UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE MASTER EN INGENIERÍA INDUSTRIAL

DISEÑO, DESARROLLO Y EVALUACIÓN DE UN SISTEMA AUTOMÁTICO DE CLASIFICACIÓN DE HELMINTOS BASADO EN MICROSCOPÍA

AUTOR: VÍCTOR RAMOS VICEDO

TUTOR: ANTONIO JOSÉ SÁNCHEZ SALMERÓN

Curso Académico: 2015-16

Agradecimientos

No concibo una manera más acertada de comenzar este documento que agradeciendo su implicación a varias personas sin las cuales no habría sido posible realizar este proyecto.

En primer lugar a mi profesor y tutor el Dr. Antonio José Sánchez Salmerón, por haberme introducido en el interesante mundo de la Visión Artificial y haberme guiado en el desarrollo del presente trabajo.

En segundo lugar a Nacho Pérez Muñoz por la ayuda prestada con la programación y los ensayos que aparecen en esta memoria.

Por último a mis padres, cuyo apoyo incondicional me ha permitido llegar hasta aquí y seguir avanzando en mi carrera.

Resumen

El crecimiento de la población mundial y el aumento en las actividades industriales desde hace unos años ha repercutido negativamente en la calidad del agua de consumo. En el caso de los vertidos domésticos, la carga contaminante está especialmente constituida por materia orgánica y microorganismos de origen fecal.

Actualmente miles de millones de personas en todo el mundo están infestados por parásitos helmintos, por lo que el control de la población de estos microorganismos y de sus huevos es una tarea de suma importancia para garantizar la salubridad del agua de consumo.

Es por esto que existen numerosas instituciones cuya actividad tiene como objetivo el estudio de la calidad de las aguas potables, con intención de controlar la existencia de parásitos en ellas. Hasta ahora el método de estudio consistía en la concentración de una muestra de agua para su posterior examen en un microscopio óptico por parte de un técnico de laboratorio buscando huevos de estos microorganismos. Como es obvio, esto implica mucha carga de tiempo y su éxito depende de la pericia y la atención del técnico en cada momento.

En base a esta situación, se ha decidido desarrollar un sistema basado en un microscopio óptico con platina motorizada, una cámara de alta resolución y un ordenador que se encargue de analizar una muestra de agua concentrada en busca de huevos de helmintos de distintas clases, clasificándolos y almacenando tomas de imagen de dichas detecciones con el objetivo de que el técnico de laboratorio solamente tenga que revisar la lista de detecciones y validar los resultados, con el consiguiente ahorro de tiempo y errores.

Esta memoria describe la parte de dicho trabajo relativa a la toma y procesamiento de imágenes, incluyendo el algoritmo de autoenfoque, generación de ruta de tomas, preprocesamiento, análisis y clasificación de las mismas.

Palabras Clave: Helmintos, Detección, Clasificación, Microscopía, Histograma de Gradientes, HOG.

Documentos incluidos en el TFM

- *Memoria*
- *Presupuesto*

Índice de la Memoria

Índice del Presupuesto

Memoria

1 Introducción teórica

1.1 Historia de los helmintos

El término "Helmintos" no hace referencia exactamente a una especie de microorganismos, sino que es un nombre general no taxonómico que se utiliza para designar a los gusanos parásitos y a los de vida libre[1]. Existe una rama de la parasitología que se encarga de estudiar estos parásitos, denominada helmintología.

Ya en el antiguo Egipto (≈1500 A.C.) se detectaron las primeras afecciones de estos gusanos en humanos, concretamente causadas por tenias, y ya se empezaron a postular tratamientos contra los síntomas. Los hebreos empezaron a dictar leyes sanitarias con el objetivo de proteger a su pueblo de enfermedades causadas por plagas de insectos y carnes infestadas con "piedras", que no eran más que larvas de varias especies de cestodos del género *Taenia,* causando enfermedades conocidas como *Cysticercus cellulosae*.

También Aristóteles (384-322 A.C.) empezó a realizar descripciones y clasificaciones sobre estos parásitos intestinales, pero no fue hasta el siglo XIX que se empezó a invertir en el estudio de las parasitosis tropicales, dando a luz a una nueva rama de la ciencia que se encargaría del estudio de estos microorganismos: la parasitología.

Ya por el 1916, se consiguió determinar por completo el ciclo vital de los helmintos causantes de la infección helmíntica más común a nivel mundial, los Áscaris (Stewart, 1916), y en 1922, Koino ingirió intencionadamente 2000 huevos de estos helmintos con objetivo de comprobar experimentalmente este ciclo, eliminando días después 667 gusanos por las deposiciones mediante un antihelmíntico (Koino, 1922).

1.2 Clases de helmintos

Como se ha explicado en el apartado precedente, el término "helminto" hace referencia a un grupo de gusanos tanto parasitarios como de vida libre. Estos gusanos se organizan principalmente en cuatro clases: Platelmintos, Nematodos, Acantocéfalos y Anélidos.

[1] Se conoce por gusanos de vida libre a aquellos que no necesitan un huésped para poder vivir y desarrollarse como los parásitos

1.2.1 Platelmintos

Los platelmintos son un filo de animales invertebrados que incluye unas 20.000 especies. La mayor parte son hermafroditas y habitan en ambientes marinos, terrestres húmedos y aéreos. La mayor parte necesitan varios huéspedes a lo largo de su vida, unos para el estado larva y otros para el estado adulto. Son los animales más simples con interneuronas.

Estos gusanos son los carnívoros triblásticos más simples que existen. En cuanto a su morfología tienen una estructura aplanada dorso-ventral similar a una cinta. Carecen de aparato circulatorio, por lo que el tubo digestivo realiza las funciones digestivas y de distribución de nutrientes. Puesto que tampoco disponen de aparato respiratorio, el oxígeno necesario para el metabolismo pasa a través de los tegumentos[2] de su cuerpo.

Tampoco disponen de aparato locomotor, por lo que se desplazan mediante vibraciones de su epitelio. Disponen de un sistema nervioso sencillo bilateral que recorre el cuerpo y un rudimentario aparato locomotor.

A esta clase pertenecen los Trematodos y los Cestodos.

1.2.1.1 Trematodos

Los trematodos (conocidos comúnmente por “duelas”) son una subclase de los platelmintos que tienen como característica distintiva la disposición de una serie de ventosas y ganchos de fijación para adherirse al huésped. Su tamaño varía entre uno y varios centímetros de longitud y tienen una morfología semejante a una hoja.

Figura 1: Larva de Schistosoma (Trematodo). Fuente: Wikipedia

[2] Los tegumentos engloban la piel y las estructuras complementarias externas como glándulas mucosas o escamas

Dentro de los trematodos se distinguen los aspidogastros, que parasitan moluscos y vertebrados marinos, y los digeneos, que parasitan órganos o tejidos humanos, por lo que son los que tienen importancia sanitaria. Entre estos últimos se pueden destacar los del género Schistosoma, que se transmiten por agua contaminada y parasitan las venas del cuerpo humano; y los del género Fasciola, que se transmiten a través de vegetales y son parásitos del hígado.

1.2.1.2 Cestodos

Los cestodos son la otra clase principal de platelmintos junto a los trematodos, que agrupan unas 4.000 especies, todas ellas parásitas. Carecen de aparato circulatorio y digestivo, dependiendo completamente de la absorción de nutrientes del hospedador a través de la piel para sobrevivir.

A diferencia de los trematodos, los cestodos están profundamente modificados para parasitar al hospedador, por lo que tienen una gran variabilidad para adaptarse del mejor modo al mismo. Su cuerpo está formado por una serie de segmentos conocidos como proglótidos formando una especie de cinta y pueden llegar a medir más de 5 metros.

A este grupo pertenece la Taenia solium (conocida coloquialmente como solitaria) y las especies del género Hymenolepis

Figura 2: Tenia Solium (Cestodo). Fuente: Diario ABC

1.2.2 Nematodos

Los Nematodos son, al igual que los Platelmintos, un filo de gusanos invertebrados con más de 25.000 especies registradas, por lo que constituyen el 4º filo más grande del reino animal por número de especies. Son conocidos como gusanos redondos gracias a su morfología y aunque pueden proliferar en entornos terrestres, son organismos principalmente acuáticos.

Esta clase incluye especies tanto parásitas como de vida libre. Miden desde menos de 1 mm hasta más de medio metro, siendo la *Placentonema gigantissima* la especie de nematodo más grande, alcanzando hasta los 8 metros de largo. Tienen un cuerpo redondo y alargado,

sin segmentos, y en algunas especies disponen de dientes o ganchillos en el extremo anterior para adherirse a algunos tejidos.

A diferencia de los Platelmintos, los Nematodos disponen de aparato digestivo, y se alimentan por aspiración de sangre o absorción de tejidos destruidos, contenido intestinal o líquidos corporales.

Entre los Nematodos se pueden distinguir los del género Áscaris, conocidos como “lombriz intestinal”, los del género Trichuris, conocidos como “gusano látigo” y los del género Anisakis, conocidos por infestar frecuentemente consumidores de productos de la cocina japonesa, donde es común comer el pescado crudo que contiene a menudo estos parásitos.

Figura 3: Huevo de Trichuris (Nematodo). Fuente: Wikipedia

1.2.3 Acantocéfalos

Esta cuarta clase de helmintos constituye un filo de gusanos parásitos que tienen una probóscide con espinas en su cabeza que les permite fijarse a la mucosa del hospedador. Se han descrito unas 1.100 especies, con un tamaño que varía entre unos pocos milímetros y 65 centímetros en la especie más grande, la *Gigantorhynchus Gigas*.

Los acantocéfalos son similares a algunos cestodos en el hecho de que carecen de boca por donde ingerir alimento, por lo que absorben los nutrientes que necesitan del hospedador, que les realiza la tarea de la digestión.

Son un grupo de helmintos con ciclos vitales muy complejos, por lo que de 1.100 especies solo se ha conseguido caracterizar completamente los de 25 de ellas.

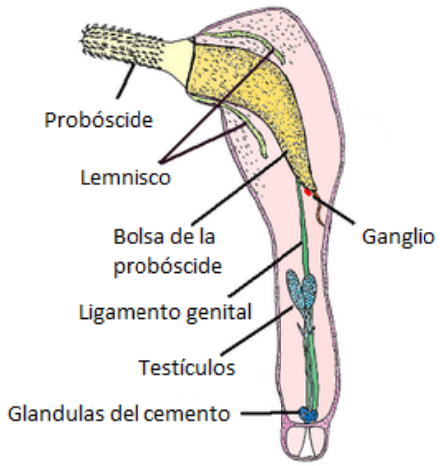

Figura 4: Estructura de un Acantocéfalo. Fuente: Wikipedia

1.2.4 Anélidos

Los anélidos constituyen la cuarta y última clase de helmintos, y se componen por más de 16.700 especies que incluyen algunas muy conocidas como son las lombrices de tierra o las sanguijuelas. La mayoría se encuentran en ambientes húmedos, aunque también se conocen especies terrestres.

En cuanto a su morfología cabe destacar que están formados por una serie de anillos llamados metámeros unidos entre sí, y su longitud alcanza desde menos de 1 milímetro hasta varios metros. Tienen una cavidad llena de fluido llamada celoma en el que están suspendidos los órganos internos.

Figura 5: Hesiocaeca methanicola (Anélido). Fuente: Wikipedia

1.3 Estado del arte

Existen multitud de estudios y artículos científicos relativos a la detección automática de parásitos intestinales humanos por microscopía óptica con excelentes porcentajes de acierto. El abanico de técnicas disponibles para esta aplicación es increíblemente extenso, por lo que los métodos utilizados en dichos estudios suelen ser siempre distintos, pese a mantener a menudo las operaciones comunes de pre procesamiento.

César Beltrán Castañón ya experimenta en su tesis doctoral con parásitos de gallina del género Eimeria basándose en características geométricas, de textura y de curvatura. En este trabajo, Beltrán utiliza por un lado un clasificador Bayesiano obteniendo un porcentaje de acierto del 80% y por otro un clasificador estadístico obteniendo un 85%. (Castañon, 2007).

En la siguiente imagen se muestran los pasos en la detección de huevos de este trabajo, en el que se opta por un procesamiento con binarización y detección de bordes. Nótese la excelente calidad de la imagen tomada, en la que prácticamente no hay elementos de fondo y los objetos a detectar están claramente definidos. El elevado porcentaje de acierto de este trabajo surge principalmente de esta buena calidad de imagen y definición de objetos.

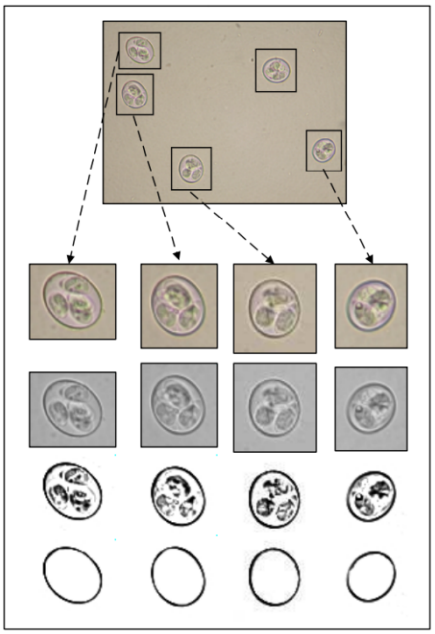

Figura 6: Proceso de detección de huevos. Fuente:(Castañon, 2007)

Yoon Seok et al realizan una identificación en 3 etapas de muestras fecales, aplicando una binarización con umbral estático para conseguir eliminar ruido de la imagen y segmentar los objetos de interés, para después utilizar la transformada de Fourier sobre el contorno de cada uno y terminar con un clasificador basado en 2 redes neuronales, en la que la primera se encarga de detectar exclusivamente artefactos, sirviendo de base para la segunda, que clasifica los tipos de parásitos con un porcentaje de acierto del 83 % (H. C. K. M.-H. C. Yoon Seok Yang, 2001).

En 2009, *Avci y Varol* propusieron un sistema basado en momentos invariantes y support vector machine de multi clase para clasificar huevos de parásitos intestinales en tres fases: Primero una fase de pre-procesamiento donde se reduce el ruido mediante un filtro de mediana, se aumenta el contraste, se utilizan operaciones morfológicas para detectar el objeto de interés y se aplica un umbral a la imagen con objetivo de descartar objetos cuyo tamaño sea demasiado grande como para poder ser un candidato. Posteriormente se realiza la extracción de características, donde se obtienen los momentos invariantes de los objetos de interés basados en la distribución de intensidad, y por último se obtiene la clasificación mediante una support vector machine (SVM) que clasifica los huevos según la especie. Con este sistema obtuvieron un porcentaje de acierto del 97.70% (Avci & Varol, 2009).

Son estos artículos los que constituyen la garantía de las posibilidades de éxito que los sistemas de visión artificial tienen en el campo de la clasificación de microorganismos.

