruido**, evitando detecciones erróneas pero también aumentando más la carga computacional. **En**

el caso del identificador que escogí como definitivo (un único triángulo verde), este problema no tenía importancia porque las discriminaciones acababan con el ruido y la aproximación del algoritmo de clasificación de formas elimina irregularidades en los bordes.

Es importante que, con tal de incrementar muy notablemente la robustez, la umbralización la hago en el espacio de color HSV en lugar de en el RGB que percibe la cámara. A pesar de que realizar esta conversión previa es la tarea que más tiempo consume en todo el proceso, **segmentar en RGB daba unos resultados completamente inaceptables**. Esto es por la forma de codificar la información, ya que, a diferencia de RGB, **HSV separa la información de intensidad de luz de la del tono del color**, siendo muchísimo más robusto a cambios de iluminación. A pesar de que esto también se estructura así en otros espacios de color, HSV es ampliamente utilizado en la visión por computador, así que en la mayoría de librerías para este ámbito, **las transformaciones entre RGB y HSV están muy pulidas**. Por otra parte, en HSV los colores se representan en un cilindro, no en un cubo, como en RGB, lo que hace bastante más sencillo establecer los márgenes inferior y superior de los umbrales.

7. Otras opciones para las comunicaciones

En un principio utilizaba comunicación **TCP** tanto para **leer la posición de la herramienta** respecto a la base como para ordenar los movimientos (mi programa aplicaba las correcciones a la posición de cada instante y ordenaba los desplazamientos o velocidades deseados). Sin embargo, luego me di cuenta de que esto era **eficientemente mejorable** enviando únicamente las correcciones al robot y que él se encargue de aplicarlas a su posición y de moverse, haciendo que el UR se cargue con esas operaciones mientras la Raspberry sigue con la siguiente captura de imagen y su procesado. Para esto escribía **cuatro registros a través de Modbus TCP**, dos con las **correcciones** cartesianas (en milímetros, ya que sólo se pueden almacenar enteros) a hacer por el robot y otros dos con los **signos de dichas correcciones** (0 si era en el sentido negativo del eje y 1 si era en el positivo), ya que sólo se permiten números positivos en los registros. **De este modo, pasé de tener que leer 17 paquetes TCP** sólo para saber la herramienta (lo leía desde el programa del robot con get_actual_tcp_pose() directamente) y también, aunque tenga menos repercusión, **me ahorraba la creación, e inicialización de varias variables, acceder a sus posiciones de memoria, sumarle el contenido de las correcciones, hacer *casts* a *strings* para enviar el comando, etc.**

Es importante puntualizar que **esto no sirve para el programa que usa control visual clásico**, ya que ese obtenía las velocidades articulares, así que lo más óptimo que se me ocurrió fue pasar directamente el comando, ya que si no tendría que escribir 12 registros (6 velocidades y 6 signos) y desde el programa del robot leerlos todos, hacer

comparaciones, modificar variables, etc. Además, tal y como implementé **ese programa, no requiere la intervención de ningún programa del robot, bastaba con conectar la Raspberry al armario pero funcionaba peor que el definitivo.**

8. Análisis de resultados y conclusiones

Con tal de analizar los resultados que he obtenido (y discriminar o tomar algunas mejoras), **medí los tiempos para comprobar cómo de útil sería modificar ciertas partes** de mi algoritmo. Para obtener medidas lo más reales posibles, tomé **cinco iteraciones distintas** del bucle principal del algoritmo de programación, todas ellas **no-consecutivas** y analicé teniendo en cuenta la media y la disparidad de estos resultados. Las medidas que obtuve fueron:

Proceso	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Media
Caputra de imagen	0.00433	0.00439	0.00493	0.00459	0.00461	**0.00457**
Conversión RGB-HSV	0.01514	0.01584	0.01526	0.01539	0.01482	**0.01529**
Segmentación	0.00923	0.02191	0.00941	0.00917	0.00871	**0.011686**
Búsqueda contornos	0.00397	0.004	0.00413	0.00418	0.00364	**0.003984**
Búsqueda área máxima	0.0002	0.00017	0.0001	0.00011	0.00013	**0.000142**
Cálculo centroide	0.00018	0.00018	0.0001	0.00014	0.00014	**0.000148**
Media móvil	0.00008	0.00005	0.0001	0.00006	0.00007	**0.000072**
Cálculo de correcciones	0.00004	0.00003	0.00003	0.00005	0.00005	**0.00004**
Escritura Modbus	0.00540	0.00704	0.00637	0.0099	0.00674	**0.00709**

Cuadro 1: Mediciones de tiempos de proceso (en segundos)

Salvo en algún caso puntual, como en la segmentación, que en la segunda medida da un tiempo de 20ms cuando evidentemente la tendencia ronda los 10 ms, en todos los casos se observa una tendencia bastante clara en cuanto a consumo temporal, por lo que puedo confirmar que **fueron unas pruebas válidas**.

