INSTITUTO TECNOLÓGICO DE CHIHUAHUA
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

“SISTEMA DE VISIÓN EMBEBIDO PARA LA LOCALIZACIÓN DE DEFECTOS EN LA SUPERFICIE DE GEL DE ELECTRODOS DE DISPERSIÓN”

TESIS

QUE PARA OBTENER EL GRADO DE

MAESTRO EN INGENIERÍA MECATRÓNICA

PRESENTA:

ING. JOSÉ ALBERTO ARZAGA FLORES

DIRECTOR DE LA TESIS:
DR. ISIDRO ROBLEDO VEGA

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

CHIHUAHUA, CHIH., JUNIO DE 2019

PAGINA PARA COPIA DE OFICIO DE AUTORIZACIÓN DE IMPRESIÓN

PAGINA PARA CARTA DE APROBACIÓN DEL COMITÉ REVISOR DE LA TESIS

DEDICATORIA

Dedico este trabajo a mis padres, porque sin ellos nada de esto sería posible, porque a ellos les debo todo, y por ser siempre inspiración y ejemplo a seguir.

A Adriana, por su apoyo incondicional, por estar siempre ahí, y porque el tiempo y esfuerzo dedicados a este trabajo no eran solo míos, gracias.

A mi director de tesis, Dr. Isidro, le agradezco por todo su apoyo, su paciencia, su guía y la motivación para concluir este proyecto.

José Arzaga

SISTEMA EMBEBIDO DE VISIÓN PARA LA LOCALIZACIÓN DE DEFECTOS EN LA SUPERFICIE DE GEL DE ELECTRODOS DE DISPERSIÓN

José Alberto Arzaga Flores
Maestro en Ingeniería Mecatrónica
División de Estudios de Posgrado e Investigación del
Instituto Tecnológico de Chihuahua
Chihuahua, Chih. 2019
Director de Tesis: Dr. Isidro Robledo Vega

RESUMEN

La complicación más común de una electrocirugía es la quemadura eléctrica, que puede derivarse de defectos en la superficie del gel de los electrodos de dispersión. Por su naturaleza, estos defectos son difíciles de detectar, ya sea durante el proceso de manufactura o previo a su uso, debido a que resulta complejo distinguirlos a simple vista y, por consecuencia las inspecciones visuales tradicionales no son efectivas. En el presente proyecto de tesis, se desarrolló un sistema embebido de visión capaz de detectar estos defectos. En este proyecto se aborda el proceso de desarrollo del sistema, desde sus aspectos más simples hasta los más complejos, abarcando desde la selección de los componentes físicos del sistema, tales como cámara, lente, iluminación y tarjeta embebida, pasando por la configuración del entorno de desarrollo, configuración de la tarjeta embebida, instalación del ambiente de programación e instalación de las librerías para visión por computadora, llegando finalmente hasta el desarrollo de los algoritmos de detección de defectos y los detalles de cada una de las etapas de procesamiento que lo componen. Factores como la textura del gel, su transparencia, la superficie irregular del mismo, el tamaño reducido de los defectos, así como la variedad de formas que pueden presentar y el brillo del aluminio en el fondo del gel, son algunos de los retos que se debieron enfrentar para desarrollar el sistema, cuyo índice de detección fue de 93.8%. Finalmente se presentan algunas conclusiones y problemas encontrados durante la elaboración del presente trabajo.

EMBEDDED VISION SYSTEM TO LOCATE DEFECTS ON GEL'S SURFACE OF DISPERSION ELECTRODES

José Alberto Arzaga Flores
Master of Mechatronics Engineering
División de Estudios de Posgrado e Investigación del
Instituto Tecnológico de Chihuahua
Chihuahua, Chih. 2019
Thesis Director: Dr. Isidro Robledo Vega

ABSTRACT

The most common complication during an electrosurgery is the electric burn, it can derivate from defects on gel's surface of dispersion electrodes. By its nature, those defects are difficult to detect, either during the manufacturing process, or prior its use. This is because it's complex to distinguish them at fore sight. Therefore, traditional visual inspections are not effective. In this thesis project, an embedded vision system capable of detecting those defects was developed. This project addresses the process of system development, from the simplest to the most complex aspects, ranging from the selection of physical components of the system, such as camera, lens, lighting and embedded board, through the configuration of the development environment, configuration of the embedded board, installation of the programming environment and installation of the libraries for computer vision, and finally, the development of the algorithms for defects detection, and the details of each of the processing stages that are part of it. Factors such as the gel texture, transparency, the uneven surface; the small size of the defects, as well as the variety of their shapes, and the brightness of the aluminum covered by the gel are some of the challenges that were faced during the development of the system, which has a detection rate of 93.8%. Finally, some conclusions and problems encountered during the elaboration of this work are presented.

ÍNDICE

Listado de Figuras ix
Listado de Tablas xiii
I. Antecedentes 1
II. Marco Teórico 6
  2.1 Fundamentos y Conceptos Básicos 6
    2.1.1 Visión de Máquina 6
    2.1.2 Sistema de Visión 7
    2.1.3 Sistema de visión embebido 9
    2.1.4 Iluminación 11
    2.1.5 Campo de visión 15
  2.2 Software de Desarrollo 16
    2.2.1 Python 17
    2.2.2 OpenCV 19
  2.3 Estado del Arte 20
    2.3.1 Métodos de umbralizado 21
    2.3.2 MSER y plantilla 24
    2.3.3 Luz estructurada 26
    2.3.4 Modelos estadísticos 28
III. Caracterización de Defectos en Electrodos de Dispersión 32
  3.1 Defecto de De-laminación 33
  3.2 Defecto de Ojo de Pescado 33
  3.3 Defecto de Rasgado 34
  3.4 Defecto de Aluminio Expuesto 35
  3.5 Defecto de Burbuja 35
  3.6 Defecto de Contaminación 36
IV. Sistema de adquisición de imágenes 37
  4.1 Cámara 37
  4.2 Lente 39
  4.3 Iluminación 44

4.4 Tarjeta Embebida 52
4.4.1 Configuración del sistema operativo 55
4.4.2 Configuración de OpenCV 63
4.4.3 Configuración del procesador 68
4.4.4 Configuración de Hardware 71
V. Software de detección de defectos 74
5.1 Estructura de Archivos 74
5.2 Configuraciones para la Ejecución del Software 76
5.3 Flujo del Programa 79
5.3.1 Segmentación 82
5.3.2 Detección de Defectos en Bordes 89
5.3.3 Detección de Defectos en Superficie 94
VI. Resultados 101
6.1 Resultados de la Etapa de Segmentación 102
6.2 Resultados la Etapa de Detección de Defectos en Bordes 103
6.3 Resultados de la Etapa de Detección de Defectos en Superficie 105
6.4 Resultados Generales 106
VII. Conclusiones 109
VIII. Referencias Bibliográficas 113
Anexo 1. Archivo main.py 116
Anexo 2. Archivo segmentation.py 125
Anexo 3. Archivo innerDefects.py 137
Anexo 4. Archivo outterDefects.py 147
Anexo 5. Archivo common.py 154
Anexo 6. Descripción de muestras 161

LISTADO DE FIGURAS

Figura 1.1 - Diagrama a bloques del sistema de visión embebido. 5
Figura 2.1 - Ejemplo de aplicación de visión por computadora. 7
Figura 2.2 – Esquema de un sistema de visión típico. 8
Figura 2.3 – Cámara, tarjeta embebida, microcontrolador para procesamiento. 10
Figura 2.4 – Sistema de visión embebido utilizando una tarjeta de desarrollo de la compañía Freescale. 11
Figura 2.5 – Elementos involucrados en la iluminación: objeto, fuente de luz y cámara. 12
Figura 2.6 – Reflejo de luz en superficies suaves (izquierda) y rugosas (derecha). 13
Figura 2.7 – Técnicas de iluminación: a) iluminación de domo difuso, b) difuso en eje, c) campo brillante, d) campo oscuro, e) de fondo difuso y f) de fondo colimada. 15
Figura 2.8 – A. Campo de visión vertical. B. Angulo de visión. C. Distancia de trabajo. 16
Figura 3.1- Electrodo de dispersión y su corte transversal. 32
Figura 3.2 – Ejemplos del defecto de de-laminación. 33
Figura 3.3 – Ejemplos de defecto de ojo de pescado. 34
Figura 3.4 – Ejemplo de defecto de rasgado. 34
Figura 3.5 – Ejemplo de defecto de aluminio expuesto. 35
Figura 3.6 – Ejemplo de defecto de burbuja. 36
Figura 3.7 – Ejemplo de defecto de contaminación. 36
Figura 4.1 – De izquierda a derecha, cámara Unibrain Fire-i 785b, FLIR CMLN-13S2M-CS, Basler puA2500-14cm. 37
Figura 4.2 – Imágenes obtenidas a una distancia de trabajo de 60 cm con lentes de a) 75 mm, b) 50 mm, c) 35 mm, d) 25 mm y e) 18 mm. 40
Figura 4.3 – Izquierda, lente #54-689. Derecha, lente M12B3525M12. 43
Figura 4.4 – Izquierda, imágenes obtenidas con la lente #54-689. Derecha, imágenes obtenidas con la lente M12B3525M12. 43
Figura 4.5 – Iluminación de anillo, imagen adquirida con iluminación de campo brillante. 44
Figura 4.6 – De izquierda a derecha, patrones de iluminación utilizados, montaje de proyector y cámara, proyección de patrón sobre electrodo de dispersión. 45
Figura 4.7 – Imágenes adquiridas con diferentes patrones de iluminación. 45
Figura 4.8 – Iluminación difusa, lámpara de luz difusa e imagen adquirida con iluminación difusa. 46

Figura 4.9 – Áreas a resaltar con cada tipo de iluminación. 47
Figura 4.10 – Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo. 47
Figura 4.11 – Iluminación de campo oscuro a distancias cortas. La zona A aparece más iluminada para la cámara que la zona B. 48
Figura 4.12 – Imagen capturada utilizando Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo. 49
Figura 4.13 – Iluminación de campo oscuro, dos fuentes de luz simétricas a 35 cm y utilizando luz de fondo. 49
Figura 4.14 – Imagen capturada utilizando Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo. 50
Figura 4.15 – Efecto de variar la distancia de la fuente de luz en una escena. 51
Figura 4.16 – Imagen capturada utilizando iluminación de campo oscuro con una sola fuente de luz a 80 cm y utilizando luz de fondo. 52
Figura 4.17 – Imagen comparativa de las tarjetas Jetson TX1 y Jetson TX2. 54
Figura 4.18 – Tarjetas Jetson TX2. 55
Figura 4.19 – Interfaz de instalación de JetPack-3.3. Pantalla 1. 56
Figura 4.20 – Interfaz de instalación de JetPack-3.3. Pantalla 2. 57
Figura 4.21 – Interfaz de instalación de JetPack-3.3. Pantalla 3. 57
Figura 4.22 – Interfaz de instalación de JetPack-3.3. Pantalla 4. 58
Figura 4.23 – Interfaz de instalación de JetPack-3.3. Pantalla 5. 58
Figura 4.24 – Interfaz de instalación de JetPack-3.3. Pantalla 6. 59
Figura 4.25 – Interfaz de instalación de JetPack-3.3. Pantalla 7. 59
Figura 4.26 – Interfaz de instalación de JetPack-3.3. Pantalla 8. 60
Figura 4.27 – Interfaz de instalación de JetPack-3.3. Pantalla 9. 61
Figura 4.28 – Interfaz de instalación de JetPack-3.3. Pantalla 10. 61
Figura 4.29 – Interfaz de instalación de JetPack-3.3. Pantalla 11. 62
Figura 4.30 – Interfaz de instalación de JetPack-3.3. Pantalla 12. 62
Figura 4.31 – Versión de Python y OpenCV instalada en la tarjeta Jetson TX2. 68
Figura 4.32 – Utilización de procesadores en tarjeta Jetson TX2, configuración por defecto. 69
Figura 4.33 – Utilización de procesadores en tarjeta Jetson TX2, todos los nucleas activos. 70
Figura 4.34 – Utilización de procesadores en tarjeta Jetson TX2, todos los nucleas activos, máxima frecuencia. 70