2 Desarrollo del trabajo

2.1 Planteamiento

Como se ha podido apreciar en las referencias citadas en el apartado anterior, todos los estudios realizados relativos a la detección de estos parásitos no se basan en los gusanos, sino en sus huevos. Esto es porque la fase huevo constituye la etapa contagiosa de dichos parásitos, y es muy resistente a las condiciones ambientales y a la desinfección en las estaciones de tratamiento de aguas residuales.

Por tanto, el presente proyecto va a basarse de igual manera en la detección de los huevos de estos helmintos. Concretamente, las especies requeridas por la empresa beneficiaria de la solución a detectar son las incluidas en los géneros siguientes:

- Género Áscaris
- Género Hymenolepis
- Género Trichuris
- Género Schistosoma
- Género Taenia

La detección va a basarse en un microscopio óptico de la casa OPTIKA® con dos ópticas intercambiables, de x40 y x200 aumentos, por lo que el proceso de detección seguirá la metodología de tomar imágenes a x40 para detectar potenciales huevos, y en el lugar donde detecte un posible candidato, se cambiará a la óptica de x200 para obtener una toma más detallada y proceder a la clasificación del mismo bien en una de las 5 clases o bien en un artefacto[3].

[3] Se denomina artefacto a un objeto que inicialmente se ha catalogado como objeto de interés por su similitud con ellos pero después de un procesamiento más exhaustivo se ha descartado.

Figura 7: Microscopio utilizado. Fuente: OPTIKA

También se dispondrá de una cámara USB de alta resolución (2048x1536 píxeles) de la casa MOTIC® para la toma de imágenes y su procesamiento a través de un ordenador.

Figura 8: Cámara USB utilizada. Fuente: MOTIC

En cuanto al software utilizado será un programa elaborado por otro componente del equipo de trabajo dedicado exclusivamente a esta aplicación. Dicho programa estará programado en lenguaje JAVA para poder desarrollar la interfaz al nivel requerido por la empresa beneficiaria, y utilizará librerías de tratamiento de imágenes como ImageJ. No obstante, en el presente proyecto no se detallará el funcionamiento de dicha interfaz.

Puesto que el trabajo realizado ha sido principalmente de programación de código para conseguir los objetivos propuestos, la presente memoria se basará en la explicación de dichos fragmentos de código para solucionar los problemas planteados. Para su realización se ha utilizado tanto MATLAB como JAVA. La primera opción ha sido necesaria por la facilidad de MATLAB para purgar código y ver los resultados sin necesidad de estar conectado al microscopio. La opción de JAVA se ha utilizado en los algoritmos dedicados al autoenfoque de la cámara y a la generación de la trayectoria de la toma de imágenes sobre el portaobjetos que contiene la muestra.

Este apartado de la memoria se ha estructurado del siguiente modo:

1. Primero se describirá el bloque de trabajo relativo a la detección del perímetro del cubreobjetos a estudiar automáticamente y a la generación de la trayectoria que debe seguir el microscopio para abarcar toda el área de dicho cubre.
2. En segundo lugar se describirá el algoritmo utilizado para obtener imágenes enfocadas automáticamente.
3. Finalmente se describirá el bloque de código relativo a la detección e identificación de huevos de helmintos, incluyendo otras metodologías que han sido probadas y finalmente se han descartado.

2.2 Determinación de trayectoria a seguir

2.2.1 Problema a resolver

La primera situación que había que resolver era cómo determinar la trayectoria que debía seguir la toma de imágenes para cubrir completamente la muestra sometida a estudio, ya que bajo ningún concepto podía quedar área de muestra sin fotografiar.

2.2.2 Opciones consideradas

Como primera aproximación se pensó en definir una ruta fija en zigzag que cubriese un área fija del microscopio, donde el operario colocara perfectamente alineado el portaobjetos con respecto a un punto de referencia definido.

No obstante se desestimó rápidamente esta opción porque la validez de la muestra iba a depender de la pericia del técnico al colocar el cubre. Hay que tener en cuenta que, trabajando con tantos aumentos, hasta la más mínima inclinación puede dejar fuera del recorrido varias decenas de tomas, por lo que se tuvo que pensar en un método de generar la trayectoria en función de la colocación de la muestra, teniendo en cuenta que la inclinación máxima no iba a superar un par de grados.

Esta solución final consistió en un programa en varias etapas que detecta las aristas del borde del cubreobjetos y genera un rectángulo perfectamente alineado que engloba completamente al cubreobjetos. Una vez se tiene este rectángulo solo queda generar la ruta en zigzag que debe seguir el microscopio.

2.2.3 Desarrollo de la solución

Como se ha señalado en el apartado precedente, la elaboración del código relativo a la generación de la trayectoria a seguir por el microscopio para cubrir la totalidad del área del cubre se ha realizado directamente en JAVA, pues era imposible realizar pruebas off-line con MATLAB de dicha funcionalidad.

La representación del código se ha realizado con el software *Sublime Text 3*, un procesador de texto que reconoce multitud de lenguajes de programación y que permite diferenciar por colores las palabras con algún significado especial para el compilador.

El código de la solución final devuelve un vector de dimensión Nx2 con las coordenadas en X e Y correspondientes a cada punto de la ruta a trazar de longitud N[4], y está compuesto por 5 subprogramas o "clases" que se definen a continuación:

1. Subprograma **Ruta()**

```
1  private static int[][] Ruta() {
2      int anchotoma=2048;
3      int altotoma=1536;
4      int tomasx, tomasy, xini, yini, n=0, alto, ancho;
5      int [] coord=new int[4];
6
7      coord=Rectangulo();
8      ancho=coord[2]-coord[0];
9      alto=coord[3]-coord[1];
10     tomasx=(int) ancho/anchotoma;
11     tomasy=(int) alto/altotoma;
12     int [][] ruta;
13     ruta=new int[tomasx*tomasy][2];
14     for (int i=coord[0];i<coord[2];i=i+anchotoma){
15         for (int j=coord[1];j<coord[3];j=j+anchotoma){
16             ruta[n][0]=i;
17             ruta[n][1]=j;
18             n++;
19         }
20         if (i<coord[2]){
21             i=i+anchotoma;
22             for (int j=coord[3];j>coord[3];j=j-anchotoma){
23             ruta[n][0]=i;
24             ruta[n][1]=j;
25                 n++;
26             }
27         }
28     }
29     return ruta;
30 }
```

Figura 9: Clase Ruta(). Fuente: Elaboración propia

[4] Durante todo el proyecto se ha considerado el eje de coordenadas situado en la esquina superior izquierda de la imagen, con el eje X horizontal hacia la derecha y el eje Y vertical hacia abajo.
También hay que tener en cuenta que todas las variables referidas a estas coordenadas serán de tipo entero, ya que los píxeles de una imagen no pueden tomar valores decimales.

Es la clase a la que llamará el programa principal para obtener el vector de dimensión Nx2 mencionado anteriormente. No tiene argumentos de entrada, y se declara con la siguiente instrucción, en la que los atributos “static” y “private” hacen referencia a las cualidades de la clase (aunque su explicación detallada va más allá de los objetivos del presente documento) y el atributo “int [ ] [ ]” señala que la función devolverá una variable de tipo vector de enteros de 2 columnas.

En el primer fragmento de código se declaran las variables con las que va a trabajar la función. En primer lugar se declaran dos variables correspondientes al tamaño (en píxeles) que tiene cada imagen tomada por la cámara. Es decir, la resolución de dicho dispositivo, que es de 2048x1536 como se ha señalado. Seguidamente se declaran el resto de variables necesarias.

En la línea 7 empieza el cuerpo del programa, con una instrucción que llama a la función *Rectangulo()*y guarda la variable que devuelve en *coord,* que como se ha visto en la declaración, es un vector de 4 enteros. Estos 4 números enteros hacen referencia a la posición de las 4 aristas del rectángulo virtual que envuelve al cubre. En la siguiente imagen se puede comprender con facilidad lo explicado.

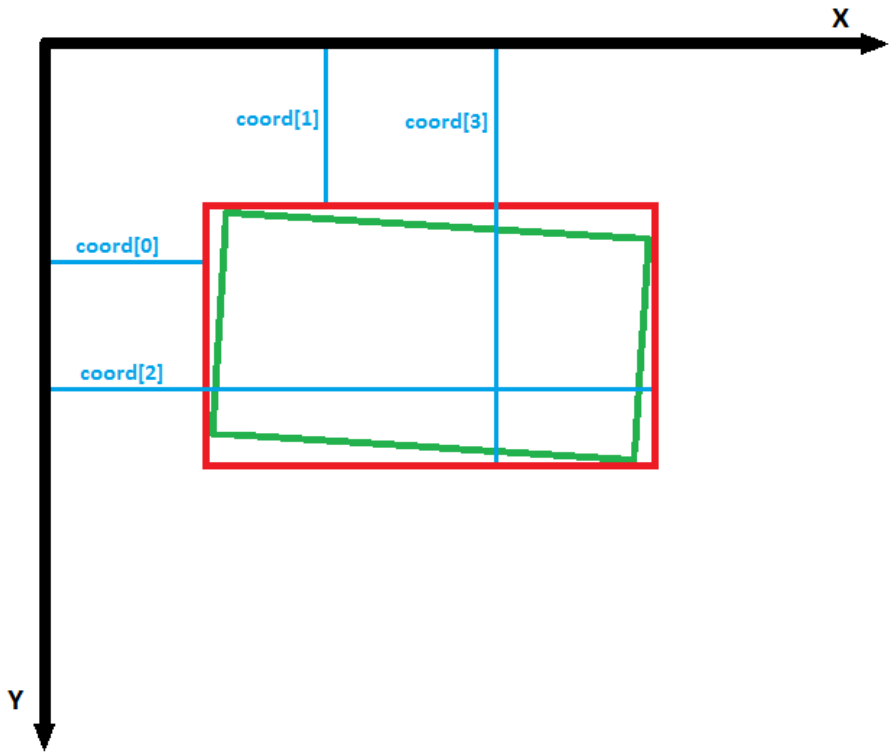

Figura 10: Representación del rectángulo envolvente. Fuente: Elaboración propia

De este modo se puede obtener la anchura y altura del rectángulo verdadero sobre el que hay que generar la trayectoria a seguir por la cámara para no dejar ningún área sin estudiar, que es lo que se realiza en las siguientes 2 líneas. Una vez conocidas la anchura y la altura se dividen entre la anchura y la altura de cada toma de la cámara para conocer cuántas tomas son necesarias en cada eje. Ya que se

realiza una división, es necesario obtener solo el valor entero, pues no puede haber un número de tomas decimales, para lo que se realiza un *casting*, que consiste en guardar sólo la parte entera de la división, no importando la parte decimal pues hay un margen de error admisible implementado en la función *Rectangulo()* que se verá más adelante.

Ahora se obtiene el número de tomas totales necesarias multiplicando las tomas a lo alto por las tomas a lo ancho, y se define la variable *ruta [ ] [ ]* con un número de puntos de ruta definido por el valor total de tomas. Nótese que no es usual inicializar variables en el cuerpo del programa, pero en este caso era necesario pues hasta este punto no se conocía el tamaño de la variable a inicializar.

Una vez definida la variable *ruta [ ] [ ]* sólo queda rellenarla con los puntos correspondientes a la ruta a seguir por la cámara. Para esto se utilizan 3 bucles anidados, un primero para recorrer en el eje X la zona de estudio hacia la derecha y los otros dos para recorrer dicha zona hacia abajo y hacia arriba. El resto del funcionamiento es muy sencillo, para cada incremento de la variable *i* en x se produce un incremento en la variable j que recorre el rectángulo envolvente de arriba abajo, y al llegar a la arista inferior, sólo en caso de que la variable *i* no haya llegado a la arista derecha, se incrementa en una unidad y la variable *j* vuelve a decrecer hasta llegar a la arista superior, donde se repite el proceso hasta completar toda el área de estudio.

Una vez rellenada la variable *ruta [ ] [ ]* sólo queda devolverla al programa que había llamado a la función *Ruta()* para que empiece a recorrerla con la cámara y empiece el funcionamiento de los algoritmos de reconocimiento que se explicarán en apartados posteriores.

2. *Subprograma **Rectangulo()***

```
private static int[] Rectangulo() {
    int[] coordrect=new int[4];
    int[] vertice=new int[2];
    float[] REV=new float[2];
    float[] REH=new float[2];
    int recta=0, posvx=0, poshx=0, posvy=0, poshy=0, margen=100;
    poshx=cameraManager.getPosition(0);
    posvy=cameraManager.getPosition(1);

    while(recta!=1){
        cameraManager.moveX(LEFT, 2048);
        posvx=cameraManager.getPosition(0);
        BufferedImage  image = cameraManager.snapImage();
        recta = identRecta(image);
    }
    REV=ecuacionRectaVertical(image);
    cameraManager.moveXTo(poshx);
    recta=0;
    while(recta!=1){
        cameraManager.moveY(UP, 1536);
        poshy=cameraManager.getPosition(1);
        BufferedImage  image = cameraManager.snapImage();
        recta = identRecta(image);
    }
    REH=ecuacionRectaHorizontal(image);
    vertice[0]=(int) (REV[1]-REH[1]+posvx*REV[0]+posvy*REH[0]+posvy-poshy)/(REH[0]-REV[0]);
    vertice[1]=(int) (coordrect[0]+posvx)*REV[0]+REV[1]+posvy;
    if (REH[0] < 0)
        coordrect[0]=(int) vertice[0]-margen;
        coordrect[1]=(int) 2*poshy-vertice[1]-margen;
        coordrect[2]=(int) 2*poshx-coordrect[0]+margen;
        coordrect[3]=(int) 2*posvy-coordrect[1]+margen;
    else
        coordrect[0]=(int) 2*posvx-vertice[0]-margen;
        coordrect[1]=(int) vertice[1]-margen;
        coordrect[2]=(int) 2*poshx-coordrect[0]+margen;
        coordrect[3]=(int) 2*posvy-coordrect[1]+margen;
    return coordrect;
}
```

Figura 11: Clase Rectangulo(). Fuente: Elaboración propia

Este fragmento de código es llamado por la función *Ruta()* descrita en el apartado precedente para obtener las coordenadas de las aristas del rectángulo que engloba el cubre, por tanto es una función de tipo vector de enteros. Aquí el microscopio está situado sobre el centro del cubre.

La declaración de variables tiene 2aspectos a tener en cuenta. Primero está el hecho de que se definen dos variables tipo vector de floats, que contendrán los valores de la pendiente y el punto de corte con el eje y (m y n) de las rectas correspondientes a los bordes izquierdo y superior del cubre. El hecho de utilizar variables tipo float en un sistema de coordenadas enteras como son las pixélicas se justifica simplemente con la mayor precisión en los cálculos. Posteriormente se redondearán a valores enteros mediante casting. El otro aspecto a señalar en esta parte de declaración es la función *cameraManager.getPosition()*, que se encarga de devolver el valor de la coordenada X de la situación actual de la cámara (si el argumento es un 0) o de la coordenada Y (si el argumento es un 1). El prefijo “cameraManager” es necesario

pues la función “getPosition()” no está definida en la clase actual (main). No obstante, como se ha señalado anteriormente, la explicación de la sintaxis del lenguaje de programación JAVA se escapa a los objetivos del presente proyecto, por lo que no se entrará en detalle.