De estas medidas puedo extraer dos lecturas importantes:

- **El tiempo aproximado de una iteración** completa del bucle, que resultará de sumar todoas las medias, es decir: (0.04583 + 0.0502 + 0.04718 + 0.04962 + 0.04965)/5 = **0.048496 s**. Redondeando, estimo que en cada iteración se invierten 50 milisegundos, por lo que al robot se le envían correcciones con esa cadencia, siendo aproximadamente **5 veces la máxima velocidad** a la que pueden enviársele órdenes (8 ms)

- **El procesado visual es lo que más tiempo consume**. Con un total de 0.031244

(aproximadamente 30 ms), en el procesado visual se invierte sobre un **70 % del total** del tiempo invertido en cada bucle, por lo que éste **debe de ser el principal objetivo en las optimizaciones futuras que vayan a hacerse**

Tras el trabajo realizado así como los resultados obtenidos, **concluyo** que creo que si se siguiese desarrollando esta aplicación, podría llegarse a algo realmente práctico y útil en el mundo industrial. Tal y como mencioné en la introducción, considero que con un sistema de este tipo se logra mejorar sustancialmente la comodidad y facilidad de programación, ya que el usuario tan sólo tendrá que realizar la tarea con normalidad, con la única diferencia de colocar una pegatina en su herramienta o, en el peor de los casos, añadirle una pequeña placa donde pueda colocársele dicho identificador.

Por otro lado, **a título personal**, estas prácticas me han servido para desarrollarme tanto desde un punto de vista personal como desde un **enfoque técnico**. En este último aspecto he adquirido experiencia con los robots de Universal Robots así como mucha más soltura con Python. También he aprendido bastante sobre redes y comunicaciones de una forma más transversal. Considero que he mejorado mi capacidad de análisis de resultados y de organización, tanto de mi tiempo como en el desarrollo de *software* y de piezas 3D. **En lo personal**, he conocido puntos de vista de gente con experiencia laboral fuera del ámbito académico y he acumulado una gran cantidad de vivencias que me han motivado aún más a seguir formándome e incluso me han inspirado para volver a sentirme atraido por campos por los que había disminuido mi interés.

9. Trabajos futuros

Tal y como mencioné en la introducción, este proyecto no tenía el objetivo de lograr una implementación totalmente refinada lista para lanzarse al mercado, sino iniciar una posible vía hacia dicha funcionalidad. Por esto, me pareció de gran relevancia mencionar algunos de los trabajos futuros que podrían realizarse para avanzar a partir del punto al que he llegado.

9.1. Optimización

Una de las maneras más inmediatas de lograr acelerar el procesado desde la Raspberry es **deshabilitar servicios inútiles** que estén ejecutándose en la misma. Este dispositivo es un ordenador y, como tal, la ejecución del programa de esta aplicación no es su prioridad y está ejecutando una gran cantidad de subprocesos que limitan el tiempo durante el que mi programa tiene disponible los recursos de computación.

Otro avance en cierta medida relacionado con el anterior sería **implementarlo todo en una FPGA para, posteriormente, crear un ASIC** que realice todo el proceso o, al menos, el procesado de la imagen (que es lo más pesado) de la forma más rápida

posible. En este tema hay que tener varios aspectos en cuenta. El principal es la cámara, ya que su velocidad de captura y la forma que tenga de enviar los datos que capte casi con total seguirdad pasarán a ser el cuello de botella de la aplicación. Podría reducirse el impacto de esto de una forma tan sencilla como utilizando la cámara del mercado que mejor se adapte a las necesidades como de una tan compleja como diseñando una cámara con una interfaz de comunicación especial para esta aplicación.

Por último, hay ciertas **partes del código que pueden ser paralelizadas**, tanto del procesado de la imagen (como la umbralización) como de algunas otras operaciones, en las que habría que evaluar hasta qué punto es práctico tener que abrir y cerrar un hilo de ejecución únicamente para realizarlas, aunque puede que sea útil mantenerlo activo hasta que se acabe el programa haciendo que los hilos sean bucles infinitos y sincronizándolos con el programa con *mutex* o cualquier otra técnica que se ajuste lo mejor posible a nuestros requerimientos.