Figura 4.35 – Conexiones en la tarjeta Jetson TX2. 71
Figura 4.36 – Diagrama a bloques de las conexiones en la tarjeta Jetson TX2. 73
Figura 5.1 – Diagrama de flujo de la función main( ). 80
Figura 5.2 – Programa en ejecución. Izquierda: sin utilizar cámara. Derecha: utilizando cámara.81
Figura 5.4 – Imagen de Electrodo, vista en cada canal de los espacios RGB y HSV. 84
Figura 5.5 – Primera fila: Filtros de suavizado aplicados sobre imgA. Segunda fila: filtros de suavizado aplicados sobre imgH. 85
Figura 5.6 – Histograma de imgAFiltrada 86
Figura 5.7 – Imagen imgAFiltrada binarizada utilizando distintos métodos de umbralizado. 86
Figura 5.8 – Primera fila: Imagen binA después de operaciones morfológicas. Segunda fila: Transformada de distancia de binA, umbralizado de transformada de distancia y mapa de colores de áreas. 87
Figura 5.9 – Primera fila: máscaras utilizadas para segmentación. Segunda fila: Imagen segmentada. 89
Figura 5.10 –Diagrama de flujo de la función EdgeDefects ( ). 90
Figura 5.11 –Filtros de suavizado aplicados a la imagen de entrada en la función EdgeDefects ( ). 91
Figura 5.12 –Detectores de bordes aplicados sobre imagen suavizada. 92
Figura 5.13 – Máscara canny después del proceso de filtrado de bordes y dilatación. 92
Figura 5.14 – Resultado del proceso de detección de defectos en el borde del electrodo. 94
Figura 5.15 –Diagrama de flujo de la función SurfaceDefects( ). 95
Figura 5.16 –Izquierda: Imagen de entrada con la ROI marcada. Derecha: ROI. 96
Figura 5.17 –Filtros de suavizado aplicados en la ROI. 97
Figura 5.18 – Histograma de imagen suavizada. 98
Figura 5.19 – Binarización de imagen suavizada. 98
Figura 5.20 – Detectores de bordes aplicados sobre imagen suavizada. 99
Figura 5.21 – Combinación de la máscara por umbral y máscara de bordes. 99
Figura 5.22 – Imagen de entrada e Imagen de salida con localización de defectos. 100
Figura 6.1 – Grafica de los resultados obtenidos en la etapa 1 de procesamiento. 103
Figura 6.2 – Grafica de los resultados obtenidos en la etapa 2 de procesamiento. 103
Figura 6.3 – Grafica de los resultados obtenidos en la etapa 2 de procesamiento separados por tipo de muestra. 104

Figura 6.4 – Gráfica de los resultados obtenidos en la etapa 3 de procesamiento. 105
Figura 6.5 – Grafica de los resultados obtenidos en la etapa 3 de procesamiento, separados por tipo de muestra. 106
Figura 6.6 – Grafica de los resultados generales obtenidos por el algoritmo de detección. 107
Figura 6.7 – Gráfica de los resultados generales obtenidos por el algoritmo de detección, separados por tipo de muestra. 107
Figura 7.1 – Borde de gel por dentro del borde de aluminio. 109
Figura 7.2 – Borde de gel por fuera del borde de aluminio. 110
Figura 7.3 – Efecto de un electrodo sin sujetar correctamente. 111

LISTADO DE TABLAS

Tabla 4.1 – Comparación de características de las cámaras evaluadas. 38

Tabla 4.2 – Comparación de lentas y sus distancias de trabajo. 42

Tabla 4.3 – Entornos de desarrollo utilizados. 64

Tabla 6.1 – Cuatro posibles resultados del proceso de detección. 101

Tabla A6.1 – Listado de muestras, descripción de sus defectos, y resultado de su procesamiento. 161

I. ANTECEDENTES

La electrocirugía es un procedimiento médico que utiliza corrientes eléctricas oscilantes de alta frecuencia y puede aplicarse a destruir lesiones benignas y malignas, controlar el sangrado, cortar y/o extirpar el tejido [1]. Para lograr esto, se utiliza el principio de diatermia (calor inducido por electricidad) de modo que el tejido se calienta mientras que el dispositivo emisor se mantiene frio. Los elementos generales de los que se compone una unidad de electrocirugía son:

1. Generador electro quirúrgico (ESU): Es la fuente de energía utilizada en la operación.
2. Electrodo activo: Es un elemento conductor conectado al ESU que direcciona la corriente generada hacia un área muy pequeña en el cuerpo del paciente.
3. Paciente: La corriente entra al cuerpo del paciente a través del electrodo activo y circula hasta el electrodo de retorno.
4. Electrodo de retorno: Es un elemento conductor conectado al ESU, que permite el retorno de la corriente desde el cuerpo del paciente hasta el generador.

Existen dos tipos de electrocirugía, la bipolar y la monopolar. En la configuración bipolar ambos electrodos se encuentran juntos (a una distancia de 1mm a 3mm), al activar la corriente esta fluye de un electrodo a otro limitando el área de lesión; mientras que en la configuración monopolar, la corriente entra por el electrodo activo, fluye a través del cuerpo, y regresa al ESU a través del electrodo de retorno [2]. En este último caso, la corriente entra el cuerpo a través de un área muy pequeña en el electrodo activo. La función del electrodo de retorno es sacar esa corriente del cuerpo sin producir el efecto de corte del elemento activo, para lograr esto, el electrodo de retorno es generalemnte de una superficie amplia y de muy baja impedancia, ya que su función es dispersar la corriente de salida a través de toda el área de contacto. Estos electrodos son llamados electrodos de dispersión.

Existen algunas complicaciones que pueden derivarse de una electrocirugia, la mas común es quemadura eléctrica. Si el electrodo de dispersión, no tiene una buena superficie de contacto con el paciente, pueden provocarse arcos eléctricos entre la superficie expuesta del electrodo y

la piel, si bien estos arcos no provocarían un choque eléctrico, si generarían severas quemaduras [2]. Para evitar estas situaciones muchos fabricantes de parches de dispersión utilizan geles conductivos que aseguren la adherencia del electrodo a la piel. El problema radica en que los procesos de manufactura de dichos electrodos no son capaces de garantizar que toda la superficie del electrodo este cubierta con gel conductivo. De modo que se pueden presentar zonas de metal expuesto. Estas zonas expuestas representan un riesgo de arcos eléctricos. Existen otro tipo de defectos como son irregularidades en la superficie del gel (adelgazamientos o discontinuidades), que si bien no exponen la superficie metálica del electrodo, son puntos suseptibles a generar arcos eléctricos. Esta problemática puede ser abordada desde el enfoque de Inspección Visual Automatizada, también conocida como Visión de Máquina.

Dentro del ambiente industrial, el termino “Visión de Máquina”, se refiere a “el uso de dispositivos ópticos de sensado sin contacto para recibir e interpretar automáticamente una escena real con el fin de obtener información y controlar máquinas y/o procesos” [3]. Cada vez es mas común encontrar este tipo de sistemas en las industrias ya que por su versatilidad pueden ser utilizados para verificar la presencia de componentes, revisar texto y códigos, medir dimensiones, realizar alineaminentos y localizar diversos tipos de defectos o patrones. Históricamente los controles de calidad se realizaban seleccionando muestras aleatorias de la línea de producción para realizar inspecciones manuales, luego mediante análisis estadísticos los resultados se extrapolaban al resto de la producción. Este enfoque deja cabida a defectos sin detectar, que podían representar fallas en procesos subsequentes o embarques de material defectuoso a los clientes. Por otro lado, la visión de máquina puede asegurar hasta un 100% de efectividad en las inspecciones [4].

La visión de máquina en la industria esta evolucionando a sistemas cada vez mas pequeños y económicos, además, se espera de ellos un mejor desempeño. Por esta razón, cada vez con mayor frecuencia, dichos sistemas se basan en plataformas embebidas, plataformas compactas para uso específico, pequeñas y flexibles, en muchas ocaciones con hardware dedicado para adquisición de imágenes que otorgan como resultado sistemas de visión compactos y relativamente baratos, capaces de adaptarse a las limitaciones que puede tener el ambiente industrial (espacio, consumo de energía, escalabilidad, ruido, etc.) [5]. En este proyecto de tesis

se aprovecharon las ventajas que ofrecen los sistemas de visión embebidos para mejorar el proceso de inspección de la superficie de gel en los electrodos de dispersión. Existen muchos ejemplos de sistemas de visión de máquina que han sido diseñados para realizar inspecciones y caracterizaciones especificas, tal es el caso de [6], y si bien, existen trabajos previos para detectar defectos en superficies, que se asemejan a la que se abordó en esta tesis, como la caracterización de defectos en cuerpos transparentes mediante distintas condiciones de iluminación que plantea [7] o la inspección en tiempo real de defectos en superficies especulares propuesta por [8], no existen trabajos previos, concretamente sobre superficies metálicas reflectivas con recubrimientos semi-transparentes texturizados y no uniformes. La contribución de este trabajo comprende la caracterización de los defectos en este tipo de superficie y la propuesta de algoritmos para su detección utilizando un sistema de visión embebido.

En la electrocirugía, los electrodos de dispersión utilizados en condiciones ideales no representan riesgo alguno para el paciente ni para el cirujano, pero estas condiciones ideales no siempre pueden ser alcanzadas en la práctica. Diversos factores en la aplicación real de la electrocirugía pueden representar riesgos. Algunos de los más peligrosos son las quemaduras accidentales que se producen por problemas en el electrodo de dispersión, estos pueden ir desde: la aplicación sobre una superficie o tejido no adecuado, uso de producto con fecha de caducidad vencida y, por ende, comportamiento anormal, mala superficie de contacto dejando huecos entre la piel y la superficie conductora y defectos en la manufactura de los electrodos que pueden generar arcos eléctricos [9]. La mayoría de estos puntos son dependientes del personal encargado de realizar la preparación para la electrocirugía y pueden ser evitados asegurando que el personal encargado de dicha preparación tenga el entrenamiento adecuado. El último punto es más difícil de controlar ya que por la naturaleza de los defectos de manufactura es difícil garantizar que puedan ser encontrados al momento de realizar el procedimiento quirúrgico.