Una vez en la línea 10 empieza el cuerpo del programa con un bucle que se repite hasta que la variable *recta* (inicializada a “0”) toma el valor “1". En cada iteración del bucle se desplaza la cámara 2048 píxeles (1 toma) hacia la izquierda, se graba la posición en la variable *posvx* y se toma una imagen con la cámara que se guarda en un tipo especial de variable de imagen llamada *BufferedImage*. Una vez se tiene una imagen se llama a la función *identRecta*, a la que se le pasa la imagen tomada, y se guarda el valor de retorno de esa función en la variable *recta*. Este valor será un 1 si en la imagen aparecía el borde izquierdo del cubre y será un 0 en caso contrario.

En el caso de que no haya recta se volvería a realizar otra iteración del bucle hasta que se encuentre el borde del cubre o bien se alcance el límite de la bandeja del microscopio. En caso de detectar una recta se dará por finalizado el bucle y se llamará esta vez a la función *ecuacionRectaVertical()* para obtener la pendiente y punto de corte con el eje y de dicha recta.

Una vez detectado, se retorna a la posición de inicio, se vuelve a asignar a la variable *recta* el valor 0 y se repite el proceso para detectar el borde superior del cubre.

Ahora se calcula la posición del vértice superior izquierdo del cubre mediante resolución de un sistema de ecuaciones formado por las dos ecuaciones explícitas de la recta. Con estos valores calculados se entra en uno de los bloques condicionales, dependiendo de si la pendiente del borde superior es positiva o negativa (*REH[0]*<>0), lo que se deriva en modificaciones en el procedimiento de cálculo de aristas del rectángulo.

En las siguientes figuras se describen las distancias correspondientes a los cálculos de las coordenadas de las artistas del rectángulo de estudio, en función de si la inclinación del cubre se corresponde con un giro en el sentido de las agujas del reloj o en sentido opuesto:

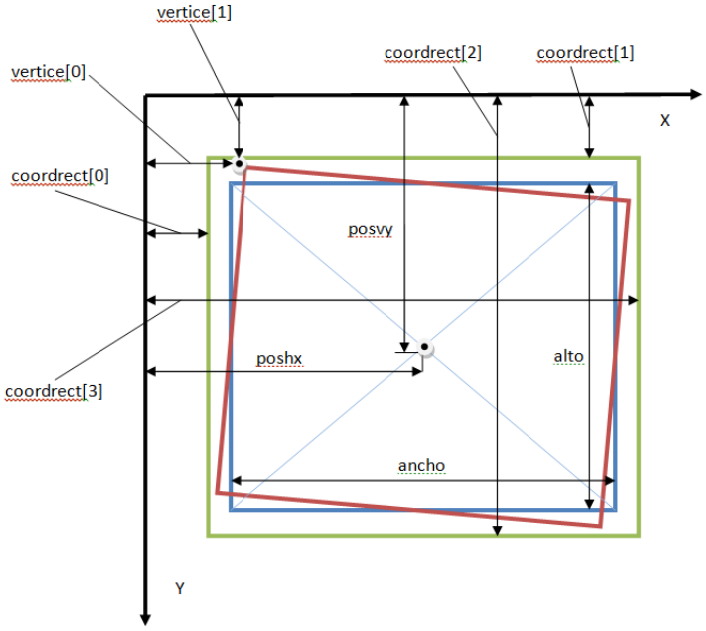

Figura 12: Distancias correspondientes a desfase del cubre en sentido de las agujas del reloj. Fuente: Elaboración propia

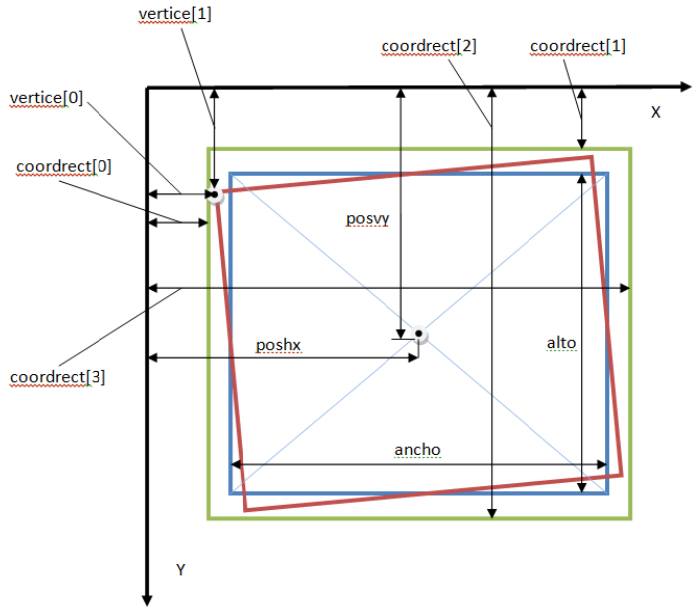

Figura 13: Distancias correspondientes a desfase del cubre en sentido contrario al de las agujas del reloj. Fuente: Elaboración propia

Con las coordenadas de las aristas del rectángulo exterior ya calculadas finaliza esta función al devolver los valores a la función *Ruta()*, que es la que la había llamado.

3. Subprograma **identRecta()**

Esta clase es llamada por la función *Rectangulo()* para procesar la toma de imagen actual en busca de un objeto que se corresponda con uno de los bordes del cubre. En caso de encontrarlo se llamará a otra función para caracterizar la recta de dicho borde y encontrar la posición del vértice superior izquierdo. A esta función *identRecta()* se la llama pasándole como argumento la imagen tomada en formato *BufferedImage*, y devuelve un valor entero que puede ser 1 en caso de haber encontrado una recta, tanto si se corresponde con el borde horizontal como el vertical, o un 0 en caso de no haber encontrado ninguna.

Puesto que este código es muy extenso (105 líneas), se va a dividir en 3 partes para su descripción.

```
1  private static int identRecta(BufferedImage image) {
2     int width = image.getWidth();
3     int height = image.getHeight();
4     int N=0;
5     int[] A= new int[8];
6     int min=255;
7     int max=0;
8     int[] V=new int[255];
9     int[] U=new int[255];
10    int contador=1;
11    int pixel=0;
12    int minx=0,maxx=0, miny=0, maxy=0;
13    ImageProcessor Proce= new ColorProcessor((Image) image);
14    ImageProcessor Binar= new ByteProcessor(width,height);
15    ImageProcessor Etiq= new ByteProcessor(width,height);
16    Proce.autoThreshold();
17    for(int x=1;x<width;x++){
18        for(int y=1;y<height;y++){
19            pixel=Proce.getPixel(x,y);
20            if (pixel==-1){
21                Binar.set(x,y,255);
22            }
23        }
24    }
25    for(int i=1;i<=8;i++){
26    Binar.erode();
27    }
28    for(int i=1;i<width;i++){
29        for(int j=1;j<height;j++){
30            if (Binar.getPixel(i,j)==255)  {
31                min=255;
32                max=0;
33                A[0] = Etiq.getPixel(i,j+1);
34                A[1] = Etiq.getPixel(i-1,j+1);
35                A[2] = Etiq.getPixel(i-1,j);
36                A[3] = Etiq.getPixel(i-1,j-1);
37                A[4] = Etiq.getPixel(i,j-1);
38                A[5] = Etiq.getPixel(i+1,j-1);
39                A[6] = Etiq.getPixel(i+1,j);
40                A[7] = Etiq.getPixel(i+1,j+1);
```

Figura 14: 1º Parte de la clase identRecta(). Fuente: Elaboración propia

Esta primera parte del código correspondiente a la clase *identRecta* comienza con la declaración e inicialización de algunas variables necesarias. Entre ellas cabe destacar las variables *Proce*, *Binar* y *Etiq*, pues son de un tipo de variable poco común.

Proce es una variable del tipo *ImageProcessor* que se obtiene realizando un casting de la variable *image*, que contiene la imagen de la toma actual. Este cambio es necesario para poder aplicarle a dicha imagen operaciones como el umbralizado o la dilatación, necesarias para este código.

Binar es también una variable del tipo *ImageProcessor*, pero a diferencia de *Proce*, se inicializa como una imagen vacía, de las mismas dimensiones que *image*. Es en Binar donde se alojará la imagen binarizada y en *Etiq* donde se registrarán los distintos objetos de la imagen como más adelante se verá.

Una vez definidas las variables el código empieza aplicando a la variable *Proce* una operación de umbralizado para obtener una imagen binaria. No obstante se comprobó que esta función *autoThreshold* propia de la clase *ImageProcessor* no resulta en una imagen de valores 0 o 255 únicamente, sino que aparecen -1 en los píxeles de fondo y valores intermedios que afectan al siguiente procesado de la imagen, por lo que esta se pre procesa mediante 2 bucles anidados que van recorriendo cada píxel de la imagen y registran un 255 en la posición correspondiente de la variable *Binar* en caso de detectar un valor de -1 (fondo) en *Proce*. Esto es así porque el borde se detecta como una línea negra que se correspondería con fondo en la función autoThreshold, así que hay que poner un 255 en *Binar* para los píxeles de fondo de *Proce*.

Una vez realizado este *umbralizado* se aplica una operación de erosión 8 veces para eliminar pequeños puntos de la imagen que habían quedado tras la operación anterior.

El resultado de este paso es el que se aprecia en las siguientes imágenes. La de la izquierda se corresponde con la variable *image*, y la de la derecha se corresponde con la variable *Binar*.

Una vez se tiene la imagen es necesario etiquetar los distintos objetos que en ella aparecen que no han sido eliminados con la erosión, para estudiar la semejanza de cada uno con una recta separadamente.

Para esto se han implementado dos bucles anidados que van recorriendo cada píxel de la variable *Binar*, en la que en este momento del código contiene la imagen binarizada del borde. Para cada píxel se determina si es un píxel de interés (255) o de fondo (0). En caso de ser un píxel de fondo se realiza una nueva iteración y se pasa al píxel siguiente. En caso contrario se pasa al siguiente bloque de código.

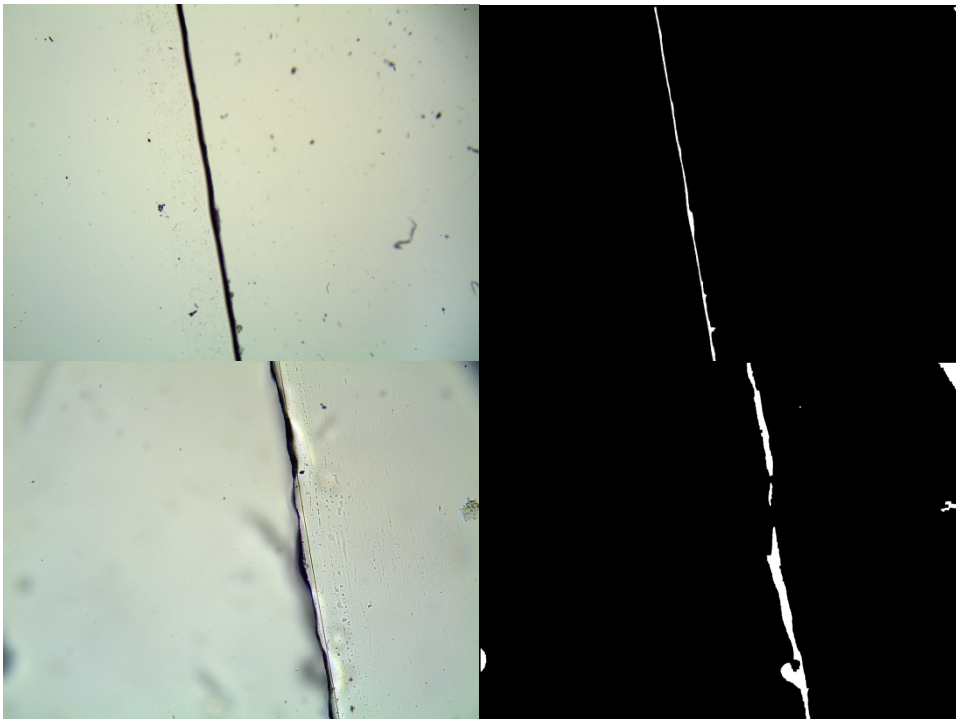

Figura 15: Pre procesado de la imagen de borde del cubre. Fuente: Elaboración propia

En este bloque se empieza inicializando las variables *min* y *max* a 255 y 0 respectivamente. Más adelante se verá su utilidad. Seguidamente se estudian los 8 vecinos del píxel y se registran sus valores en el vector *A*. Este vector se recorre a continuación para actualizar los valores de las variables *min* y *max* en función de los valores de los vecinos del píxel estudiado.

1	2	3
0	Pixel	4
7	6	5

Figura 16:Vecinos de un píxel. Fuente: Elaboración propia

```
41                  for (int k=0; k<=7;k++) {
42                      if (A[k]<min && A[k]!=0)
43                          min=A[k];
44                      if (A[k]>max)
45                          max=A[k];
46                  }
47                  if (max==0)  {
48                      N=N+1;
49                      V[N]=N;
50                      Etiq.set(i,j,N);
51                  }
52                  else  {
53                      Etiq.set(i,j,min);
54                      for (int k=0; k<=7;k++) {
55                          if (A[k]>min)
56                              V[A[k]]=min;
57                      }
58                  }
59              }
60          }
61      }
62      for (int i=0; i<255;i++){
63          if (V[i]!=i && V[i]!=0)
64              V[i]=V[V[i]];
65      }
66      U[0]=V[0];
67      for (int i=0; i<255;i++){
68          if (V[i]!=V[i-1] && V[i]!=0){
69              contador=contador+1;
70              U[i]=contador;
71          }
72          else if (V[i]!=0)
73              U[i]=U[i-1];
74      }
75      for (int i=2; i<width;i++){
76          for (int j=2; j<height;j++){
77              if (Etiq.getPixel(i,j)!=0)
78                  Etiq.set(i,j,U[Etiq.getPixel(i,j)]);
79          }
80      }
```

Figura 17: 2º Parte de la clase identRecta(). Fuente: Elaboración propia

En la primera iteración en la que se encuentra un píxel de interés *Etiq* estará vacía, y por tanto no existirán vecinos en el pixel de *Etiq*. En este caso *max* y *min* no variarán sus valores y se pasará al siguiente condicional que se habrá cumplido por ser *max*=0. Este condicional aumenta en 1 el valor de la variable *N*, que representa el número de objetos encontrados en la imagen. También registra el valor del objeto actual en el vector *V* y coloca dicho valor en el píxel de *Etiq* correspondiente al actual de *Binar*.