Dada la menor exigencia de recursos durante la reproducción de movimientos, podría **instalarse otro controlador menos potente** que el principal que se encargue de llevar a cabo esta tarea, permitiendo así reducir el consumo, aunque seguramente sea una reducción despreciable dado el elevado consumo que implicará el uso del robot.

9.2. Mejoras de funcionalidades

A pesar de que ya de por sí la funcionalidad de mejorar la programación por guiado tradicional puede ser suficiente, esta puede verse afectada por algunos factores que podrían empeorar o, en casos extremos, llegar a inutilizarla. Por esto y, dado que se usa visión artificial, crear una **interfaz que permita calibrar los umbrales y algunas partes del proceso** de forma sencilla puede ser una mejora muy práctica, así como bastante sencilla de implementar.

Por otra parte, a costa de aumentar ligeramente la voluminosidad del soporte, **incluir una batería** o hacer que **reciba energía del conector de herramientas del robot** sería muy práctico, ya que supondría poder prescindir de uno de los dos cables que requiere la herramienta. Para esta última opción, podría transformarse una de las salidas a 12 voltios del conector a los 5 que requiere la Raspberry.

Siguiendo con la intención de reducir el cableado, si se desarrollase una **interfaz wifi para el UR** podría lograrse una aún mayor simpleza en este aspecto, logrando eliminar todos los cables que se exigen para la aplicación. El principal problema de esto reside en la seguridad, que se ve vulnerada por las comuniciones inalámbricas. Por esto, otra buena opción podría se desarrollar una **comunicación a través del conector de herramienta o investigar otras opciones** inalámbricas más seguras que el Wi-Fi convencional. Una opción muy factible y más aún si se alimenta a la Raspberry desde el conector de herramienta sería utilizar las comunicaciones que incorporan los nuevos robots e-series para realizar la escritura de los registros Modbus, pudiendo eliminar todos los cables de la herramienta.

9.3. Otras mejoras

Dada la simplificación que realicé pasando del control visual clásico a uno mucho más simple, este método sólo puede utilizarse para programar actividades que se realicen sobre superficies más o menos planas, ya que el robot desplazará su extremo en el plano donde se encontrase inicialmente. Una opción para mejorar esto podrìa ser **combinar la visión artificial con odometría más *convencional***, como podría ser un giroscopio (a ser posible, combinado con un acelerómetro para poder hacer uso de un filtro que combine ambas medidas, estabilizando mucho más las lecturas).

También podría realizarse un **control más avanzado**, aunque seguramente suponga un aumento del coste computacional. Quizás incluir realimentaciones o partes integral o derivativa mejoren el comportamiento. En favor de la robustez, considero que se debería priorizar la estabiliidad. Dado que el robot ya corrige muy bien la forma de alcanzar las posiciones, haciendo el control *de verdad* por sí mismo, comenzaría centrándome en estabilizar las *conclusiones* a las que el algorimto lleva, es decir, lograr hacer que las correcciones que se envían por Modbus sean todo lo estables posibles permitiendo un *tracking* lo más veloz posible.

Otra posible mejora sería el **desarrollo de un URCap** que permita que el programa del robot sea más estético y pueda ser más sencillo utilizarlo para otras aplicaciones más complejas en las que pueda ser útil este desarrollo.

Por último, para una aplicación industrial real, la **herramienta** que he diseñado no es válida, ya que no garantiza protecciones demasiado relevantes, por lo que habría que contemplar el uso de otros materiales, opciones para sellarla, quizás incluir LEDs de estado que indiquen el correcto funcionamiento de cada componente (ventilación, sensores, comunicaciones, etc.).

Referencias

[1] Especificaciones del módulo de cámara: https://elinux.org/Rpi_Camera_Module#Technical_Parameters_.28v.2_board.29

[2] Lectura del sensor de distancia HC-SR04: https://robologs.net/2015/07/31/tutorial-de-raspberry-pi-y-hc-sr04/

[3] Documentación de OpenCV: https://opencv.org/releases.html

[4] Documentación de pymodbus (para clientes síncronos): https://pythonhosted.org/pymodbus/library/sync-client.html

[5] Herramienta *on-line* para diagramas: https://www.lucidchart.com/

[6] Herramienta para diseño 3D: https://www.freecadweb.org/

---

[7] Tutoriales oficiales de Universal Robots: https://www.universal-robots.com/how-tos-and-faqs/how-to/

[8] Zacobria (foro de consulta sobre robots UR): http://www.zacobria.com/

[9] Robotiq (foro de consulta de robótica): https://robotiq.com/

---

[10] Lectura de la IMU MPU6050 (la utilicé para algunas pruebas): http://blog.bitify.co.uk/2013/11/reading-data-from-mpu-6050-on-raspberry.html