Los procesos de manufactura de los electrodos de dispersión son procesos de ensamble que implican diversas etapas, en cada una de ellas se trabaja con alguno de los componentes que conforman el parche. La primera etapa corresponde a la fabricación de la placa de aluminio recubierta en su cara frontal con gel conductor adhesivo. La segunda etapa consiste en la unión de la placa de aluminio, en su cara posterior, al material o papel adhesivo que sujetará el parche

al cuerpo del paciente. La última etapa implica la unión de un papel encerado a la superficie adhesiva del parche que sirve como protección. Este papel encerado se remueve una vez que el parche será utilizado dejando expuestas las superficies adhesivas, tanto del papel como del gel, para pegarlas al cuerpo del paciente. La superficie de gel en ocasiones no cubre la totalidad de la placa de aluminio, o la cubre, pero no de manera uniforme. Esto genera defectos de manufactura que pueden ocasionar graves quemaduras cuando el parche es utilizado. La forma en que se intenta contener el envío de material defectuoso a los clientes es mediante inspecciones visuales realizadas por operadores que revisan cada parche producido. Un gran inconveniente de las inspecciones visuales es que su eficiencia típicamente es del 80% [10], lo que representa un gran riesgo considerando la severidad de los daños que pueden causar al usuario. Por esta causa, surge la necesidad de contar en la industria con un método más robusto para la identificación de los defectos de manufactura durante el proceso de ensamble. En este proyecto de tesis se ha diseñado y construido un sistema de visión de máquina embebido que pueda inspeccionar de forma más confiable la superficie de gel de los electrodos de dispersión.

El proceso de manufactura de los electrodos de dispersión es en su mayoría manual. Como todo proceso de manufactura es susceptible a fallas y generación de defectos. Entre estos defectos se encuentran los relacionados con la superficie de gel adhesivo que recubre los electrodos. Esta superficie, es irregular, semi-transparente, con una textura fibrosa. Todos esos factores combinados hacen que las inspecciones visuales humanas sean muy poco eficientes para detectar defectos sobre la misma, ya que resulta muy complicado poder distinguir anormalidades y, más aún, si se trata de anormalidades pequeñas.

Si bien las superficies reflejantes representan un desafío para los sistemas de visión debido a reflexiones que pueden disimular defectos reales, el sistema de visión embebido desarrollado en este proyecto es capaz de identificar de forma efectiva los defectos que para un humano resulta imposible. La figura 1.1 muestra la estructura del sistema de visión embebido para la localización de defectos en la superficie de gel de los electrodos de dispersión.

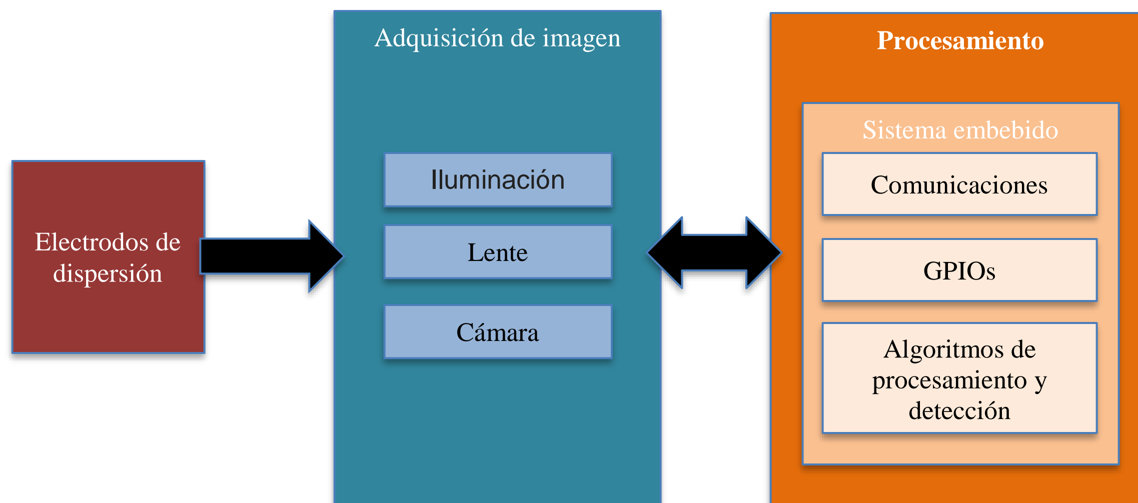

Figura 1.1 - Diagrama a bloques del sistema de visión embebido.

El punto de partida fue la identificación de los posibles defectos en los electrodos de dispersión, a partir de ahí se seleccionaron los elementos de hardware necesarios para la implementación de la etapa de adquisición de imágenes y finalmente se seleccionó un sistema embebido adecuado para dicho hardware.

El presente documento de tesis se organiza de la siguiente manera, en el Capítulo 2 se presenta el Marco Teórico que describe los conceptos técnicos que permitieron el desarrollo de este proyecto de tesis. En el Capítulo 3 se presenta la clasificación de los defectos sobre los que se enfocó el presente trabajo, así como las características de cada uno. En el Capítulo 4, se aborda el sistema de visión desarrollado, desde el enfoque de los elementos de hardware que lo conforman. El Capítulo 5 se enfoca en el algoritmo de detección de defectos desarrollado, mientras que en el Capítulo 6 y 7 podemos encontrar los resultados obtenidos y las conclusiones finales respectivamente y al final se encuentran algunos anexos que contienen el código fuente del proyecto.

II. MARCO TEÓRICO

2.1 Fundamentos y Conceptos Básicos

2.1.1 Visión de Máquina

Para definir la visión de máquina, debemos partir del concepto de “visión por computadora”. Esta última es una ciencia que toma sus bases de disciplinas como matemáticas, física, biología, ingeniería y ciencias de la computación y, a su vez, se relaciona directamente con otros campos como el aprendizaje de máquina, procesamiento de señales, inteligencia artificial. En términos generales, visión por computadora puede entenderse como la transformación de información de una imagen o una secuencia de imágenes en una decisión o en una nueva representación de la imagen original. Los seres humanos somos seres muy visuales, todos los días tomamos grandes cantidades de información visual a través de nuestros ojos, que luego nuestro cerebro se encarga de procesar, sin siquiera ser nosotros conscientes de ello. Al ser un proceso natural y fácil para nosotros, asumimos que la visión por computadora y particularmente la interpretación de imágenes es algo sencillo, pero ciertamente no es así. El percibir una flor, un auto o un animal en una imagen es una tarea sumamente fácil para una persona, pero cuando esta misma tarea, se trata de abordar mediante una computadora se vuelve extremadamente complicada, primero si se parte del hecho de que la computadora no tiene ningún conocimiento previo que le permita asociar un objeto con un patrón determinado (cosa que nuestra experiencia nos da). Además, lo que nosotros vemos como una imagen una computadora lo ve como un arreglo de números que por sí solo no tendrá ningún significado. La visión por computadora se encarga de dar a ese arreglo de números una interpretación y convertirlos en una nueva imagen o en una decisión. Una definición más formal nos dice que la visión por computadora es la ciencia encargada de que los sistemas computacionales adquieran, procesen y analicen imágenes digitales [11]. Las computadoras son buenas calculando, pero malas razonando; una computadora será eficiente para realizar mediciones, identificar diferencias entre objetos, encontrar zonas con alto contraste, pero le resultará complicado trabajar con objetos irregulares, seguir objetos en movimiento y clasificar o razonar sobre un objeto. La figura 2.1 muestra un ejemplo típico de una aplicación de visión por computadora, en este ejemplo una cámara en un automóvil adquiere imágenes del camino y la unidad de procesamiento identifica objetos presentes en dicha imagen,

tales como automóviles, señalización y macas de la carretera. Este es solo un ejemplo, aunque existe una infinidad de campos y aplicaciones en los que se puede utilizar esta ciencia.

Figura 2.1 - Ejemplo de aplicación de visión por computadora.

Finalmente, se puede concluir que el término visión de máquina es utilizado para definir la aplicación de la visión por computadora a tareas industriales. [11]

2.1.2 Sistema de Visión

Un sistema de visión es un módulo compuesto por varios elementos, que evalúa información de una fuente de imágenes, generalmente una cámara; extrae información de esas imágenes y realiza una acción con el resultado [11]. La figura 2.2 muestra un diagrama de la estructura típica de un sistema de visión, este esquema se compone de tres etapas. La primera etapa (a la izquierda), corresponde a la adquisición de la imagen, donde la escena del mundo real es digitalizada a través de sensores luminosos. Se compone por los siguientes elementos: cámara, lente, iluminación y objeto. La siguiente etapa es la de procesamiento digital de imágenes. Este procesamiento es realizado en una unidad de cómputo de propósito general o especifico, mediante diversas técnicas se extrae la información relevante de la imagen. La última etapa, la salida, corresponde a la acción generada a partir de la información obtenida en la etapa previa,

ya sea una acción mecánica, como separar una pieza en una línea de producción o, simplemente, una indicación visual de que se encontró determinado rasgo en una imagen.

Figura 2.2 – Esquema de un sistema de visión típico.

Existen muchos factores que se deben considerar al trabajar con un sistema de visión, ya que pueden afectar en el resultado del mismo, algunos de los principales son:

- Condiciones de iluminación
- Posición del objeto de interés respecto a la cámara
- Variaciones físicas del objeto de interés
- Variaciones físicas de la escena donde se encuentra el objeto
- Ruido inducido en la etapa de adquisición de la imagen

Para afrontar estos problemas, un sistema de visión típicamente trabaja con dos pasos esenciales:

- **Filtrado de la información de entrada:** Para reducir la información a ser procesada en un sistema de visión, el primer paso consiste en filtrar la información disponible. Mucha de la información contenida en una imagen resulta irrelevante o redundante para el propósito de sistema, además de que esta información es innecesaria, en el momento del procesamiento puede representar consumo de recursos adicionales, por ello la mayoría de los sistemas de visión como un primer paso realizan un filtrado

de la imagen de entrada, la salida de este filtrado define la cantidad de información que será utilizada y, por ende, el procesamiento que será necesario. Existen diversas técnicas para el filtrado, una de ellas consiste en reducir la imagen, o cortarla, para trabajar solamente con el área que contiene la información relevante, denominada región de interés (ROI por sus siglas en inglés, Región Of Interest). El conocer información previa sobre lo que se está trabajando sirve para filtrar aquello que no coincida con los elementos buscados, por ejemplo, si se trata de detectar objetos en una escena y dichos objetos son grandes, se puede filtrar toda la información de elementos pequeños pues resultan innecesarios para dicha aplicación. Otro tipo de filtrado, que también es muy común, es aquel que se encarga de la eliminación del ruido presente que pudiera causar problemas en la interpretación o posterior manipulación de la imagen. Este ruido puede ser producido por diferentes causas como: fuentes eléctricas, por suciedad en la lente o la cámara, ambiental, por movimiento de la cámara u otros.