En la siguiente iteración que se detecte un píxel vecino del anterior, *min* y *max* valdrán 1, que es el valor asociado al objeto actual, y por tanto el programa entrará en el *else* de la línea 52, en el que se asignará al píxel correspondiente de *Etiq* dicho valor de objeto, y en caso de que hayan dos vecinos que se hayan etiquetado como objetos distintos, se dejará una referencia en el vector *V*, en el que se relacionarán dichos píxeles. A modo de ejemplo, si un píxel tiene un vecino con el número 2 y otro con el número 3, en *V[3]* donde originalmente había un 3, se situará un 2 para registrar la conexión entre los píxeles.

Una vez recorrida toda la imagen se han etiquetado los objetos que aparecen en ella, pero es posible que un objeto esté definido por varios números en píxeles conectados porque habían comenzado a etiquetarse desde distintos extremos. Para normalizar esta situación se ha implementado el siguiente bucle que actualiza el valor de cada posición con el valor de la posición correspondiente al valor actual. Una vez hecho esto se llena un segundo vector *U* con valores desde 1 hasta el número de objetos en función del contenido de *V*. Con el siguiente ejemplo se entenderá el proceso.

En este ejemplo aparecen 5 objetos diferentes, pero durante la detección del 3º objeto se ha empezado el etiquetado desde 3 inicios distintos, lo que ha ocasionado que se considerara un objeto como 3 diferentes y parecieran 7 objetos Mediante la 1º actualización de *V* se referencian las relaciones entre los píxeles del mismo objeto (*min*=3) Una vez realizada esta referencia se normalizan las referencias a un mismo objeto. Por último se forma el vector *U* en función de los distintos valores alojados en el vector *V*

Con este vector actualizado que contiene las referencias correctas a los objetos se corrige la imagen *Etiq* para obtener una imagen con todos los objetos correctamente etiquetados mediante dos bucles anidados.

```
81      for (int k=1; k<=contador;k++){
82          minx=width;
83          maxx=1;
84          miny=height;
85          maxy=1;
86          for(int i=1;i<width;i++){
87              for(int j=1;j<height;j++){
88                  if (Etiq.getPixel(i,j)==k){
89                      if (i<minx)
90                          minx=i;
91                      if (i>maxx)
92                          maxx=i;
93                      if (j<miny)
94                          miny=j;
95                      if (j>maxy)
96                          maxy=j;
97                  }
98              }
99          }
100     }
101     if(maxx-minx>0.4*width || maxy-miny>0.4*height)
102         return 1;
103     else if (k=contador)
104         return 0;
105 }
```

Figura 18: 3º Parte de la clase identRecta(). Fuente: Elaboración propia

En esta última parte de la clase *identRecta()* se recorre la imagen una vez por cada objeto para determinar los valores mínimos y máximos de las coordenadas X e Y de cada uno, y obtener así su anchura y altura. Esto servirá como criterio para determinar si dicho objeto se trata de una recta (ocupa más del 40% de la anchura o altura de la toma), en cuyo caso la función devolverá un 1, o no se trata de una recta, en cuyo caso bien seguirá estudiando el siguiente objeto o bien devolverá un 0 si se han estudiado todos los objetos que se habían registrado.

4. Subprogramas **ecuacionRectaHorizontal()** y **ecuacionRectaVertical()**

   Estos fragmentos de código constituyen la última parte del programa, y se encargan de detectar la recta que aparece en una imagen y obtener su ecuación general o implícita. Se van a comentar en un mismo apartado porque son códigos muy similares, que solo se diferencian en un par de líneas. Como se puede deducir, el primero se encargará de detectar la recta del borde superior del cubre y la segunda del borde izquierdo. Para la explicación de este apartado se va a dividir el código en 3 partes, de las cuales la 1º y la 3º son comunes a las dos clases.

```
1  private static float[] ecuacionRectaHorizontal(BufferedImage image) {
2      int width = image.getWidth();
3      int height = image.getHeight();
4      int pixel=0, k=0, cont=0, N=0, yesq, xesq;
5      int[] rectax=new int[6];
6      int[] rectay=new int[6];
7      float[] ecuacion=new float[2];
8      float Sx=0, Sy=0, Sxx=0, Syy=0, Sxy=0, m=0, n=0;
9      ImageProcessor imagen = new ColorProcessor((Image) image);
10     ImageProcessor bordes = imagen.duplicate();
11     ImageProcessor bordesbin= new ByteProcessor(width,height);
12     bordes.autoThreshold();
13     for(int x=1;x<width;x++){
14         for(int y=1;y<height;y++){
15             pixel=bordes.getPixel(x,y);
16             if (pixel!=-1){
17                 bordesbin.set(x,y,255);
18             }
19         }
20     }
21     for(int i=1;i<=8;i++)
22         bordesbin.erode();
```

Figura 19: 1º parte de la clase ecuacionRectaHorizontal(). Fuente: Elaboración propia

En este primer fragmento de código se realizan las inicializaciones de variables, de las que cabe destacar las variables *imagen*, *bordes* y *bordesbin*, que como en el apartado anterior, se obtienen mediante un casting inicial de la variable *image* para poder aplicarles operaciones especiales de imagen. También cabe destacar que esta función es de tipo *float[]* porque los coeficientes que devuelve son la pendiente y el punto de corte con el eje X de la recta.

Una vez declaradas estas variables se realiza la operación de *autoThreshold* a la variable bordes, que no es más que una copia de la variable imagen. Ahora se realiza un barrido de la imagen como en el apartado anterior para dejar en la variable *bordesbin* una imagen binaria con valores 0 (fondo) o 255 (objeto). Seguidamente se aplica una operación de erosión 8 veces para eliminar pequeños objetos que pudiesen aparecer en la imagen y reducir el grosor del borde para mejorar la detección de su recta directriz.

```
23      for(int i=0; i<=5; i++){
24          for(int j=200; j<1500;j++){
25              pixel=bordesbin.getPixel(20+400*i,j);
26              if (pixel!=0){
27                  k=j+1;
28                  while(true){
29                      pixel=bordesbin.getPixel(20+i*400,j);
30                      if(pixel==0 || k==1500)
31                          break;
32                      k++;
33                  }
34                  if(k-j>2){
35                      rectay[cont]=(int)(j+k-1)/2;
36                      rectax[cont]=(int) 20+300*i;
37                      cont++;
38                  }
39                  break;
40              }
41          }
42      }
```

Figura 20: 2º parte de la clase ecuacionRectaHorizontal(). Fuente: Elaboración propia

```
23      for(int i=0; i<=5; i++){
24          for(int j=200; j<1848;j++){
25              pixel=bordesbin.getPixel(j,20+300*i);
26              if (pixel!=0){
27                  k=j+1;
28                  while(true){
29                      pixel=bordesbin.getPixel(k,20+i*300);
30                      if(pixel==0 || k==1848)
31                          break;
32                      k++;
33                  }
34                  if(k-j>2){
35                      rectax[cont]=(int)(j+k-1)/2;
36                      rectay[cont]=(int) 20+300*i;
37                      cont++;
38                  }
39                  break;
40              }
41          }
42      }
```

Figura 21: 1º parte de la clase ecuacionRectaVertical(). Fuente: Elaboración propia

El siguiente apartado presenta ciertas diferencias para el caso de la recta vertical y la horizontal, aunque el funcionamiento es realmente sencillo. Básicamente lo que hace este fragmento de programa es recorrer la imagen binarizada en horizontal (para la detección de la recta vertical) o vertical (para la detección de la recta horizontal) en 5 tramos equidistantes en busca de un píxel de interés. Cuando lo encuentra registra ese punto en la variable k y sigue avanzando dejando fija la variable j en busca de un píxel de fondo (que indica que se sale de la recta). Una vez encuentra este píxel de fondo sale del bucle y determina si la resta entre la variable j y la k es mayor a 2 píxeles (esto representa el grosor de la recta, y se realiza para evitar falsos positivos con píxeles externos a la recta). En caso positivo registra la posición de la mitad del grosor y el tramo que ha procesado de los 5 y pasa al siguiente tramo.

```
43      N=rectax.length;
44      for(int i=0;i<N;i++){
45          Sx+=rectax[i];
46          Sy+=rectay[i];
47          Sxx+=rectax[i]*rectax[i];
48          Syy+=rectay[i]*rectay[i];
49          Sxy+=rectax[i]*rectay[i];
50      }
51      m=(N*Sxy-Sx*Sy)/(N*Sxx-Sx*Sx);
52      n=(Sxx*Sy-Sx*Sxy)/(N*Sxx-Sx*Sx);
53      ecuacion[0]=m;
54      ecuacion[1]=n;
55      return ecuacion;
56  }
```

Figura 22: 3º parte de la clase ecuacionRectaHorizontal(). Fuente: Elaboración propia

Una vez se han procesado los 5 tramos se obtienen 5 puntos que pertenecen al borde, pero no forman exactamente una recta, por lo que es necesaria una interpolación por mínimos cuadrados idéntica tanto para la recta horizontal como para la vertical con el objetivo de obtener los parámetros de dicha recta. Esto se realiza mediante las operaciones del bucle siguiente[5], después de las cuales solamente queda devolver los valores como retorno de la función.

Se han realizado pruebas en MatLab dibujando la recta en color rojo correspondiente a los coeficientes devueltos por el programa en Java sobre las imágenes, y los resultados demuestran el correcto funcionamiento del programa.

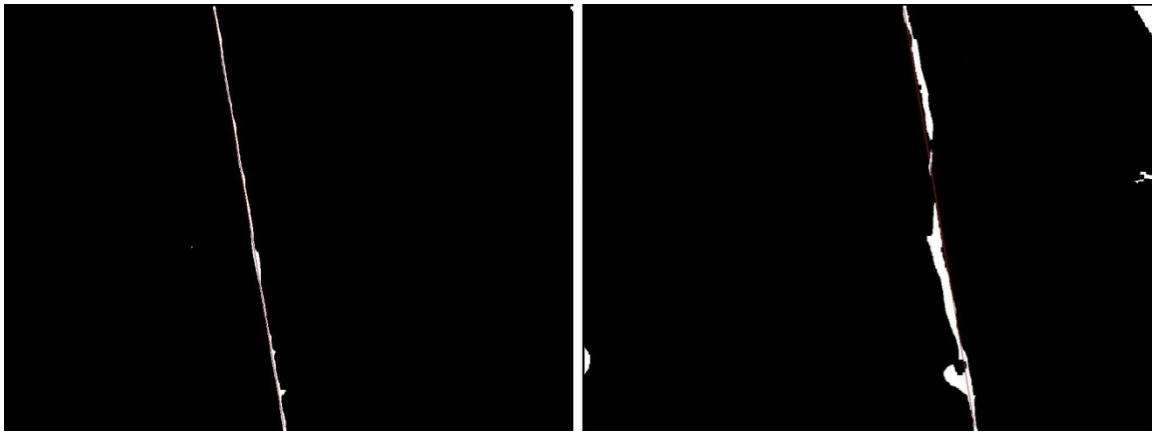

Figura 23: Identificación de 2 rectas del borde del cubre. Fuente: Elaboración propia

[5] Créditos al profesor Juan Zúñiga Román, del departamento de física atómica, molecular y nuclear de la Universidad de Valencia por el método de ajuste por mínimos cuadrados.

2.3 Autoenfoque de la cámara

2.3.1 Problema a resolver

Una vez programada la ruta de toma de imágenes que englobaría el cubre, es necesario disponer de una función que se encargue de obtener imágenes enfocadas, pues la efectividad en la posterior detección de helmintos depende en gran medida de calidad de la imagen tomada y la cámara utilizada es de foco fijo. Pese a que existen funciones que realizan esta operación directamente, se decidió implementar una nueva función propia para depender en la menor medida posible de programas de terceros.

2.3.2 Opciones consideradas

Existen librerías con funciones que realizan esta operación directamente, pero se decidió implementar una nueva función propia para depender en la menor medida posible de programas de terceros.

En primer lugar se consideró necesaria una función que actuara en cada zona a explorar del cubre, pues la posible inclinación con respecto al plano horizontal que tuviera este conllevaría que con la cámara enfocada en una zona, se tomaran imágenes desenfocadas en otra.

No obstante, después de realizar pruebas con el microscopio se advirtió que este desenfoque no se producía a no ser que hubiesen partículas de tamaño considerable debajo del porta, situación que no se produce por la inspección del técnico de laboratorio, por lo que finalmente se implementará una función que obtenga la posición del eje vertical correspondiente con un enfoque óptimo de todas las tomas que se vayan a realizar con esa muestra. Esta operación se realizará al inicio del programa principal, justo antes de empezar con el algoritmo de determinación de la trayectoria.

2.3.3 Desarrollo de la solución

El programa que se encargará de colocar la cámara en una posición del eje vertical correspondiente al enfoque máximo está formado por 2 subprogramas. La posición inicial de la cámara será tal que un desplazamiento hacia abajo resultará en un mejor enfoque.

El primero se define como *autofocus()* y es la clase que se llama para realizar el autoenfoque, sin tener parámetros de entrada ni de salida. El segundo es *calculoVarianza()*, y es una subclase de *autofocus()* dedicada a cuantificar la cantidad de píxeles correspondientes a bordes en la imagen tomada, parámetro que se utilizará como indicador del enfoque como se explicará más adelante.

1. Subprograma **autofocus()**

```
public void autofocus() {
    double incremento=1, varianza, varianza1;
    ImageProcessor image = new ColorProcessor((Image) cameraManager.snapImage());
    varianza=calculoVarianza(image);
    while(incremento>0){
        cameraManager.moveZ(DOWN, 5);
        varianza1=varianza;
        ImageProcessor image = new ColorProcessor((Image) cameraManager.snapImage());
        varianza=calculoVarianza(image);
        incremento=varianza-varianza1;
    }
    incremento=1;
    while(incremento>0){
        cameraManager.moveZ(UP, 1);
        varianza1=varianza;
        ImageProcessor image = new ColorProcessor((Image) cameraManager.snapImage());
        varianza=calculoVarianza(image);
        incremento=varianza-varianza1;
    }
    cameraManager.moveZ(DOWN, 1);
}
```

Figura 24: Clase autofocus(). Fuente: Elaboración propia

Esta primera clase definida *autofocus()* es la que hay que llamar desde la clase *main()* para mover la cámara a una posición en el eje vertical correspondiente al máximo enfoque de las imágenes. Es por tanto una clase sin argumentos de entrada ni salida.

El programa comienza declarando 3 variables auxiliares tipo *double* e inicializando una de ellas a “1”. Seguidamente se declara una variable tipo *ImageProcessor* llamada *image*, que es en esencia un casting de la variable tipo *BufferedImage* que retorna la clase *snapImage()*. Como se ha señalado en la explicación del apartado precedente, este casting se realiza para tener la posibilidad de aplicar a dicha imagen operaciones propias de la clase *ImageProcessor*.

Una vez se tiene la variable *image*, que contiene la imagen actual probablemente desenfocada, se llama a la clase *calculoVarianza()* para obtener el parámetro que servirá de indicador de la calidad del enfoque y que se asociará a la variable varianza. Esta clase auxiliar se desarrollará en el apartado siguiente.

Ahora se entra en un bucle tipo *while()* condicionado por el valor de la variable incremento, que debe ser positivo para volver a entrar en el bucle. Todavía no se ha calculado un valor para esta variable, razón por la que ha sido necesaria su inicialización a “1” para asegurar su primera entrada al programa.

Como primera acción dentro del bucle se lleva a cabo un desplazamiento hacia abajo en el eje Z. Este desplazamiento es de 5 pasos del motor paso a paso que controla el tornillo de potencia que mueve la cámara en el eje Z. Seguidamente se almacena el valor de la variable varianza en varianza1, para poder compararlos posteriormente, y se toma una nueva imagen en la posición actual para calcular a continuación la nueva varianza.

Con la resta entre la varianza de la imagen actual y la de la imagen anterior se obtiene el incremento de varianza, que debería ser positivo si se ha enfocado la imagen al descender. En este caso se vuelve a entrar al bucle y se realiza un nuevo descenso. En el momento en

que el incremento de varianza dé un número negativo significará que se habrá sobrepasado el punto de máximo enfoque, por lo que no se volverá a entrar en el bucle.

Para regresar al punto de enfoque se vuelve a inicializar la variable incremento a “1” y se ejecuta un bucle idéntico al anterior salvo por dos aspectos. El primero de ellos es que ahora el sentido del desplazamiento es ascendente, ya que se había sobrepasado el punto de máximo enfoque. El segundo aspecto es que ahora en cada iteración se subirá una distancia igual a 1 paso del motor, lo que permite obtener un punto más preciso.

Al igual que en el primer bucle, cuando se llegue a un punto en el que el incremento sea negativo significa que la cámara está un paso por encima del punto de máximo enfoque, por lo que el bucle termina y la última línea de código del programa ejecuta un desplazamiento final de la cámara 1 paso en sentido descendente para situarla en dicho punto.

Con la finalización de la ejecución del programa, el sistema queda calibrado para tomar imágenes enfocadas de toda el área de estudio.

2. Subprograma **calculoVarianza()**

```
double calculoVarianza(Imageprocessor image) {
    double varianza = 0;
    int ancho = image.getWidth();
    int alto = image.getHeight();
    image.findEdges();
    double var = 0;
    for (int i = 0; i < ancho; i++) {
        for (int j = 0; j < alto; j++) {
            var = var+image.getPixel(i, j);
        }
    }
    varianza = var/(ancho*alto);
    return varianza;
}
```

Figura 25: Clase calculoVarianza(). Fuente: Elaboración propia

En esta subclase auxiliar de *autofocus()* se calcula un parámetro que se aloja en la variable *varianza*, e indica el mayor o menor grado de enfoque de la imagen.

El programa se basa en el hecho de que en una imagen enfocada se detectan mejor los bordes, por lo que contabilizando la cantidad de píxeles que forman la imagen de bordes obtenida mediante la función *findEdges()* se puede obtener un indicio del enfoque.

La función *findEdges()* es un método de la clase *ImageProcessor* y su función es aplicar el operador Sobel a la imagen que se le pasa como argumento. El resultado es una imagen con “1” en los píxeles correspondientes a bordes y “0” en el resto. Contando estos píxeles y dividiéndolos por la cantidad de píxeles totales se obtiene el porcentaje de píxeles de borde de la imagen.

Puesto que la imagen es la misma, la única razón para que una imagen con distinto enfoque que otra devuelva una densidad de bordes mayor es que dicho menor enfoque haya producido que un borde se difumine y no se detecte mediante el operador Sobel.

El operador Sobel se utiliza en el procesamiento de imágenes fundamentalmente para la detección de los bordes de los objetos de una imagen y se basa en el cálculo del gradiente de la intensidad de los píxeles que la componen. Dicho gradiente se obtiene para las direcciones X e Y independientemente y luego se calcula la norma de dichos valores para obtener el gradiente global. Este cálculo se realiza en un área de 3x3 píxeles alrededor del píxel objetivo, por lo que es un algoritmo bastante simple, pero cuyo resultado es aceptable para la mayoría de aplicaciones de visión artificial como la presente.

Las operaciones realizadas para cada dirección del gradiente son las siguientes matrices de convolución:

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{y} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Figura 26: Máscaras de convolución para filtro Sobel. Fuente: Wikipedia

Con estos valores se puede obtener tanto el módulo como la dirección del gradiente. No obstante para la presente aplicación solo va a ser necesario el cálculo del módulo, que es lo que devuelve la función *findEdges()*.

2.3.4 Resultados Autoenfoque

Se han realizado pruebas en MatLab para comprobar este funcionamiento con una imagen enfocada y otras 5 imágenes desenfocadas a 5 distancias distintas mediante el software de edición de imágenes Photoshop, obteniendo un resultado que defiende la viabilidad de este método como indicador del enfoque, apreciando un claro descenso en el porcentaje de borde en imágenes desenfocadas con un grado de desenfoque prácticamente inapreciable para la vista. Las imágenes utilizadas junto a los porcentajes de borde detectados se muestran en la página siguiente.

Figura 27: Imagen de ensayo enfocada.
Porcentaje: **3.62%**

Figura 28: Imagen de ensayo desenfocada 1.
Porcentaje: **3.35%**

Figura 27: Imagen de ensayo desenfocada 2.
Porcentaje: **2.78 %**

Figura 30: Imagen de ensayo desenfocada 3.
Porcentaje: **2.31%**

Figura 31: Imagen de ensayo desenfocada 4.
Porcentaje: **2.17%**

Figura 32: Imagen de ensayo desenfocada 5.
Porcentaje: **2.08%**

2.4 Detección y clasificación

2.4.1 Problema a resolver

Una vez resueltos los desafíos relativos a la obtención de imágenes del área de estudio con un enfoque adecuado es momento de implementar la función encargada de detectar y clasificar los huevos de helmintos en una de las 5 clases requeridas (Áscaris, Hymenolepis, Trichuris, Schistosoma y Taenia). Una pequeña cantidad de errores dados por considerar como huevo algún artefacto son asumibles, mientras que el caso opuesto (considerar como artefacto un huevo) no se puede dar en ningún caso para que el programa sea validado.

2.4.2 Opciones consideradas

Para conseguir el objetivo de la detección y clasificación de huevos de helminto se pensó inicialmente en un sistema de procesamiento píxel a píxel mediante segmentación, separando en primer lugar los píxeles de interés de los píxeles de fondo para posteriormente etiquetar los distintos objetos y clasificarlos según una serie de características como color, forma o momentos invariantes.

No obstante, por razones que se verán en el punto siguiente cuando se desarrolle la solución fue necesario abandonar esta línea de trabajo. Como alternativa se planteó el utilizar un algoritmo basado en Histograma de Gradientes, una técnica que no trabaja píxel a píxel como la anterior.

2.4.3 Desarrollo Segmentación

El funcionamiento del algoritmo de esta opción comienza con el mismo programa que se utilizó para la detección de la recta del borde del cubre salvo por una diferencia: Antes de etiquetar cada objeto en la imagen ReEtiq se valora si las dimensiones de dicho objeto son del orden de las que tendría un posible huevo[6], de forma que se eliminan los artefactos de menor tamaño y los bordes de la imagen que se podrían interpretar como objetos de interés.

Una vez se tiene la imagen etiquetada con objetos susceptibles de ser huevos por tamaño e intensidad de píxel se pasa a la siguiente parte del código, encargada de generar la máscara correspondiente a cada objeto para mostrar los píxeles correspondientes de la

[6]Mediante repetidos ensayos se ha situado el umbral inferior en 50 píxeles y el superior en 1000 píxeles

imagen original. De esta manera se puede estudiar si los parámetros de calibración del filtro de tamaño de objeto son correctos o es necesario modificarlos.

En la generación de esta máscara se crea una imagen en la que los píxeles del objeto correspondiente tienen el valor “1” y el resto “0” para aplicar a continuación una operación de dilatación 5 veces y así visualizar un área más grande que el objeto y comprobar su entorno. El fragmento de código encargado se muestra en la siguiente figura:

```
for i=1:Objetos
    mascara=zeros(alto,ancho);
    mascara(ReEtiq==i)=1;
    SE=strel('arbitrary',[1 1 1;1 1 1;1 1 1]);
    for k=1:5
        mascara=imdilate(mascara,SE);
    end
    mascara=cat(3,mascara,mascara,mascara);
    if (i==1)
        Objetiva=(uint8(image).*uint8(mascara));
    else
        Objetiva=Objetiva+(uint8(image).*uint8(mascara));
    end
    imshow(Objetiva);
end
```

Figura 32: Código de generación de máscara. Fuente: Elaboración propia

2.4.4 Resultados Segmentación

Después de varios ensayos con todos los tipos de helminto se ha comprobado que los parámetros de tamaño eran acertados, pues reducían la cantidad de artefactos detectados como posibles huevos sin dejar ningún huevo por detectar en los huevos de las clases Áscaris y Taenia.

No obstante los huevos de la clase Hymenolepis no se detectaban por tener un contorno demasiado poco definido, lo que ocasionaba que los artefactos destacaran más que los propios huevos, lo que sería equivalente a una señal en la que el ruido es superior a la propia señal.

También habían problemas si los huevos estaban en contacto con un objeto tal que el tamaño de ambos fuera superior al umbral de tamaño como ocurre en la imagen del huevo de la clase Trichuris, lo que ocasionaba que el filtro de tamaño considerara ambos objetos como artefactos, situación inasumible.

Con la clase Schistosoma se repetía el primer problema, ya que la muestra de agua estaba tan contaminada con artefactos que se hacía prácticamente imposible detectar los huevos.

Debido a estas razones, se decidió abandonar esta línea de trabajo y probar con un sistema de identificación basado en ventanados y gradientes de imagen que se desarrollará en el apartado siguiente.

A continuación se muestran los resultados de las pruebas del algoritmo con los huevos señalados mediante un círculo rojo. Se aprecia los casos en los que el algoritmo acierta y los casos en los que descarta los huevos (inasumible):