- **Extracción y procesamiento de los rasgos clave en la imagen:** Luego de que la imagen fue filtrada, el siguiente paso es la extracción de rasgos relevantes, ya sea para tomar decisiones directamente sobre ellos o para traducirlos en información más útil para la aplicación específica. Para la selección y la extracción es de suma importancia conocer los rasgos característicos de lo que se busca, algunos ejemplos pueden ser color, forma, textura, tamaño, posición o brillantez.

2.1.3 Sistema de visión embebido

El término sistema de visión embebido implica una combinación de dos términos y las tecnologías a que estos términos se refieren, un sistema embebido y visión por computadora. Tal como se mencionó en la Sección 2.1.2, un sistema de visión es un módulo compuesto por varios elementos, que evalúa información de una fuente de imágenes (generalmente una cámara), extrae información de esas imágenes y realiza una acción con el resultado; mientras que un sistema embebido, puede definirse como un sistema de cómputo con una aplicación específica, que a diferencia de otros sistemas computacionales, como las computadoras

personales o supercomputadoras, que son de uso general [12], los sistemas embebidos son destinados a aplicaciones definidas y solamente requieren de hardware específico, esto les da algunas ventajas sobre los sistemas computacionales convencionales, que pueden ser:

- Son de menor tamaño, generalmente se basan en microcontroladores o microprocesadores.
- Consumen menos energía, al usar menos recursos de hardware.
- Tienen un menor costo debido al uso únicamente del software y hardware necesario.

Por lo anterior, se puede definir a un sistema de visión embebido como un sistema de cómputo de aplicación específica con capacidades para adquirir y procesar imágenes [13]. Tiene las mismas bondades que un sistema embebido y las capacidades de un sistema de visión. La figura 2.3 muestra 3 elementos típicos de un sistema embebido de visión, la cámara o sensor de adquisición (que puede o no estar encapsulado), la tarjeta embebida, que sirve de interfaz entre el sensor y la unidad de procesamiento y, finalmente, la unidad de procesamiento, que en sistemas embebidos generalmente se trata de un circuito integrado de dimensiones reducidas, ya sea microcontrolador o microprocesador.

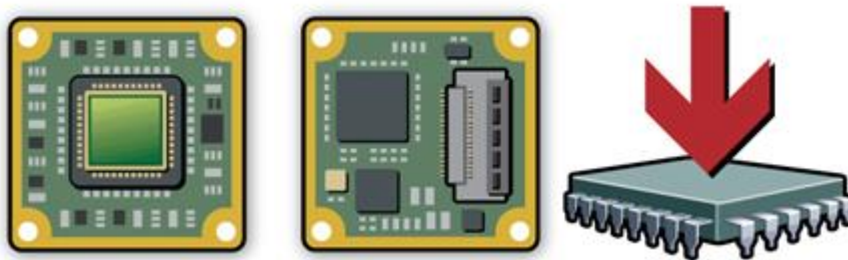

Figura 2.3 – Cámara, tarjeta embebida, microcontrolador para procesamiento (derechos a disposición del autor).

Algunos ejemplos de este tipo de sistemas son los módulos de navegación en automóviles, las básculas en supermercados que identifican el objeto que se está pesando, los dispositivos portátiles para retinografías, los dermatoscopios portátiles y otros. La figura 2.4 muestra un sistema embebido de visión de la marca Freescale, donde puede apreciarse, que la conexión de la cámara se realiza directamente con la tarjeta embebida, lo que reduce sus dimensiones a comparación de un sistema de visión tradicional.

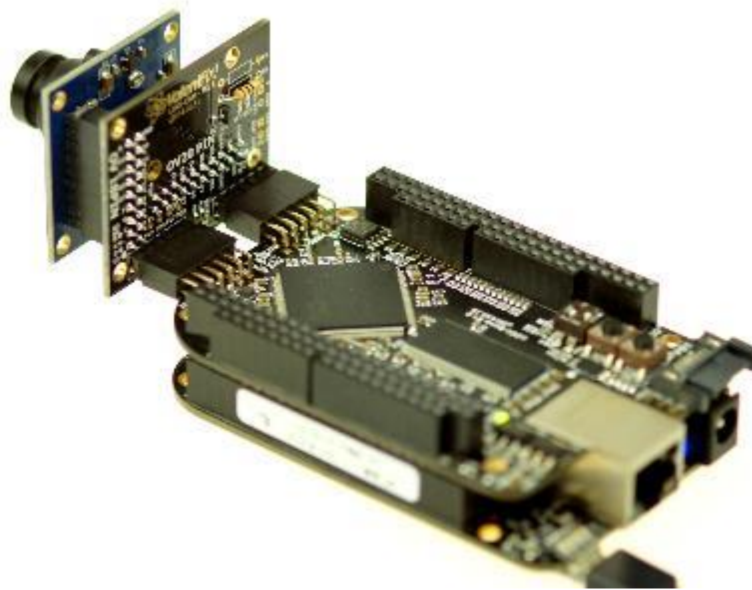

Figura 2.4 – Sistema de visión embebido utilizando una tarjeta de desarrollo de la compañía Freescale.

2.1.4 Iluminación

Uno de los principales retos al trabajar con el análisis de imágenes es la iluminación, ya que un sistema de visión depende de la buena calidad de las imágenes y la calidad de las imágenes depende de la luz. Una mala iluminación genera mucho ruido en la imagen y esto hace necesario más procesamiento para limpiarla y/o filtrarla. Al seleccionar la iluminación que se utilizará en el sistema de visión se deben tener en consideración tres objetivos principales a alcanzar:

- Maximizar el contraste de los rasgos de interés.
- Asegurar el mismo resultado entre diversos objetos.
- Mantener una escena estable que no se vea afectada por la luz ambiental.

La figura 2.5 muestra los elementos involucrados en la selección de la iluminación para un sistema de visión [11], los cuales se describirán a continuación:

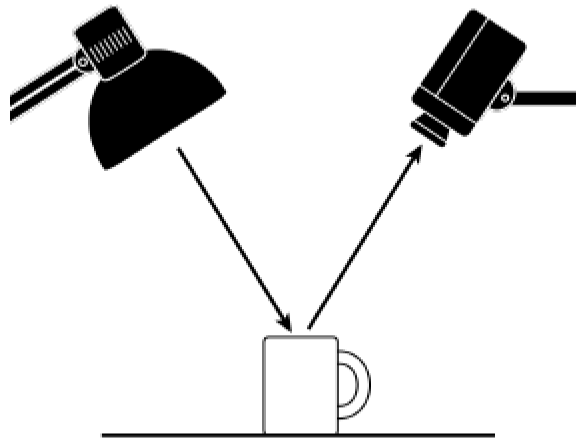

Figura 2.5 – Elementos involucrados en la iluminación: objeto, fuente de luz y cámara.

- **La fuente de luz**. Algunos tipos comunes de fuentes de luz son fluorescentes, LED, halógenas, xenón, haluro metálico y sodio de alta presión. Al seleccionar el tipo de lámpara, debe considerarse su luminosidad, medida en lúmenes, su direccionalidad y el ángulo con el que la luz se emitirá y su color o longitud de onda. También es importante asegurar que la luz ambiental no interfiera con la fuente de luz, esto se puede lograr aislando físicamente el sistema de visión o asegurando que la intensidad de la fuente sea mayor a la intensidad de la luz ambiental de modo que el efecto de esta última sea irrelevante.

- **El objeto**. Cuando la luz se refleja en una superficie, siempre sigue la ley de reflexión, que señala que la luz es reflejada por la superficie con el mismo ángulo con el que incide en ella. Dependiendo de la superficie de un objeto, la luz que incide sobre él, se reflejará de distinta manera, por ejemplo, una superficie lisa, suave y uniforme reflejará todos los rayos de luz en el mismo ángulo, por lo que lucirá brillante. Por el contrario, una superficie rugosa, como un pedazo de madera, bajo la misma luz lucirá opaco, ya que los rayos incidentes de luz serán reflejados en distintas direcciones. La geometría del objeto también influye en la reflexión de la luz ya que una superficie plana y una curva no se comportan de la misma manera. Otro factor a considerar es la composición del objeto, ya que algunos materiales tienden a absorber la luz mientras que otros la reflejan, inclusive algunos materiales

absorberán ciertas longitudes de onda mientras que reflejan otras, como es el caso del agua.

En la figura 2.6, podemos ver el efecto que tienen distintas superficies en el reflejo de los rayos de luz. De acuerdo a la interacción con la luz, algunos tipos de superficies son:

- **Lambertianas:** superficies mate o regulares, donde la luz se refleja de manera predecible, por ejemplo, madera o papel.
- **Dispersión sub superficial:** la luz penetra, interactúa con el material y sale en un punto distinto, por ejemplo, leche o la piel.
- **Especular:** superficies muy brillantes, reflejan los rayos provenientes de la fuente, pero también pueden tener reflejos de objetos adyacentes, por ejemplo, metales y espejos.

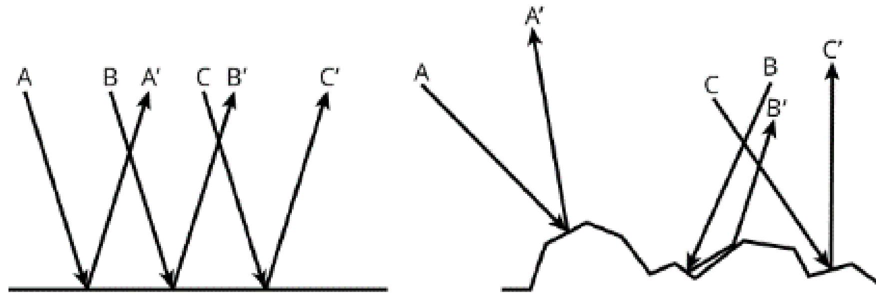

Figura 2.6 – Reflejo de luz en superficies suaves (izquierda) y rugosas (derecha).

- **La cámara**.- Se debe recordar que la cámara no captura al objeto como tal, sino la luz que el objeto refleja, por ello es importante asegurar que la cámara sea capaz de capturar la luz emitida y reflejada por la fuente de iluminación, ya que de otra forma será como si la iluminación no estuviera presente. Un ejemplo de ello puede ser una iluminación ultravioleta al utilizar una cámara infrarroja, dicha iluminación no tendrá ningún efecto en las imágenes capturadas.