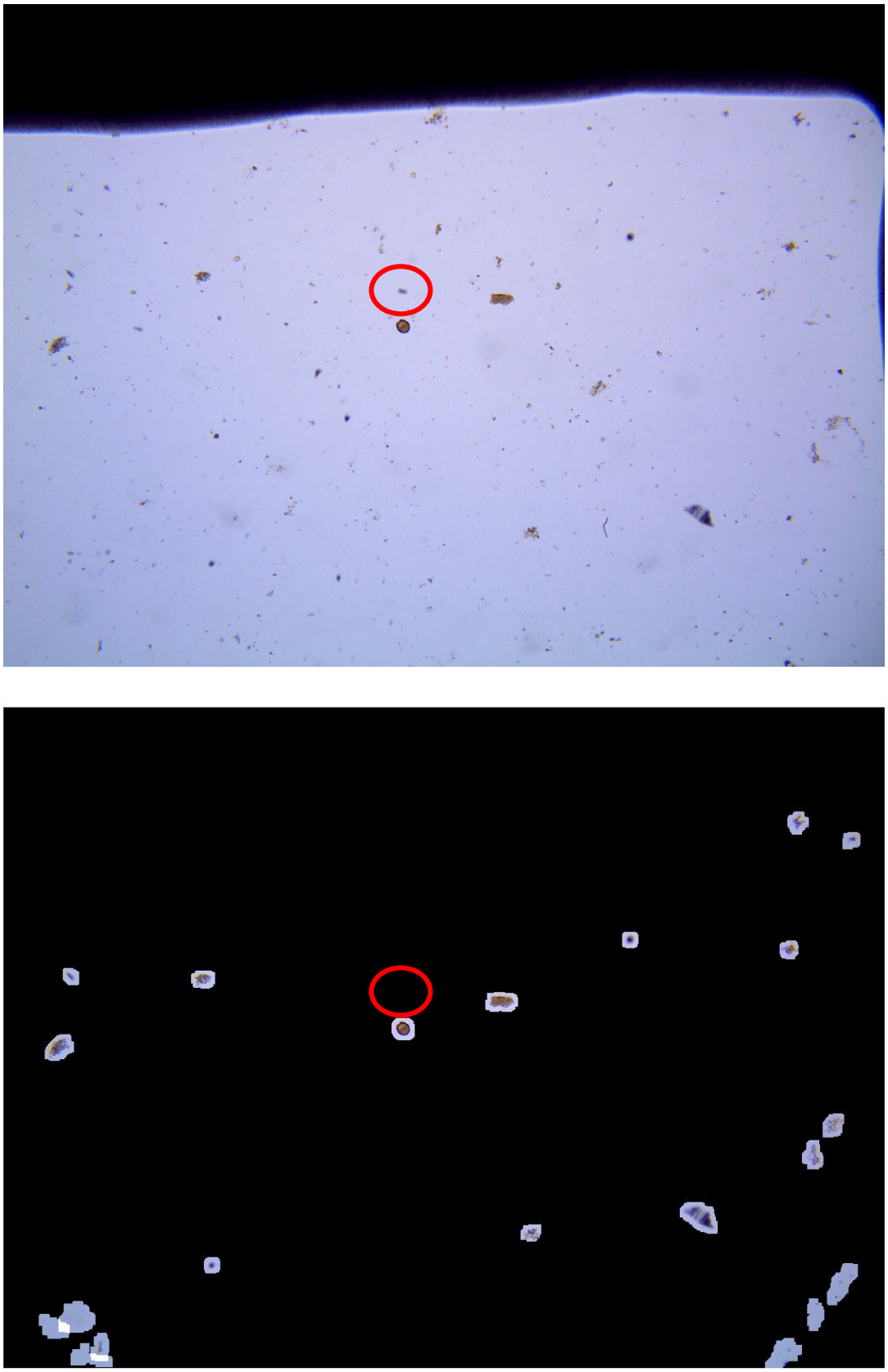

Figura 33: Resultado Áscaris. Fuente: Elaboración Propia

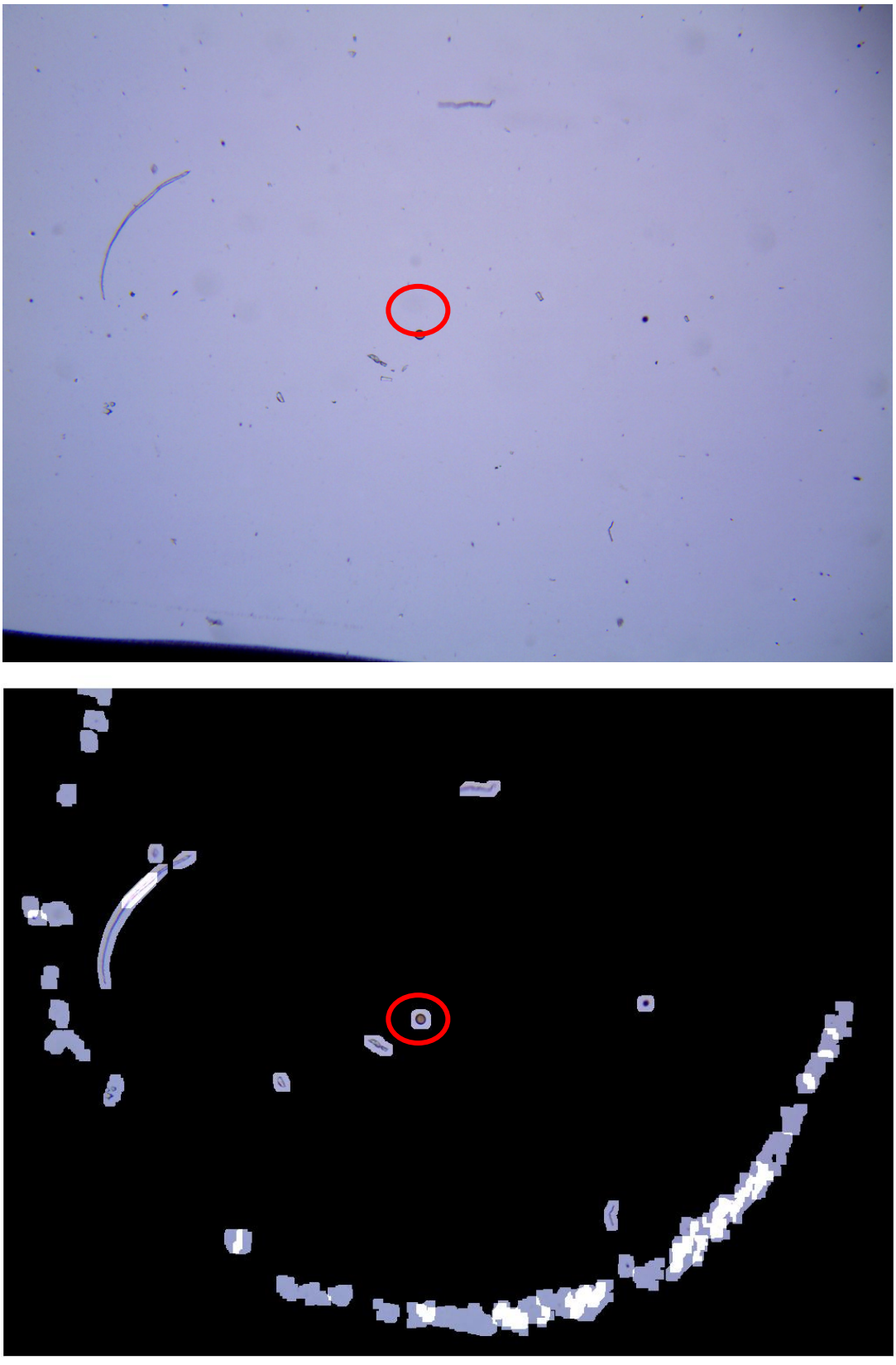

Figura 34: Resultado Taenia. Fuente: Elaboración propia

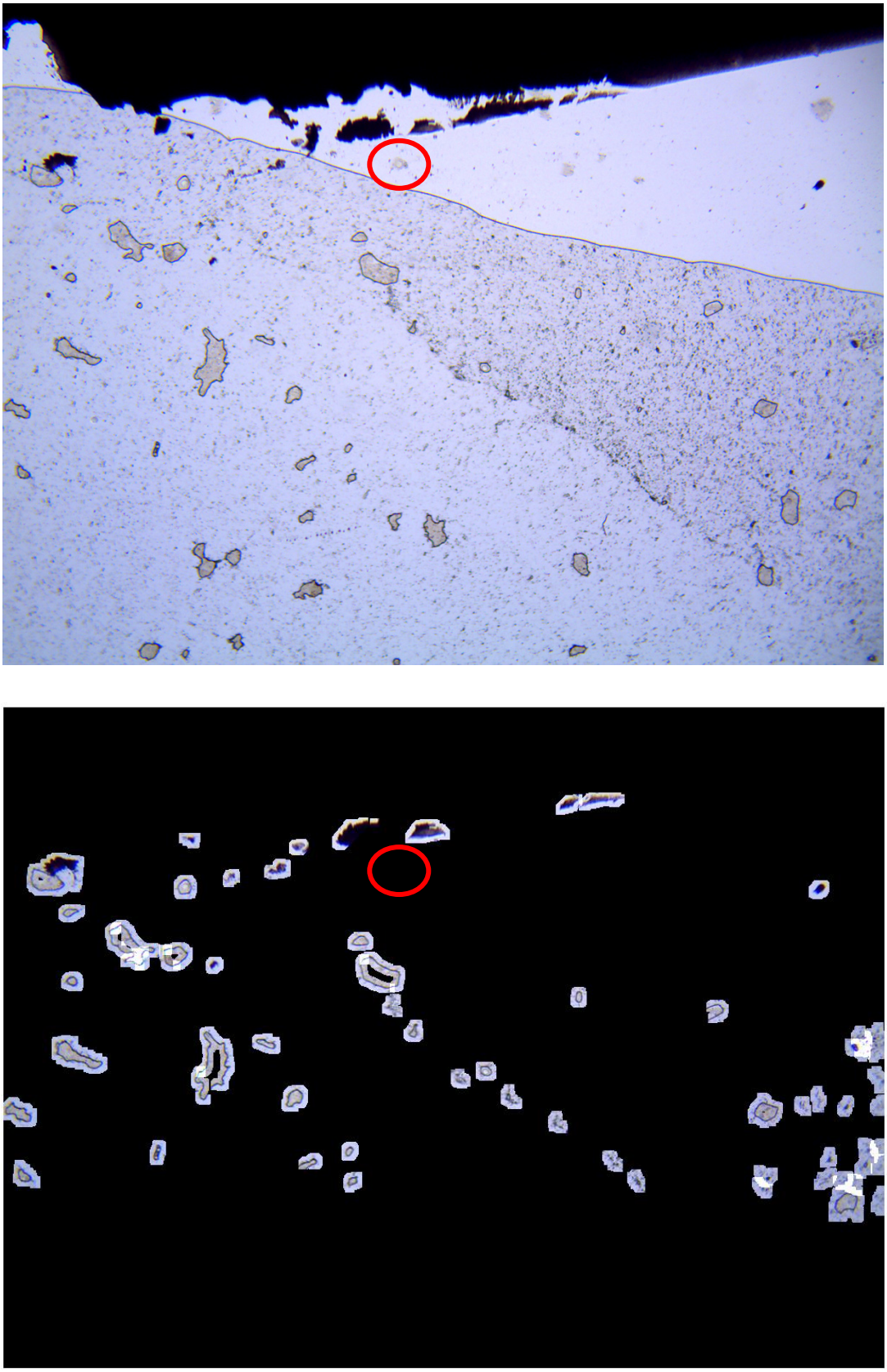

Figura 35: Resultado Hymenolepis. Fuente: Elaboración propia

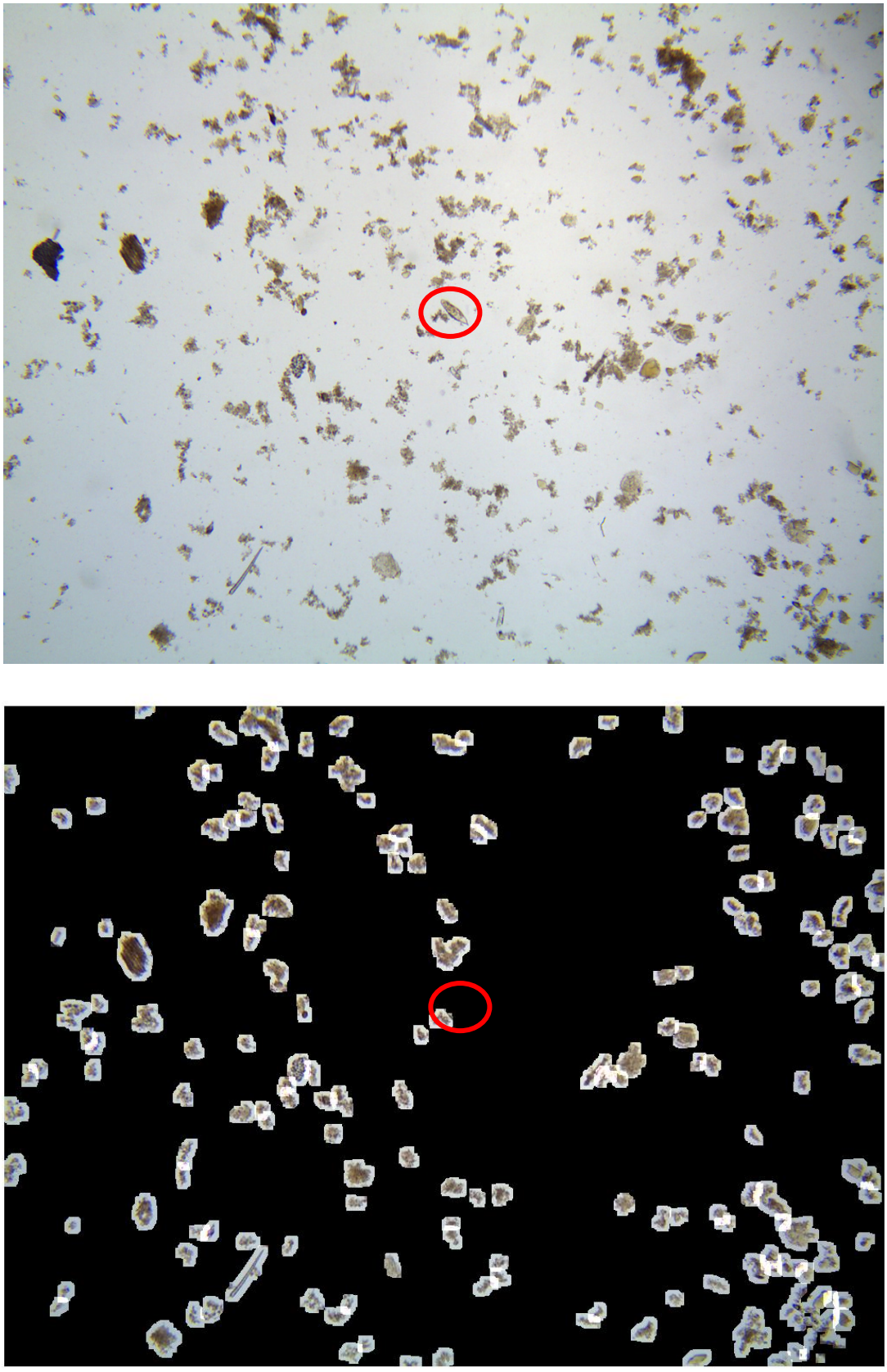

Figura 36: Resultado Schistosoma. Fuente: Elaboración propia

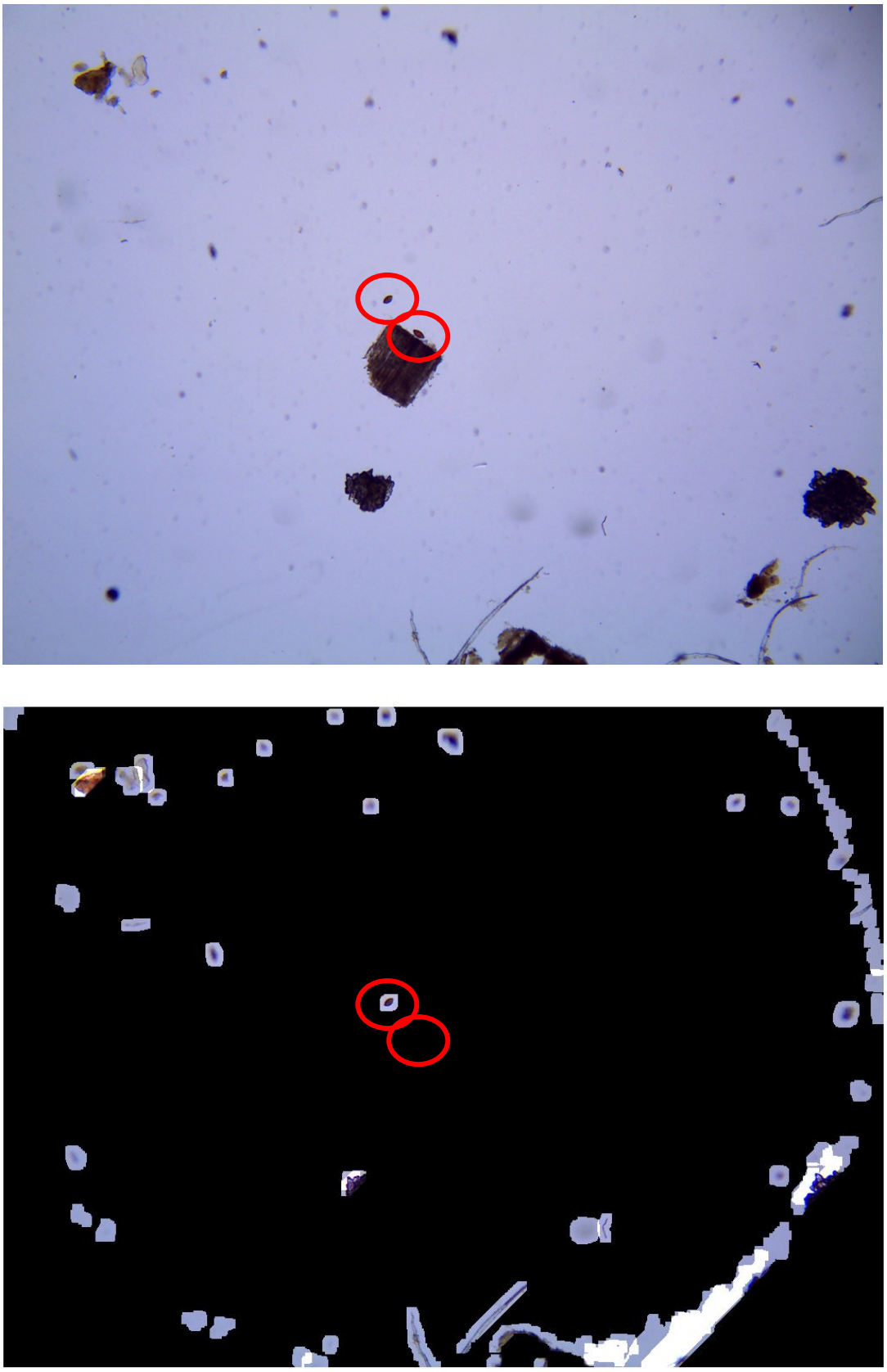

Figura 37: Resultado Trichuris. Fuente: Elaboración propia

2.4.5 Desarrollo Histograma de Gradientes y clasificador k-NN

La segunda línea de trabajo por la que continuó el desarrollo de la aplicación fue el uso de un sistema de identificación basado en Histogramas de Gradientes Orientados (HOG). Este sistema se basa en dividir la imagen en todas las ventanas posibles de unas dimensiones predefinidas para obtener una "firma" de cada una en función de la dirección que toman los gradientes de los píxeles.