Algunas técnicas de iluminación (ver figura 2.7) que pueden utilizarse para los sistemas de visión son las siguientes [14]:

- **Domo difuso**: se utiliza una fuente de luz difusa dentro de un domo, es efectiva para superficies curvas. Requiere cercanía al objeto.
- **Difuso en eje**: se utiliza una fuente de luz difusa frente al objeto, es efectiva para superficies especulares. Requiere cercanía al objeto.
- **Campo brillante**: se utiliza una fuente de luz puntual frente al objeto, es la iluminación más común, es efectiva para resaltar defectos topográficos. En superficies especulares ocasiona reflejos fuertes
- **Campo oscuro**: se utiliza una fuente de luz puntal localizada a un costado del objeto, es efectiva para resaltar defectos en las superficies. No es buena para iluminar superficies planas y suaves.
- **Luz de fondo difusa**: se utiliza una fuente de luz difusa localizada detrás del objeto, crea una silueta de alto contraste, muy efectiva para encontrar huecos u orificios. Los bordes de la silueta pueden aparecer borrosos.
- **Luz de fondo colimada**: se utiliza una fuente de luz puntual localizada detrás del objeto, crea una silueta de alto contraste y con bordes bien definidos, muy efectiva para obtener mediciones totales del objeto. Se pierde el detalle de la superficie.

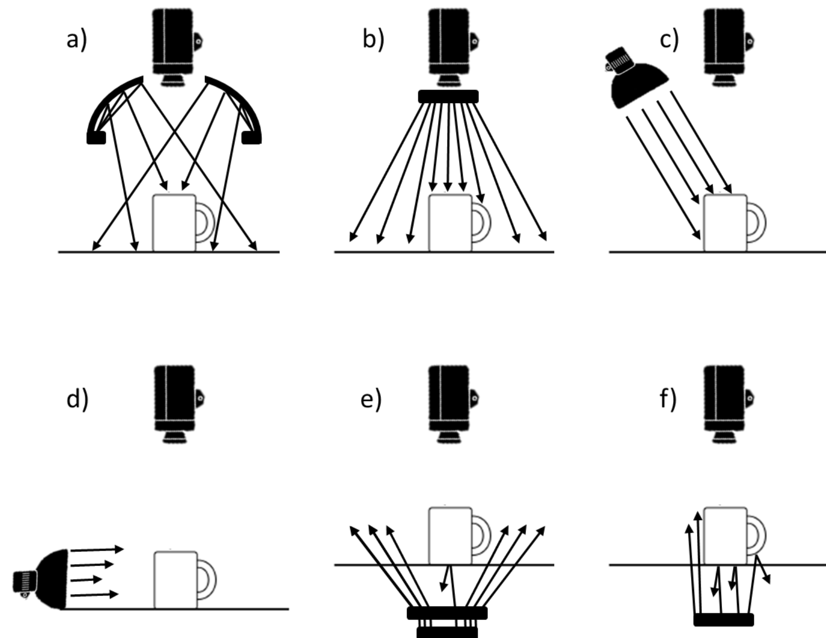

Figura 2.7 – Técnicas de iluminación: a) iluminación de domo difuso, b) difuso en eje, c) campo brillante, d) campo oscuro, e) de fondo difuso y f) de fondo colimada.

2.1.5 Campo de visión

Otro factor que se debe considera al diseñar o implementar un sistema de visión es el campo de visión. Este concepto se refiere a la longitud máxima que una cámara con determinada lente puede abarcar o capturar, esta distancia puede ser medida de forma horizontal, vertical o diagonal, y se conocerá como campo de visión horizontal, vertical o diagonal respectivamente [15]. El campo de visión está determinado por dos variables, la distancia de trabajo y el ángulo de visión. La distancia de trabajo se refiere a la separación que existe entre la lente y la escena que se está capturando, mientras que el ángulo de visión se refiere a la amplitud de captación que tiene la lente, es decir, el ángulo dentro del cual la lente es capaz de captar la escena. Mientras el ángulo de visión es un valor fijo para cada lente, el campo de visión puede cambiar conforme la lente se acerque o aleje de la escena. La figura 2.8, muestra estas tres características en una escena, así como el efecto en el campo de visión al variar el ángulo de visión.

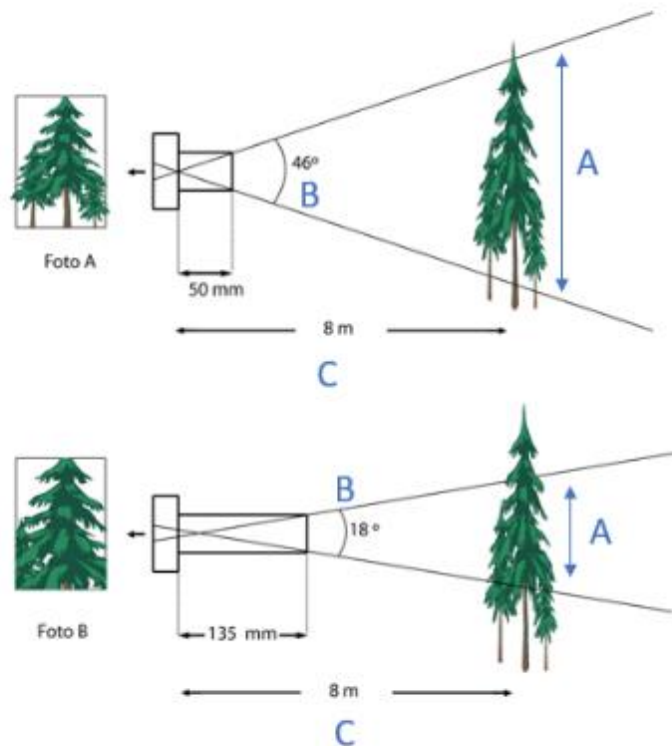

Figura 2.8 – A. Campo de visión vertical. B. Angulo de visión. C. Distancia de trabajo.

Se puede observar que, a mayor ángulo de visión, el campo de visión será también mayor, de manera similar, la relación que existe entre la distancia de trabajo y el campo de visión es directamente proporcional, de modo que, si la cámara se acerca a la escena, se reduce la distancia de trabajo y, por lo tanto, el campo de visión será menor.

2.2 Software de Desarrollo

Otro aspecto fundamental de un sistema de visión es el software de procesamiento digital, qué corresponde al conjunto de operaciones matemáticas que son aplicadas a la imagen de entrada para producir la imagen de salida. El desarrollo de estos algoritmos suele realizarse mediante software dedicado al desarrollo de aplicaciones de visión por computadora. Existe un gran número de librerías de software dedicadas a este tipo de aplicaciones tal como: Matlab, OpenCV, pypylon, ITK, VTK, BoofCV, VXL, VLFeat y Aforge.net, por mencionar algunos.

Cada una de estas librerías fue desarrollada en un determinado lenguaje de programación. Los lenguajes de programación, así como software de desarrollo tienen sus ventajas y sus desventajas. De este modo, la selección del entorno de desarrollo adecuado, así como del lenguaje de programación depende en gran medida del propósito que se persigue en determinada aplicación. Para el presente trabajo se decidió utilizar la librería OpenCV. Esta librería es ampliamente utilizada para el desarrollo de aplicaciones, tanto para investigación como en la industria. OpenCV incluye funciones para desarrollo de proyectos en lenguaje C++ y Python. En este proyecto de tesis se decidió utilizar Python.

2.2.1 Python

Python es un lenguaje de programación de alto nivel, orientado a objetos, interpretado y con semántica dinámica. Fue creado en 1991 [16]. Es un lenguaje de programación fácil de leer, ya que su sintaxis está enfocada en la legibilidad. Permite generar código de menor tamaño que su equivalente en otros lenguajes como: C, C++ o Java. Todas estas características lo hacen ideal para el problema propuesto en este proyecto.

Python se ha vuelto una de las opciones más populares para trabajos científicos y de investigación que requieren algún tipo de procesamiento computacional. Una de las principales razones de esto, es que Python cuenta con una gran comunidad establecida de científicos, ingenieros, estudiantes e investigadores que continúan desarrollándolo, extendiéndolo y promoviendo su uso. Originalmente este lenguaje no fue diseñado para cubrir las necesidades de la comunidad científica, pues a pesar de contar con una gran colección de tipos de datos que incluyen desde cadenas de texto, listas y diccionarios, le hacía falta una parte fundamental, un tipo de arreglo para computación numérica.

En 1995 una parte de la comunidad de usuarios de Python, formo el grupo “matrix-sig” que tenía como objetivo la creación de un nuevo tipo de datos para manejar matrices. Como parte de esta iniciativa nació un módulo llamado *Numeric*, capaz de manejar matrices. Si bien esto brindaba muchas ventajas a la comunidad científica, seguía sin cubrir todas sus necesidades. Durante varios años ingenieros e investigadores que utilizaban este módulo continuaron creando

y agregando complementos para facilitar su uso y cubrir muchas de sus necesidades específicas, hasta que el año 2003 algunos miembros de la comunidad de Python decidieron tomar varios de estos módulos creados, uniéndolos y generando así un paquete llamado *SciPy*. En él se recopilaban una serie de operaciones numéricas comunes que hacían uso de la estructura de datos *Numeric*. Por la misma época, fueron también liberadas las primeras versiones de *IPython* y *matplotlib*¸ la primera de ellas es una ventana de comandos interactiva que rápidamente se popularizó entre la comunidad científica, mientras que la segunda es la librería de gráficos en dos dimensiones más utilizada para ciencias computacionales. A pesar de que *Numeric* fue la base para muchos de los módulos para procesamiento numérico, estaba basada en un código relativamente obsoleto y difícil de actualizar y mantener, por lo que miembros del *Space Telescope Science Institute* en Baltimore crearon un nuevo paquete para manejo de matrices, llamado *Numarray*. El uso de este paquete ocasionó la separación de la comunidad en dos grupos, aquellos que continuaban utilizando y desarrollando *Numeric* y aquellos que se inclinaban por *Numarray*. En el 2006, esta división llego a su fin, cuando fue liberada la primera versión del paquete *Numpy* que combinaba las virtudes de *Numeric* y *Numarray*. Numpy sigue siendo al día de hoy la opción más utilizada por la comunidad científica de Python, para la manipulación de matrices [17].

Librerías como *ScyPy*, *matplotlib*, y *Numpy*, proveen un ambiente de computación numérica muy similar a Matlab, pero también existen otras librerías que se asemejan a programas como *Mathematica*, *Maple* o *Magma*, que están más orientados al campo de las matemáticas. Uno de ellos es *SymPy*, un sistema de algebra computacional escrito en Python. Otra opción es *mpmath*, que proporciona aritmética multipunto de punto flotante, además soporta números de punto flotante reales y complejos y tiene funciones para series y productos infinitos, integrales, derivadas, límites, ecuaciones no lineales, ecuaciones diferenciales ordinarias, funciones especiales, aproximación de funciones y álgebra lineal. Finalmente, tenemos la librería *Sage*, que tiene soporte para varios dominios matemáticos, que incluyen álgebra lineal, cálculo, teoría de números, criptografía, álgebra conmutativa, teoría de grupos, combinatoria, teoría de grafos y muchos más.

Generalmente los lenguajes de alto nivel tienden a ser más lentos que los de bajo nivel, si bien esto aplica para Python, este nos ofrece la ventaja de poder llamar o ejecutar librerías o módulos programados en lenguajes de bajo nivel como C o C++, teniendo así las ventajas del desarrollo en un lenguaje de alto nivel, sin comprometer los tiempos de ejecución. Otro aspecto importante de Python es que se trata de un lenguaje de código abierto, de modo que no se requiere ningún tipo de licencia para utilizarlo. Otra gran ventaja que ofrece Python, debido a que no necesita el proceso de compilación, por tratarse de un lenguaje interpretado, el ciclo de edición, prueba y depuración es mucho más rápido, agilizando el proceso de desarrollo en general.

2.2.2 OpenCV

OpenCV es una librería de software de visión por computadora construida para proporcionar una infraestructura común para las aplicaciones de esta área y para acelerar el uso de la percepción de máquina en aplicaciones comerciales [18]. Esta librería comenzó como un proyecto de investigación de Intel en 1998 y ha estado disponible desde el 2000 bajo la licencia de código abierto [19], es decir, al igual que Python, puede adquirirse sin necesidad de ningún tipo de licencia y puede ser utilizada libremente, incluso en aplicaciones comerciales.

Esta librería contiene más de 2500 funciones para diversas aplicaciones de visión por computadora. Una de las principales ventajas que ofrece es la gran popularidad que tiene, ya que es utilizada por grandes compañías como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota. De modo que es fácil encontrar soporte, resolver dudas y encontrar ejemplos de su uso. Además, esta librería tiene interfaces para distintos lenguajes de programación como: C++, Python, Java e incluso Matlab y es compatible con diversas plataformas como: Mac OS, Linux, Windows y Android. El hecho de que la mayoría de los algoritmos implementados en esta librería fueron desarrollados en C o C++, la hacen ideal para su uso en sistemas embebidos.

OpenCV tiene una estructura modular, lo que significa que contiene varias liberarías compartidas o estáticas. OpenCV puede instalarse de dos maneras, utilizando el paquete pre-compilado, o compilándolo desde sus archivos fuentes. Si se opta por la segunda opción podrá

decidirse cuales librerías serán incluidas en la instalación. Las librerías o módulos más importantes que conforman OpenCV [20] son:

- **core**. - Es un módulo compacto que define las estructuras básicas de datos, incluye el arreglo multidimensional *Mat* y las funciones básicas utilizadas por el resto de los módulos.
- **imgproc. -** Es un módulo de procesamiento de imágenes que incluye filtros lineales y no lineales, transformaciones geométricas, conversiones de espacios de color y procesamiento de histogramas, entre otras.
- **video. -** Este módulo sirve para el análisis de video, incluye estimación de movimiento, sustracción de fondo y algoritmos de seguimiento de objetos.
- **calib3d.-** Modulo que incluye algoritmos básicos de geometría de vista múltiple, calibración de cámaras, estimación de posición de objetos, algoritmos de correspondencia estéreo y elementos de reconstrucción 3D.
- **features2d.-** En este módulo se incluyen detectores de rasgos, descriptores y comparadores de descriptores.
- **objdetect. -** Este módulo permite la detección de objetos e instancias de clases predefinidas, por ejemplo, rostros, ojos, gente, o automóviles.
- **highgui. -** Incluye funciones para crear interfaces de usuario simples.
- **videoio. -** Este módulo sirve como interfaz para capturar y codificar video.

2.3 Estado del Arte

La detección de defectos sobre superficies ha sido estudiada ampliamente, existen bastantes trabajos sobre el tema que cubren métodos como mapeo de superficies utilizando electrostática, métodos de contraste (utilizando elementos fluorescentes sobre las superficies), termografía infrarroja, pruebas de corriente de Eddy, métodos ópticos y pruebas ultrasónicas entre otros [21]. En [22] se presenta un método para medición de defectos de escala nanométrica en superficies

metálicas, a través de un micro sensor de temperatura. Por medio de este sensor, los autores pueden identificar pequeñas desviaciones en el flujo térmico de la superficie metálica. Estas desviaciones de flujo corresponden a defectos en la superficie, la sensibilidad del sensor es capaz de detectar defectos menores a 10 nm. Si bien estos métodos cumplen con su cometido en cuanto a la detección, la mayoría de las ocasiones son aplicados en ambientes de laboratorio y no en ambientes industriales ya que requieren un hardware específico y configuraciones especiales [23]. Por ello, cuando se piensa en la automatización de este tipo de detecciones y especialmente pensando en implementar dichas automatizaciones en ambientes industriales, una de las mejores opciones son los métodos de visión por computadora. La visión por computadora es una ciencia bastante extensa, que permite abordar el problema con una gran variedad de técnicas, estas dependerán de la aplicación específica, es decir, las características de las superficies (como color, material, textura, reflectividad, dimensiones, etc.) y las características de los defectos que se pueden presentar (dimensiones, posiciones, formas, cantidades, etc.,). A continuación, se presentan algunos trabajos de visión por computadora que están relacionados con el proyecto desarrollado.

2.3.1 Métodos de umbralizado

El método de segmentación más elemental es el de umbralizado, existen varios trabajos en los que se ha utilizado este enfoque para abordar diversos problemas, en uno de ellos, un sistema de visión convencional ha sido utilizado para la detección de defectos en superficies transparentes, utilizando una luz de fondo (backplane) para generar el contraste necesario para el umbralizado, alcanzando eficiencias del 99%, aunque está limitado a defectos de 0.1 mm [24]. Si bien la superficie de gel de los electrodos de dispersión, que analizamos en este proyecto, es una superficie semi-transparente, los métodos que utilizan luz de fondo para generar un alto contraste no son una opción, debido a que el gel se encuentra cubriendo una superficie de aluminio totalmente opaca que no permite que la luz pase a través de ella.

Otro enfoque de visión por computadora que puede utilizarse se menciona en [23], donde se propone un método para la detección de defectos en superficies de acero, utilizando una técnica conocida como mesoscopía, que intenta combinar los campos de visión amplios de las fotografías tradicionales, con las capacidades de alta resolución de la microscopia óptica. En

dicho trabajo, mediante una cámara lineal de muy alta resolución obtienen imágenes con 2.75 µm por pixel; Una vez adquiridas las imágenes recurren al método de segmentación más elemental, el umbralizado. Si una imagen puede expresarse en ***L*** niveles de gris, el número de puntos con un nivel de gris ***j*** se expresa con $\boldsymbol{m_j}$, mientras que el número total de puntos ***M*** se puede representar de la forma

$$M = m_1 + m_2 + \cdots + m_L \tag{2.1}$$

El histograma para esta imagen es una distribución de probabilidad dada por

$$p(k) = \frac{m_k}{M}, \quad p(k) > 0, \quad \sum_{k=1}^{L} p(k) = 1 \tag{2.2}$$

La técnica de umbralizado consiste en encontrar un valor ***t*** (umbral) que divida a la imagen en dos clases $\boldsymbol{C_0}$ y $\boldsymbol{C_1}$. La clase $\boldsymbol{C_0}$ representa pixeles con niveles de gris [0, 1, …, ***t***] y la clase $\boldsymbol{C_1}$ representa pixeles con niveles [***t***+1, ***t***+2, …, ***L***]. Para el problema dado en ese artículo, se busca un umbral que permita agrupar los defectos en una clase y la superficie de acero en otra. En [23] comparan 6 diferentes métodos para encontrar el valor adecuado del umbral ***t***,

- Método de Otsu
- Extensión del método de Otsu basado en mediana.
- Umbralizado de mínimo error (MET por sus siglas en inglés).
- Extensión del método MET basado en mediana.
- Método de Otsu de contraste ajustado (propuesto por los autores).
- Método de Otsu de contraste ajustado basado en mediana (propuesto por los autores).

Los algoritmos basados en mediana, tiene un buen desempeño en cuanto a detección, pero un pobre desempeño en rendimiento, pues requieren muchos recursos computacionales y tiempo de procesamiento con imágenes grandes pues se involucran la detección de la mediana. Resultando el método propuesto por el autor, método de Otsu de contraste ajustado, el más rápido y preciso de los 6, sin embargo, este método es muy propenso a ruido y sombras en la imagen, causando falsas detecciones. Y si bien la superficie de acero estudiada en [23] puede

asemejarse a la superficie de aluminio de los electrodos de dispersión, la naturaleza y tamaño de los defectos son muy distintos. Los defectos de los electrodos de dispersión son de mayor tamaño, por lo que una imagen de muy alta resolución resultaría un gasto innecesario de recursos computacionales en el momento del procesamiento. Otros trabajos similares han conseguido una detección fiable de defectos de 1 μm de dimensión, pero tiene una baja eficiencia computacional, por lo que los tiempos de análisis pueden tomar hasta 2 horas en superficies de 800 x 400 mm [24], lo que nuevamente indica que se trata de un método poco viable para ambientes de producción que requieren tiempos de procesamiento rápidos.

Algunos trabajos, buscan mejorar las capacidades de la detección por umbralizado combinándolo con otros métodos, tal es el caso de [25] que propone la detección de defectos de superficies de aluminio, mediante la ejecución de dos algoritmos de segmentación independientes, al finalizar, los segmentos detectados por ambos algoritmos son evaluados, y los pixeles que se hayan detectado por cualquiera de los dos métodos serán considerados como defectos. El primero de los algoritmos utilizados es una segmentación por umbral automático basado en el modelo de la imagen, es decir, para cada imagen se obtiene un valor de umbral diferente que esta dado en función del fondo de la imagen, y para ello se parte de un modelo Gaussiano para modelar el fondo de la imagen,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.3}$$

$\boldsymbol{x}$ representa el valor de gris de la imagen, mientras que $\boldsymbol{\mu}$ y $\boldsymbol{\sigma}$ representan la media y la desviación estándar respectivamente, la imagen es subdividida en regiones y para cada una de ellas se obtiene media y varianza, la media de la región que presente la menor varianza será utilizada para obtener el valor del fondo de la imagen y realizar el umbralizado de la forma

$$f(x) = \begin{cases} 1 & si \quad x \in [0, \mu - \eta_1\sigma] \cup [\mu + \eta_2\sigma, 255] \\ 0 & si \quad x \in [\mu - \eta_1\sigma, \mu + \eta_2\sigma] \end{cases} \tag{2.4}$$

Donde $\boldsymbol{\eta_1}$ y $\boldsymbol{\eta_2}$ son parámetros que permiten controlar la sensibilidad de la detección. El segundo algoritmo de segmentación utilizado es adaptativo basado en entropía. El concepto de entropía establece que una imagen de la superficie normal del aluminio no contiene información,

y la entropía en dicha imagen es casi cero, cuando aparece un defecto en la superficie del aluminio, la entropía de esa imagen aumenta. La entropía de la imagen para un valor de umbral ***d,*** estará dada por

$$H(d) = -\frac{1}{a}\sum_{i=1}^{d} h_i log_2 h_i - \frac{1}{MN-a}\sum_{i=d+1}^{L-1} h_i log_2 h_i + log_2\big(a(MN-a)\big) \tag{2.5}$$

El valor óptimo del umbral se puede expresar como

$$d = arg\max\big(H(d)\big) \tag{2.6}$$

Los resultados obtenidos en [25] muestran una mejora significativa al combinar ambos métodos de segmentación, pues el grado de detección de los defectos es mucho mayor que al ejecutar los algoritmos de forma independiente, esto se realiza sin comprometer el tiempo de procesamiento ya que ambos algoritmos son ejecutados en paralelo, al combinar los resultados de ambos algoritmos se disminuye el porcentaje de error, que se genera por el ruido proveniente de cambios en el entorno.