Posteriormente se compara cada firma con una serie de firmas almacenadas que se corresponden con los objetos a detectar buscando similitudes que indiquen la presencia de los objetos en dicha imagen. El proceso se detalla a continuación:

1. El sistema comienza mediante la separación de la imagen en ventanas de un tamaño predefinido. Las medidas de esta ventana dependen del tamaño de las características que van a definir el objeto, y no del tamaño del propio objeto. A modo de ejemplo, si se desea detectar un objeto cuadrado de dimensiones 200x300 píxeles que tiene en su interior una marca con un color, textura o borde característico de tamaño en torno a 10x10 píxeles, el tamaño de ventana debería ser lo más cercano al de esa marca característica con un margen de seguridad (por ejemplo 12x12 píxeles). Es importante recalcar el hecho de que las ventanas se solapan hasta definir todas las posibles en la imagen, hecho que se entiende con claridad mediante la siguiente imagen. En ella se muestran las 9 posibles ventanas de 3x3 píxeles que hay en una imagen de 5x5 píxeles. Nótese la elevada carga computacional que conlleva este método al tener que operar cada ventana en una imagen estándar de 2048x1536 píxeles.

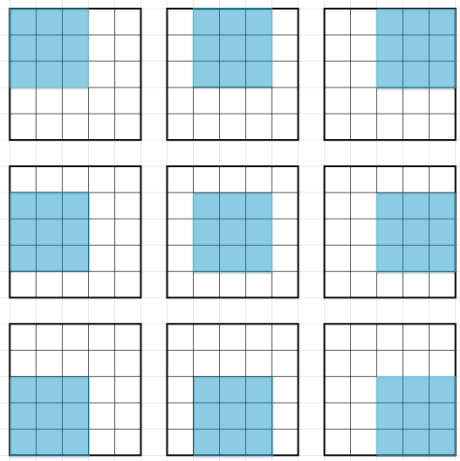

Figura 29: Ventanado de 3x3 en imagen de 5x5 píxeles. Fuente: Elaboración propia

2. El siguiente paso en el proceso es la obtención de las direcciones de los gradientes de cada una de las ventanas en que se ha dividido la imagen. Para ello primero se aplica un operador de detección de bordes a la imagen como Sobel o Prewitt. Esta operación en MatLab se realiza mediante la función *imgradient()*, que devuelve tanto la magnitud como la dirección del gradiente de cada píxel. No obstante, este método de detección no

tiene en cuenta la magnitud del gradiente, lo que lo hace insensible a las condiciones de iluminación. En las últimas versiones de MatLab existe una función llamada *extractHOGFeatures()* que permite la extracción y visualización de dichos gradientes.

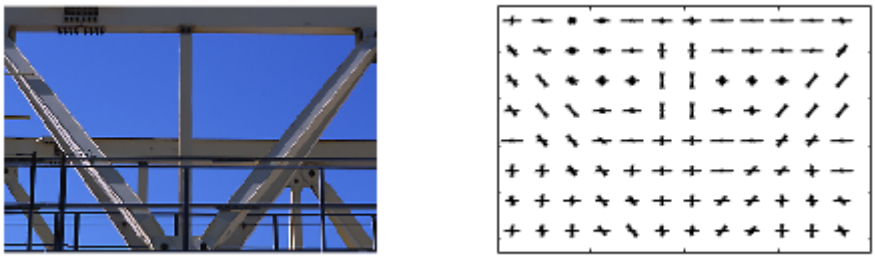

Figura 30: Ejemplo de extracción de gradientes de una imagen. Fuente: MathWorks

3. A continuación se debe obtener una "firma" en función de las direcciones que toman los gradientes de la imagen mediante su discretización. Estos gradientes toman valores continuos desde -180º a casi +180º, por lo que es necesario obtener un histograma para que el registro tenga un tamaño asequible dividiendo esos 360º en varios valores. Entre más valores compongan el histograma mayor precisión se tendrá, pero también se incurrirá en una carga computacional más elevada. Así, el histograma puede estar definido para cada 10º habiendo 18 opciones (bins), o para cada 5º habiendo 36 opciones.

   Este histograma puede ser codificado en un vector con un tamaño igual a esas componentes para su comparación. A continuación se muestra la obtención del vector descriptor en función de la distribución de gradientes en una ventana ejemplo.

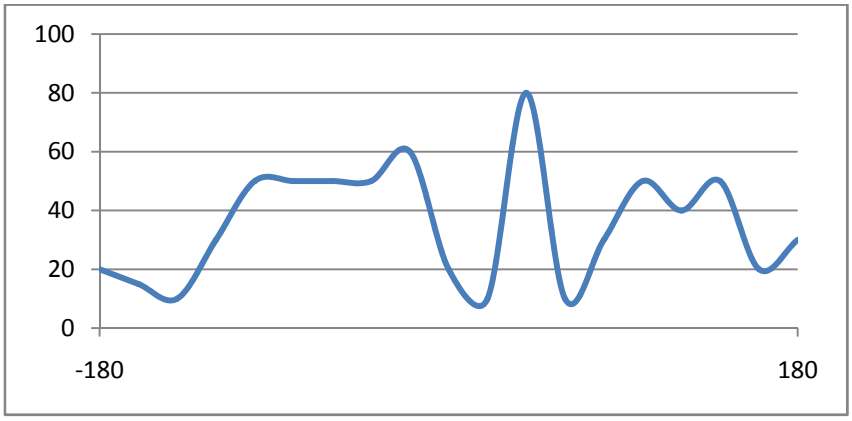

Figura 40: Distribución de gradientes en una ventana. Fuente: Elaboración propia

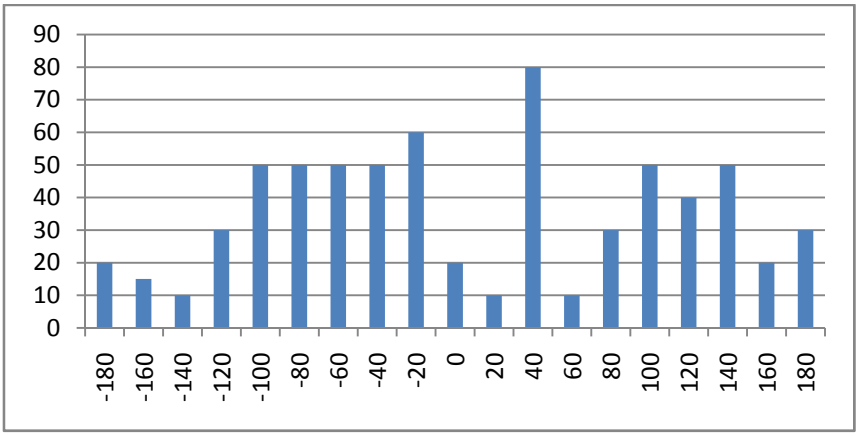

Figura 41: Histograma de gradientes para pasos de 20º. Fuente: Elaboración propia

$$V_{1,1} = \{20,15,10,30,50,50,50,50,60,20,10,80,10,30,50,40,50,20,30\}$$

Figura 31: Vector descriptor de la ventana 1,1 con 19 bins. Fuente: Elaboración propia

Estos tres primeros pasos son los inherentes al algoritmo de obtención del Histograma de Gradientes Orientados, y en su implementación en el código de la presente aplicación se ha optado por utilizar la librería de procesamiento de imágenes conocida como OpenCV, que presenta gran variedad de funciones entre las que se encuentra el cálculo del descriptor basado en histograma de gradientes.

El código utilizado es considerablemente extenso, por lo que para el presente desarrollo se ha considerado más acertada la explicación de las funciones más importantes, que son las encargadas de obtener el descriptor, obviando el resto de código encargado fundamentalmente de cálculos y operaciones auxiliares.

La función de obtención del descriptor HOG no es propia de JAVA, por lo que hay que incluir la librería en la que se haya mediante la siguiente instrucción al principio del programa:

```
import org.opencv.objdetect.HOGDescriptor;
```

Figura 42: Importación de la librería que incluye la clase HOGDescriptor. Fuente: Elaboración propia

Una vez se dispone de la función *HOGDescriptor()* es necesario definir un objeto de tipo *HOGDescriptor*, llamado *hog*, por cada imagen a procesar, añadiendo como argumentos una serie de parámetros relativos a las medidas (en un tipo de variable llamado *Size*) de las ventanas en las que dividir la imagen, las celdas que las componen y el número de direcciones en las que discretizar las direcciones (bins).

```
HOGDescriptor hog = new HOGDescriptor(/*Argumentos*/);
```

Figura 43: Definición del objeto tipo HOGDescriptor. Fuente: Elaboración propia

Con el descriptor definido es momento de llenarlo con los datos de cada imagen, para lo que se implementa la siguiente línea de código, en la que se llama al método *compute()*

de la clase *HOGDescriptor* y se le pasa como argumentos en primer lugar la imagen de la que se desea obtener el descriptor (en formato *Mat*) y en segundo lugar otra variable tipo *Mat* donde alojar el resultado.

```
hog.compute(imagen, descriptor);
```

Figura 44: Instrucción para obtener el descriptor de la imagen correspondiente. Fuente: Elaboración propia

Con esto ya se dispone en la variable *descriptor* de todos los datos referentes a los Histogramas de Gradientes Orientados obtenidos para cada una de las ventanas en que se ha dividido la imagen.

4. El último paso del proceso consiste en comparar la firma de cada ventana con una base de datos en la que previamente se han registrado las firmas de ventanas que contienen los objetos a buscar en una fase conocida como “aprendizaje”. De esta comparación se extraerá la conclusión de si en la imagen tomada aparece algún objeto de interés, en qué ventana lo hace y de qué objeto se trata. Este paso se realiza mediante un algoritmo denominado Clasificador.

   Un clasificador es un algoritmo que se encarga de procesar características de una entidad y decidir la probabilidad que hay de que dicha entidad pertenezca a una serie de conjuntos predefinidos. A modo de ejemplo, el clasificador es el encargado de decidir si la imagen de una fruta determinada pertenece al subconjunto “plátanos”, “manzanas”, “peras” etc. en función de una serie de características y otros parámetros que dependen del tipo de clasificador utilizado. Entre estos tipos destacan los siguientes por su sencillez y alto porcentaje de acierto:

   1. Clasificador Bayesiano Ingenuo

   Este clasificador probabilístico paramétrico basa su funcionamiento en el teorema de Bayes, y considera que todas las características que definen la pertenencia de un objeto a una clase son independientes entre ellas (de aquí el atributo de “ingenuo”). Este método presenta la principal ventaja de que pueden estimarse los parámetros del clasificador con un conjunto relativamente pequeño de datos, pues al suponerse independientes las variables características solo es necesario estimar las varianzas de cada una, y no la matriz de covarianza completa.

   El procedimiento de operación de este clasificador comienza con la calibración del mismo. Para esto debe disponerse de un set de datos de aprendizaje con las características que pretendan utilizarse en la identificación. Cuanto más grande sea este set de datos más preciso será el clasificador.

   Una vez se dispone de los datos debe obtenerse el valor de la media y la desviación típica de cada característica para cada clase. En caso de un clasificado no ingenuo sería necesario calcular la matriz de covarianzas, puesto que se asumiría una relación no despreciable entre las características a estudiar.

$$\mu_{jk} = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i$$

Ecuación 1: Cálculo de la media de los **N** valores **x** de la característica **j** para la clase **k**. Fuente: Material de la asignatura "Visión Artificial"

$$\sigma_{jk} = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \mu)^2}$$

Ecuación 2: Cálculo de la desviación típica de los **N** valores **x** de la característica **j** para la clase **k**. Fuente: Material de la asignatura "Visión Artificial"

A modo de ejemplo, supóngase dos clases A y B, definidas mediante tres características a, b y c. Después del paso descrito se habrán obtenido 12 parámetros para definir el clasificador. Con esto queda definido el clasificador.

Cuando se detecta un candidato que se necesita clasificar en una de las dos clases A o B en función de sus características a, b y c se debe obtener primero la probabilidad de que cada valor de cada característica pertenezca a cada clase, mediante la siguiente función, que supone una distribución normal en forma de campana de Gauss:

$$P(n_{jk}) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2} \cdot (\frac{x_j - \mu}{\sigma})^2}$$

Ecuación 3: Cálculo de la probabilidad de que el objeto **n** pertenezca a la clase **k** en base a la característica **j**. Fuente: Material de la asignatura "Visión Artificial"

Ahora el algoritmo se basa en el teorema de Bayes para calcular la probabilidad de que el objeto pertenezca a cada una de las clases, en base a las probabilidades calculadas y a las probabilidades a priori conocidas P(k) (Por ejemplo, en un saco en el que hay 5 objetos de la clase A y 5 de la clase B las probabilidades a priori serían 50%-50%).

Los porcentajes de pertenencia a cada clase vienen dados por las siguientes expresiones, utilizando el ejemplo con dos clases A y B y 3 características a, b y c:

$$P(n_A) = \frac{P(A) \cdot \prod_{j=a}^{c} P(n_{jA})}{\sum_{k=A}^{B} [P(k) \cdot \prod_{j=a}^{c} P(n_{jk})]}$$

$$P(n_B) = \frac{P(B) \cdot \prod_{j=a}^{c} P(n_{jB})}{\sum_{k=A}^{B} [P(k) \cdot \prod_{j=a}^{c} P(n_{jk})]}$$

Ecuación 4 y 5: Cálculos de las probabilidades de pertenencia del objeto **n** a las clases **A** y **B**. Fuente: Material de la asignatura "Visión Artificial"

Una vez obtenidos estos valores el algoritmo está en posición de decidir a qué clase pertenece el objeto, pues será la correspondiente al coeficiente de probabilidad más elevado. En el caso de que ambos sean parecidos significará que las características utilizadas para la decisión no son lo suficientemente excluyentes, y se deberá buscar otras más acertadas.

2. Clasificador k Nearest Neighbour (k-NN)

Este clasificador se basa en asignar a un objeto desconocido la clase a la que pertenecen sus **k** vecinos más próximos en un espacio de características. Este método presenta la ventaja de que no se requieren cálculos como en el anterior, por lo que es idóneo para aplicaciones donde la capacidad de cómputo es un aspecto a tener en cuenta.

En primer lugar se necesita un set de datos con los que calibrar el sistema, de forma que se disponga una base de datos con características de cada clase. Este enunciado se puede entender fácilmente en la siguiente imagen, en la que se representan dos clases de datos A y B según dos características a y b.