2.3.2 MSER y plantilla

En [26] se aborda la detección de defectos en superficies metálicas, específicamente defectos en vías de tren. En esencia utilizan el procedimiento típico de inspección:

- Adquisición de la imagen
- Pre-procesamiento (restauración y mejora)
- Segmentación
- Selección y extracción de rasgos
- Clasificación
- Correspondencia entre rasgos
- Toma de decisiones

- Muestra de resultados

Los algoritmos de segmentación son utilizados para detectar regiones o grupos de pixeles similares dentro de una imagen. El método de segmentación utilizado en [26] es llamado, regiones extremas de máxima estabilidad (MSER por sus siglas en inglés). Este algoritmo extrae un número de regiones covariantes de una imagen $\boldsymbol{I(x,y)}$, que son conocidas como MSERs. El concepto de regiones de máxima estabilidad (MSR por sus siglas en inglés) fue definido considerando el conjunto de todos los posibles umbrales para obtener una imagen binaria $\boldsymbol{B_{th}(x,y)}$ a partir de la imagen $\boldsymbol{I(x,y)}$,

$$B_{th} = \begin{cases} 1 & si \quad I(x,y) \geq \alpha_{th} \\ 0 & de\ otra\ forma \end{cases} \tag{2.7}$$

Una MSER es una región conectada en $\boldsymbol{B_{th}(x,y)}$ con un cambio de tamaño mínimo a través de varios umbrales $\boldsymbol{\alpha_{th}}$, conforme el rango de umbral aumenta, las binarización se va volviendo estable solo en determinadas regiones, finalmente las regiones estables serán las zonas donde la superficie metálica presente defectos. Por medio de este método se logra un nivel de detección del 98% y se atribuyen el 2% de error a las variaciones en las condiciones de iluminación.

Existen otros métodos que resultan computacionalmente más eficientes, como el propuesto en [27], donde los autores plantean una solución para identificar los posibles defectos en el revestimiento de fósforo que se utiliza en determinado tipo de LEDs. Los defectos se generan en el proceso de revestimiento y pueden ser rasgaduras, manchas, curvaturas en la superficie, zonas vacías o zonas con excesos. El método propuesto se basa en la caracterización de una imagen de referencia o plantilla, esto es, tomando varias imágenes de LEDs que estén perfectamente cubiertos con fósforo, es decir, libres de defectos, estas imágenes son utilizadas después como entrada para un algoritmo de diferencias. Debido a que los LEDs no varían en tamaño, forma, textura o color, al momento de inspeccionar algún LED, se procede a obtener una imagen, realizarle un pre-procesamiento para eliminar ruido y asegurar su orientación, finalmente se compara con las imágenes de referencia obteniendo las diferencias que existen entre cada pixel. Todos aquellos puntos que presenten una variación mayor a un umbral definido son considerados un defecto. Este método es relativamente rápido, pues puede realizar el análisis de una imagen en alrededor de 100 ms, pero es solamente válido cuando se utiliza en objetos

que son muy consistentes como es el caso de los LEDs, ya que la variación de un chip a otro es mínima, mientras que el producto objetivo de esta tesis, los electrodos de dispersión, tienen una gran variación debido a sus características físicas (textura, color y relieve del gel).

2.3.3 Luz estructurada

Una de las técnicas más populares para el análisis de superficies cuando se utiliza visión por computadora es la luz estructurada. Un sistema típico de luz estructurada consiste de una fuente de emisión de luz (típicamente laser), una unidad de proyección y una o más cámaras. Se proyectan patrones con estructuras conocidas, que pueden ser líneas horizontales, verticales, mallas, círculos, u otros arreglos más complejos, en el objeto bajo prueba, luego comparando la forma del patrón adquirido con las cámaras y el patrón proyectado, mediante el procesamiento adecuado se puede obtener la presencia y tamaño de un objeto [28]. Los sistemas de medición mediante luz estructurada, por lo general tiene un bajo costo, una alta eficiencia y gran precisión, aunque su implementación no es del todo sencilla. La parte más importante en un sistema de luz estructurada, es la extracción del patrón proyectado, para esto se utilizan detectores de rasgos, pero la mayoría de los detectores de rasgos generales como Harris, SURF, SUSAN u otros, no son adecuados para los patrones utilizados en estos sistemas, por lo que deben diseñarse detectores de rasgos específicos para cada patrón de luz, tal como se menciona en [29], donde se plantea un método de detección de rasgos basado en el detector de rasgos de plantilla cruzada. El detector propuesto tiene la finalidad de encontrar los puntos de cruce del patrón de luz diseñado por los autores, este es un patrón de luz binario con rombos blancos y negros del mismo tamaño y dentro de cada rombo blanco se encuentran figuras geométricas también negras (se utilizan 8 figuras diferentes). Las pruebas realizadas para el algoritmo propuesto para dicho patrón muestran un rango de detección de hasta el 99% de los puntos. Para la extracción del patrón en la imagen, además del algoritmo de detección de rasgos, resulta importante el sistema de adquisición, es decir, la o las cámaras empleadas. Se han realizado algunos trabajos al respecto, explorando la teoría de que más de una cámara utilizada en el mismo sistema puede facilitar el trabajo de extracción o mejorar los resultados del sistema, tal es el caso de [30], que propone un sistema con un proyector y cuatro cámaras, en un arreglo semejante a una cruz,

donde el proyector se ubica en el centro y las cámaras en cada uno de los extremos de la cruz, se proyecta un patrón de 8000 puntos blancos con fondo negro sobre la superficie a medir, la idea de utilizar cuatro cámaras es para eliminar ambigüedades y poder capturar todos los puntos aun cuando algunos de ellos se pierdan para alguna de las cámaras, ya sea por alguna sombra, ruido o simplemente la perspectiva, este será captado por otra cámara y al final se tendrá la mayor cantidad posible de puntos. Por otro lado en [31], se propone un sistema similar, utilizando un proyector LCD y dos cámaras CCD para capturar simultáneamente el patrón proyectado, el patrón utilizado al igual que en [30] consiste en una estructura blanca sobre un fondo negro, pero utiliza líneas en lugar de puntos, las dos cámaras ayudan a eliminar las áreas sombreadas en la superficie del objeto y mejoran la precisión del sensor a través de la redundancia de las mediciones. Se pueden obtener mediciones con el sistema propuesto que presentan un error relativo menor al 1%. Tanto en [30] como en [31], se considera y caracteriza la distorsión en la lente del proyector, para considerar el efecto que tendrá sobre el patrón proyectado, de ese modo se puede realizar una comparación real y objetiva entre la imagen adquirida y el patrón proyectado. Más recientemente en [32] se propone un método con una sola cámara y un único proyector, el proyector utiliza un patrón compuesto por la mezcla de dos ondas periódicas con frecuencias distintas f_1 y f_2, donde f_1 está dada por los cambios de color y f_2 por los cambios de intensidad. Si bien trabajos anteriores usaban información redundante de varias cámaras, para robustecer el sistema, este trabajo propone obtener información redundante utilizando solo una cámara y solo un proyector, pero el patrón proyectado contiene dos señales embebidas que proporcionaran dos mediciones diferentes. El sistema proyector/cámara es caracterizado como un sistema de visión en estéreo, donde el centro óptico de la cámara y el del proyector están alineados verticalmente y el patrón proyectado consiste en líneas horizontales, de modo que la fase de las ondas de color e intensidad se puede medir horizontalmente. Esta fase es utilizada luego para calcular la profundidad del objeto capturado en la imagen, al compararse contra las fases conocidas de una imagen de referencia (sin objeto). Los resultados presentados en [32] indican que el sistema es muy rápido y preciso con errores menores al 0.3%, pero tiene el gran inconveniente de ser muy sensible al ruido inducido por la luz ambiental. Si se tienen defectos sobre una superficie y estos varían en profundidad respecto a la

misma, las técnicas de luz estructurada pueden ser una opción para su detección, pero no si los defectos presentan solamente variaciones como cambios de color, textura, forma, etc. En el caso del problema abordado por el presente trabajo, se tiene defectos que cambian en profundidad, pero también otros que no, esto aunado al hecho de que en un sistema de luz estructurada se tiene que diseñar un detector de rasgos específico para el patrón a utilizar, caracterizar las distorsiones del proyector utilizado, evaluar la posibilidad de que múltiples cámaras y/o múltiples proyectores sean requeridos y, de ser el caso, encontrar el arreglo ideal para estos elementos. Todas estas consideraciones hacen de esta técnica una opción poco viable para resolver el problema planteado.

2.3.4 Modelos estadísticos

Otra forma de eliminar el ruido del entorno, principalmente el de la iluminación, es a través de técnicas estadísticas. Tal es el caso del método propuesto en [33], donde se explora el análisis de imagen multivariante (MIA por sus siglas en inglés) en superficies metálicas para detectar defectos. Cuando se utilizan sistemas de visión para inspeccionar superficies metálicas se enfrentan dos retos principales, primero la superficie tiene una alta reflexión especular, es decir, en un solo ángulo, por lo que la fuente de iluminación puede observarse reflejada en un punto de la superficie, causando zonas mucho más iluminadas que otras; el segundo problema es que la naturaleza áspera del metal no es completamente uniforme, hay puntos más brillantes u opacos que otros, lo que puede ocultar defectos o aparentarlos cuando no los hay. Partiendo de esto, los autores de [33] intentan simular lo que hace una persona al inspeccionar una superficie metálica, típicamente observa la superficie desde diversos ángulos hasta encontrar alguno o algunos puntos donde el defecto sea más notorio. En su trabajo se utiliza una cámara fija y se capturan varias imágenes cambiando la altura de la fuente de iluminación, y con ello cambiando el ángulo de incidencia de la luz en el metal, todas las imágenes capturadas son procesadas como una sola imagen multivariada $\boldsymbol{X}$ con tres dimensiones, de la forma

$$X = [L_1(x, y;\ h_1), L_2(x, y;\ h_2), \dots, L_M(x, y;\ h_M)] \tag{2.8}$$

M representa el número de imágenes con distinto ángulo de iluminación, cada imagen monocromática es de tamaño ***I***x***J***. Luego se utiliza el análisis de componentes principales (PCA por sus siglas en inglés) para descomponer la imagen multivariada ***X*** en una serie de componentes principales ***A***

$$X = \sum_{a=1}^{A} t_a P_a^T + E \tag{2.9}$$

Cada t_a representa una imagen, denominada de puntuación, no todas las imágenes de puntuación contendrán información relevante, por lo que se seleccionan solo aquellas que presenten mayor diferencia respecto a las demás. Conociendo los componentes principales, puede calcularse el residuo ***E*** que contendrá la información de los defectos

$$E = X\left[I - P_1 P_1^T - \sum_a P_a P_a^T\right] \tag{2.10}$$

Con la ecuación anterior se eliminan los componentes principales de la imagen multivariada ***X***, quedando una imagen multivariada residual ***E***

$$E = [e_1 \; e_2 \dots \; e_N] \tag{2.11}$$

Finalmente, mediante la suma residual de cuadrados, se puede revelar la información potencial de ***E,***

$$q = [e_1 e_1^T \; e_2 e_2^T \dots \; e_N e_N^T] \tag{2.12}$$

Si el proceso completo es aplicado a una imagen sin defectos, se obtendrá una imagen de referencia, q_{ref}, sin defectos. Se selecciona el valor del pixel más grande en q_{ref} para utilizarse como umbral de detección ***α.*** Finalmente se aplica el proceso completo a las imágenes a inspeccionar utilizando el umbral ***α*** para binarizar las imágenes ***q***, una operación morfológica de apertura es aplicada después para eliminar el ruido remanente después del procesamiento. En los resultados obtenidos por los autores se consiguió un 100% de detección de defectos.