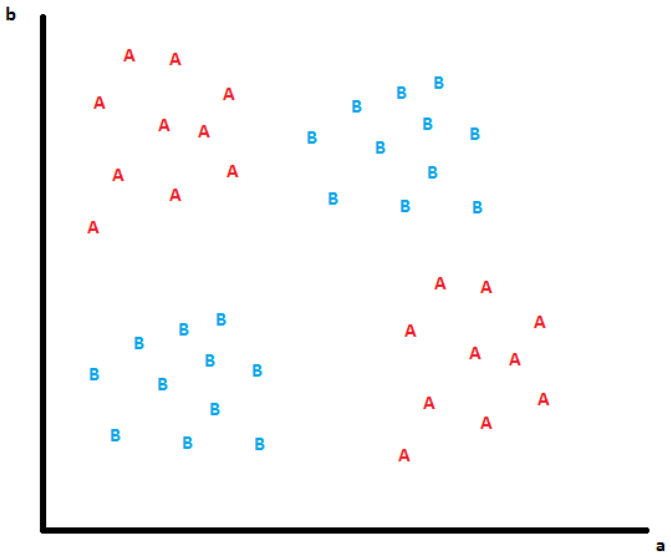

Figura 45: Banco de datos de dos clases **A** y **B** según dos características **a** y **b**. Fuente: Elaboración propia

Cuando se detecta un objeto que se desea clasificar se sitúa en la gráfica y se comprueban las clases de los **k** objetos más cercanos. Este parámetro **k** se determina heurísticamente mediante pruebas.

Una vez comprobadas las clases de estos elementos existen varias opciones:

- Asignar al objeto la clase del vecino más cercano (k=1)
- Asignar al objeto la clase de la mayoría de vecinos más cercanos (k>1)
- Ponderar la distancia de los vecinos más cercanos para determinar la clase más probable (k>1).

En base a estos criterios se determina una clase a la que asociar dicho objeto y finaliza el proceso de clasificación del algoritmo.

En la siguiente imagen se muestra un ejemplo en el que se desea clasificar un objeto "**?**" en función de dos características a y b en dos clases A y B. Se ve como utilizando un valor de k=1 la clase a la que se asociaría el objeto sería la clase B (círculo amarillo), mientras que si k=3 la clase asociada sería la A (circulo rosa).

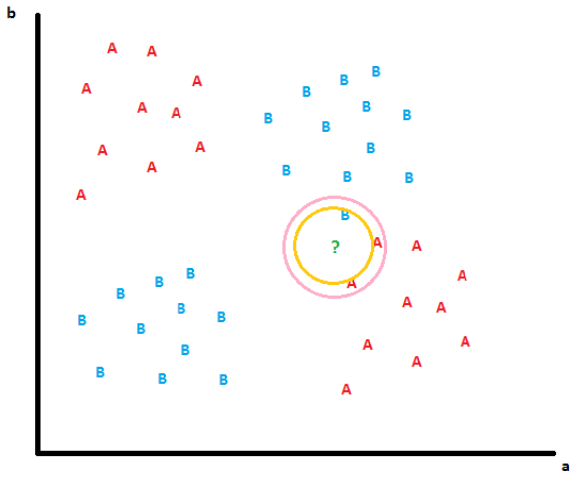

Figura 46: Ejemplo de clasificación mediante el método k-NN. Fuente: Elaboración propia

Al igual que con la implementación de la primera parte de este apartado correspondiente a la obtención del descriptor basado en HOG, la implementación del clasificador se ha realizado mediante las funciones propias de OpenCV, una de las cuales es el ya descrito clasificador k-NN, y que se ha utilizado en el presente proyecto. Al igual que en la explicación del programa de obtención del descriptor HOG, el desarrollo del programa en el que se opera con el clasificador tiene una extensión elevada, por lo que se van a mostrar las funciones principales relativas a dicho clasificador.

Para empezar es necesario importar la librería que contiene la clase del clasificador, denominada *KNearest*, y una auxiliar necesaria llamada *StatModel* mediante la siguiente instrucción al principio del programa:

```
import org.opencv.ml.KNearest;
import org.opencv.ml.StatModel;
```

Figura 47: Importación de la librería que contiene las clases KNearest y StatModel. Fuente: Elaboración propia

Ahora debe crearse un objeto llamado *model* de tipo k-NN que se calibrará con los datos obtenidos de un set de aprendizaje llamado *dataSet* y sus etiquetas de clase correspondientes en *labels*. Una vez realizada esta operación, se dispondrá de un clasificador calibrado.

```
StatModel model = KNearest.create();
model.train(dataSet, Ml.ROW_SAMPLE, labels);
```

Figura 48: Creación y entrenamiento del clasificador k-NN. Fuente: Elaboración propia

Una vez con el clasificador preparado, se le puede pasar un nuevo dato en la variable *testData* para clasificar. También es necesario pasar como argumento a la función el número de vecinos más cercanos a considerar (k) y la variable donde guardar el resultado (resultados). La línea de código correspondiente a esta operación es la siguiente:

```
((KNearest) model).findNearest(testData, k, resultados);
```

Figura 49: Clasificación de un nuevo objeto mediante el clasificador k-NN. Fuente: Elaboración propia

2.4.6 Resultados Histograma de Gradientes y clasificador k-NN

En base a los métodos descritos de detección y clasificación se efectuó una serie de pruebas con un banco de imágenes de huevos de helmintos y artefactos arrojando resultados muy satisfactorios.

En estos ensayos se varió el parámetro k que determina la cantidad de vecinos próximos que influyen en la clasificación del elemento, llegando a la conclusión de que en la gran mayoría de los casos el porcentaje de acierto aumentaba conforme se disminuía este parámetro k. Esto es debido al gran parecido de algunos helmintos y a que 6 clases en las que clasificar los candidatos es un número bastante elevado. Se ha añadido además una 7º clase correspondiente a imágenes de huevos del género Taenia que se encontraban fragmentados.

A continuación se muestran los resultados obtenidos con un grupo de entrenamiento de 2826 muestras y un grupo de validación de 844, siendo la configuración óptima la correspondiente a k=1. En ellos se muestra la matriz de confusión, organizada de forma que las filas representan las clases conocidas de los huevos, y las columnas las clases en las que han sido clasificadas. También se muestra debajo de la matriz de resultados el porcentaje de acierto y el número de falsos positivos (FP) y falsos negativos (FN).

Cabe recordar el requisito indispensable de que bajo ningún concepto está permitido clasificar un huevo como un artefacto, siendo viable el caso contrario y también el clasificar el huevo de una clase como otra clase, o lo que es lo mismo, que la primera columna, que es la que representa los candidatos clasificados como artefactos, debe estar constituida por 0 salvo la primera fila que se corresponde con los artefactos reales.

En base a este criterio, y a que los resultados presentados cumplen esta condición, se puede concluir que el sistema cumple con las especificaciones requeridas por el cliente.

Matriz de Confusión							
k=1	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	297	0	0	0	0	1	0
Áscaris	0	140	0	0	0	0	0
Hymenolepis	0	0	9	0	0	0	0
Schistosoma	0	0	0	85	0	0	0
Taenia	0	0	0	0	135	0	0
Taenia Rotos	0	0	0	0	1	123	0
Trichuris	0	0	0	0	1	0	54
Porcentaje de acierto			99,65%	FP	1	FN	0

Tabla 1: Resultados k=1. Fuente: Elaboración Propia

Matriz de Confusión							
k=3	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	296	0	0	0	0	1	1
Áscaris	1	139	0	0	0	0	0
Hymenolepis	0	0	9	0	0	0	0
Schistosoma	0	1	0	84	0	0	0
Taenia	0	1	0	0	134	0	0
Taenia Rotos	0	0	0	0	5	119	0
Trichuris	0	0	0	0	1	0	54
Porcentaje de acierto			98,70%	FP	2	FN	1

Tabla 2: Resultados k=3. Fuente: Elaboración Propia

Matriz de Confusión							
k=5	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	297	0	0	1	0	0	0
Áscaris	1	139	0	0	0	0	0
Hymenolepis	3	0	6	0	0	0	0
Schistosoma	0	0	0	85	0	0	0
Taenia	0	0	0	0	133	2	0
Taenia Rotos	0	0	0	0	7	117	0
Trichuris	4	0	0	0	2	0	49
Porcentaje de acierto			97,64%	FP	1	FN	8

Tabla 3: Resultados k=5. Fuente: Elaboración Propia

Matriz de Confusión							
k=7	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	297	0	0	1	0	0	0
Áscaris	4	136	0	0	0	0	0
Hymenolepis	5	0	4	0	0	0	0
Schistosoma	0	0	0	85	0	0	0
Taenia	1	4	0	0	125	5	0
Taenia Rotos	0	0	0	0	14	110	0
Trichuris	8	0	0	0	4	0	43
Porcentaje de acierto			94,56%	FP	1	FN	18

Tabla 4: Resultados k=7. Fuente: Elaboración Propia

Matriz de Confusión							
k=9	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	297	0	0	1	0	0	0
Áscaris	4	136	0	0	0	0	0
Hymenolepis	8	0	1	0	0	0	0
Schistosoma	2	0	0	83	0	0	0
Taenia	2	4	0	0	120	9	0
Taenia Rotos	0	0	0	0	21	103	0
Trichuris	9	0	0	0	12	0	34
Porcentaje de acierto			91,49%	FP	1	FN	25

Tabla 5: Resultados k=9. Fuente: Elaboración Propia

Matriz de Confusión							
k=11	Artefactos	Áscaris	Hymenolepis	Schistosoma	Taenia	Taenia Rotos	Trichuris
Artefactos	297	0	0	1	0	0	0
Áscaris	4	136	0	0	0	0	0
Hymenolepis	9	0	0	0	0	0	0
Schistosoma	5	0	0	80	0	0	0
Taenia	2	6	0	0	118	9	0
Taenia Rotos	1	3	0	0	23	97	0
Trichuris	9	0	0	0	15	0	31
Porcentaje de acierto			89,72%	FP	1	FN	30

Tabla 6: Resultados k=11. Fuente: Elaboración Propia

3 Conclusión y trabajo futuro

Al comienzo de este documento se citaron varios trabajos relativos a la implementación de sistemas automáticos de detección y clasificación de helmintos con porcentajes de acierto de hasta el 97,70% (Avci & Varol, 2009). Estos porcentajes tan elevados daban un indicio de la viabilidad de conseguir resultados aceptables con este tipo de sistemas de procesado de imagen, pero el porcentaje obtenido de 99,65% cumpliendo el requisito de no descartar ningún huevo como artefacto ha sido mayor de lo esperado.

Esto se ha debido principalmente a dos razones: Por un lado la elevada resolución de la cámara permitía un tratamiento más preciso de la imagen a costa de un mayor coste computacional. Por otro lado, la relativamente pequeña cantidad de imágenes suministradas por parte de la empresa obligaron a la generación de muestras artificiales basadas en escalados y giros de las imágenes disponibles que ocasionaron que alguna de las imágenes del set de entrenamiento apareciese con una ligera transformación en el set de validación. Pese a este hecho, ante un banco de muestras con un volumen más elevado y siendo todas distintas cabe esperar un porcentaje de acierto cercano al obtenido.

En cuanto al posible trabajo futuro se podría continuar por la línea referente a optimizar el proceso probando otros tipos de clasificadores o características en que basar el sistema de detección y clasificación, de forma que se consigan porcentajes de acierto mayores con sets de huevos más diversos. Otra línea de trabajo viable sería la relativa a aumentar las prestaciones del sistema consiguiendo que fuese capaz de detectar parámetros del agua en función de las partículas que aparecen en las imágenes, como cantidad de minerales, pH, salinidad etc.

Cabe mencionar como segundo objetivo cumplido con el presente proyecto la justificación de los conocimientos aprendidos durante el Máster Universitario en Ingeniería Industrial – Especialidad en Control, Automática y Robótica Industrial, y más concretamente los referentes a los contenidos de Visión Artificial, siendo esta una disciplina todavía poco común que presenta una gran cantidad de aplicaciones y buenos resultados en prácticamente todos los sectores tanto industriales como domésticos.

4 Referencias bibliográficas

Avci, D., & Varol, A. (2009). An expert diagnosis system for classification of human parasite eggs based on multi-class SVM. *ELSEVIER 36* , 43-48.

Castañon, C. (2007). Análise e reconhecimento digital de formas biológicas para o diagnóstico automático de parasitas do gênero Eimeria. *PhD thesis* .

H. C. K. M.-H. C. Yoon Seok Yang, D. K.-Y. (2001). Automatic identification of human helminth eggs on microscopic fecal specimens using digital image processiong and an artificial neural network. *IEEE* .

Koino, S. (1922). Experimental infections on human body with ascarides. *Japan Medical World, 15* , 317-320.

Stewart, F. (1916). On the life-history of Ascaris lumbricoides. *British medical journal* .

Presupuesto

En este documento se valorará y contabilizará el coste monetario asociado al proyecto. Este coste estará basado en el importe relativo a la mano de obra del autor y al coste de amortización de los programas y equipos utilizados. Se obtendrán estos presupuestos parciales para luego generar el presupuesto de ejecución material y terminar con el presupuesto de desarrollo.

1 Presupuestos parciales

1.1 Presupuesto parcial de mano de obra

Como coste de mano de obra se contabilizará la del ingeniero que desarrolla el proyecto y la del técnico de laboratorio encargado de tomar imágenes de las muestras de huevos. El sueldo medio de un ingeniero industrial se estima en 30 €/h y el de un técnico de laboratorio en 10 €/h. Se estima también una parte de mano de obra indirecta (5%) que incluye la dedicación del supervisor así como las consultas realizadas a personal ajeno al proyecto.

Presupuesto parcial de mano de obra

Concepto	Unidad	Precio unitario	Cantidad	Importe
Ingeniero Industrial	Horas	30€/h	300 h	9.000 €
Técnico de laboratorio	Horas	10 €/h	6 h	60 €
Mano de obra indirecta (5%)				453 €
TOTAL				**9.513 €**

1.2 Presupuesto parcial de amortización de los equipos y programas

Presupuesto parcial de amortización de equipos

Concepto	Precio	Periodo de amortización	Periodo utilizado	Importe
Ordenador Dell i7	800 €	5 años	3 meses	40 €
MATLAB 2015	5.000 €	5 años	3 meses	250 €
Microsoft Office 365	79 €	4 años	3 semanas	1,14 €
Microscopio OPTIKA	2.929,29 €	10 años	2 meses	48,82 €
Cámara MOTIC	882 €	10 años	2 meses	14,7 €
Costes indirectos (10%)				35,47 €
TOTAL				**390,13 €**

2 Presupuesto total

Presupuesto total	
Concepto	Importe
Presupuesto parcial de mano de obra	9.513,00 €
Presupuesto parcial de amortización de equipos	390,13 €
Presupuesto de ejecución material	**9.903,13 €**
Gastos generales (5%)	495,16 €
Beneficio industrial (6%)	594,19 €
Presupuesto total de ejecución	**10.992,47 €**
I.V.A. (21%)	2.308,42 €
Presupuesto base de licitación	**13,300.89 €**