Recientemente en [34] se plantea la detección de defectos en superficies de acero utilizando un nuevo modelo de Haar-Weibull-Varianza (HWV por sus siglas en inglés), el primer paso en

su algoritmo consiste en aplicar una difusión anisótropa a la imagen con el fin de suavizar la textura del acero (como ya se mencionó las superficies metálicas por naturaleza no son uniformes) para eliminar pseudo-defectos. Después se procede a extraer el conjunto de características Haar de la imagen, aplicando un filtro pasa bajas combinado con un filtro pasa altas a las filas de la imagen, obteniendo la transformación de filas de la forma

$$I_R(i,j) = \frac{[I(i,2j) + I(i,2j+1)]}{\sqrt{2}} \tag{2.13}$$

$$I_R\left(i,j+\frac{N}{2}\right) = \frac{[I(i,2j) - I(i,2j+1)]}{\sqrt{2}} \tag{2.14}$$

Donde $\boldsymbol{I(i,j)}$ es la imagen original, $\boldsymbol{I_R(i,j)}$ es la transformada de filas, $\boldsymbol{i}$ tiene valores de 0 a $\boldsymbol{M}$-1 mientras que $\boldsymbol{j}$ tiene valores de 0 a $\frac{\boldsymbol{N}}{\boldsymbol{2}}$-1. Después el mismo filtro es aplicado a las columnas de la $\boldsymbol{I_R(i,j)}$

$$I_C(i,j) = \frac{[I_R(2i,j) + I_R(2i+1,j)]}{\sqrt{2}} \tag{2.15}$$

$$I_C\left(i+\frac{M}{2},j\right) = \frac{[I_R(2i,j) - I_R(2i+1,j)]}{\sqrt{2}} \tag{2.16}$$

Donde $\boldsymbol{I_C(i,j)}$ es la transformada de columnas resultante, $\boldsymbol{i}$ tiene valores de 0 a $\frac{\boldsymbol{M}}{\boldsymbol{2}}$-1 y $\boldsymbol{j}$ tiene valores de 0 a $\boldsymbol{N}$-1. Con el proceso de trasformación se obtiene cuatro sub imágenes de la forma

$$A(i,j) = I_C(i,j) \tag{2.17}$$

$$D_1(i,j) = I_C\left(i,j+\frac{N}{2}\right) \tag{2.18}$$

$$D_2(i,j) = I_C\left(i+\frac{M}{2},j\right) \tag{2.19}$$

$$D_3(i,j) = I_C\left(i+\frac{M}{2},j+\frac{N}{2}\right) \tag{2.20}$$

A es la matriz de coeficientes aproximados, $\boldsymbol{D_1}$, $\boldsymbol{D_2}$ y $\boldsymbol{D_3}$ son las matrices de coeficientes horizontales, verticales y diagonales respectivamente. Después se extrae el parámetro Weibull

de la imagen resultante, la distribución Weibull se puede obtener de la función de densidad de probabilidad siguiente

$$f_{\alpha,\beta}\big(I(x)\big) = \frac{\beta}{\alpha}\left(\frac{I(x)}{\alpha}\right)^{\beta-1} e^{-\left(\frac{I(x)}{\alpha}\right)^{\beta}} \tag{2.21}$$

Donde $\boldsymbol{\alpha}$ y $\boldsymbol{\beta}$ representan los parámetros de escala y forma. Una novedad del método propuesto en [34] es la introducción de la varianza en el modelo Weibull para incrementar la distancia interclases entre el fondo y los defectos. Esto se realiza añadiendo la varianza en los parámetros de escala y forma con determinado peso

$$\alpha' = \xi\alpha + \zeta\sigma \tag{2.22}$$

$$\beta' = \xi\beta + \zeta\sigma \tag{2.23}$$

Donde $\boldsymbol{\xi}$ y $\boldsymbol{\zeta}$, son factores de peso, y $\boldsymbol{\xi + \zeta = 1}$. Las imágenes procesadas muestran que en el espacio HWV se forman dos grupos claramente diferentes y separados, uno correspondiente a los defectos y otro al fondo, finalmente se aplica un método de umbralizado de Otsu para realizar la segmentación. Los resultados obtenidos muestran una eficiencia de hasta el 96%, pero la gran ventaja que presenta este novedoso método es que funciona para defectos arbitrarios inclusive en condiciones de bajo contraste.

III. CARACTERIZACIÓN DE DEFECTOS EN ELECTRODOS DE DISPERSIÓN

Los electrodos de dispersión de un solo uso o desechables se componen de una superficie adhesiva, sobre la cual se coloca una placa conductora, generalmente de aluminio, que a su vez se encuentra recubierta por un gel conductor que sirve para eliminar todos los espacios vacíos entre la placa conductora y la piel sobre la cual se aplica. La figura 3.1 muestra la estructura básica de un electrodo de dispersión, así como una representación de un corte transversal en el mismo, que permite apreciar las 3 capas que lo componen, superficie adhesiva, material conductor y gel conductor.

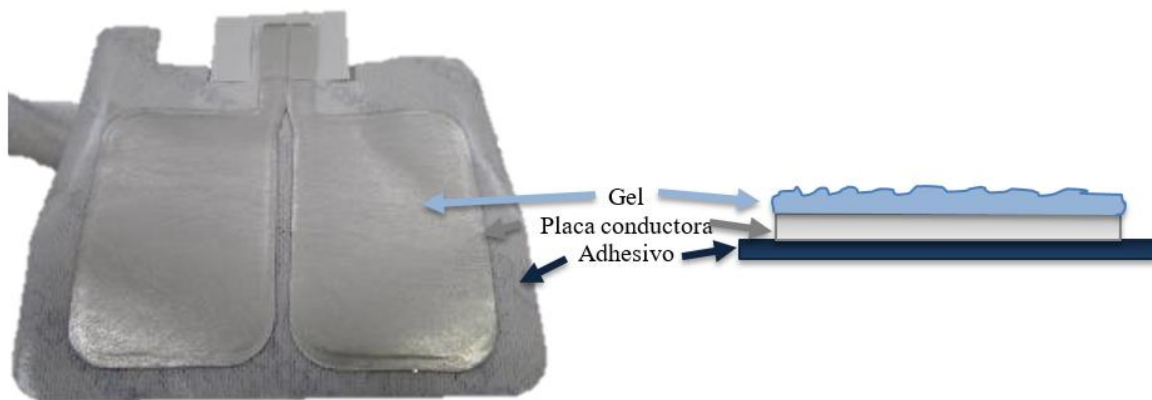

Figura 3.1- Electrodo de dispersión y su corte transversal.

Existen varios tipos de imperfecciones que pueden presentarse en la superficie del gel durante el proceso de fabricación de los electrodos, si bien algunas de ellas no representan ningún riesgo, otras pueden ser causantes de fallas durante el uso del mismo, de modo que todas ellas son consideradas defectos en el producto. Durante la elaboración de esta tesis, como primera etapa del desarrollo, se obtuvieron muestras de electrodos de dispersión con las diversas imperfecciones que pueden presentar. De estas muestras se segregaron las que resultan aceptables de aquellas que no. Luego se crearon grupos, basados en las características físicas comunes, resultando cinco grupos o tipos en los que se pueden clasificar todas las imperfecciones consideradas como defecto. A continuación, se presentan los distintos grupos y sus características.

3.1 Defecto de De-laminación

Este tipo de defecto se presenta siempre en los contornos del electrodo, específicamente en las orillas de la superficie de gel, consiste en un desprendimiento o desplazamiento de gel, puede ser causado por el operador durante el proceso de ensamble al manipular el electrodo, ya sea por remover parte del recubrimiento o solamente por desplazarlo hacia el centro del electrodo, provocando así que las orillas del aluminio sean visibles o casi visibles. En ocasiones puede presentarse con una profundidad tal que expone el aluminio, mientras que, en otras, solamente genera depresiones en el gel o zonas con una capa más delgada que, si bien no exponen el aluminio, claramente se advierte un severo adelgazamiento en su recubrimiento. Se localiza en zonas aisladas en cualquier parte del contorno, pequeñas y de geometría irregular. La figura 3.2 muestra dos ejemplos de este tipo de defecto.

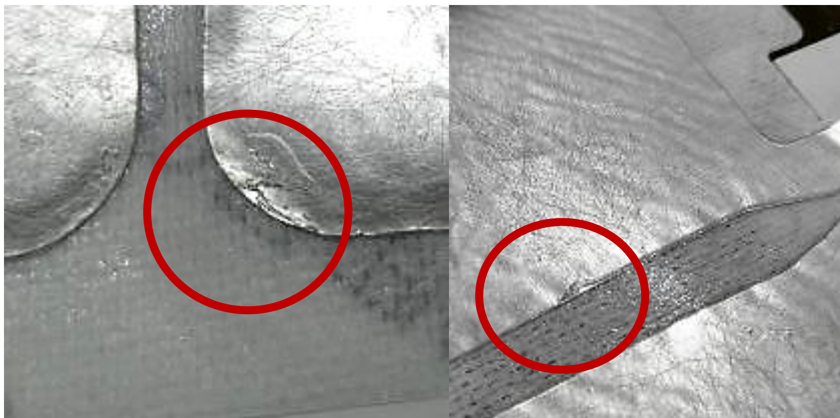

Figura 3.2 – Ejemplos del defecto de de-laminación.

3.2 Defecto de Ojo de Pescado

Este tipo de defecto se localiza siempre en el área interna del electrodo (no en los bordes) con geometrías irregulares. Se genera durante la fabricación de la capa de gel y se caracteriza por adelgazamientos en la superficie. Siempre se presenta en la superficie de gel opuesta al aluminio, es decir, siempre en la superficie expuesta al aire. El tamaño de este tipo de defectos, así como su ubicación, varía dentro de la superficie ya que pueden aparecer en cualquier zona, y con cualquier dimensión. La figura 3.3 muestra algunos ejemplos de este tipo de defectos.

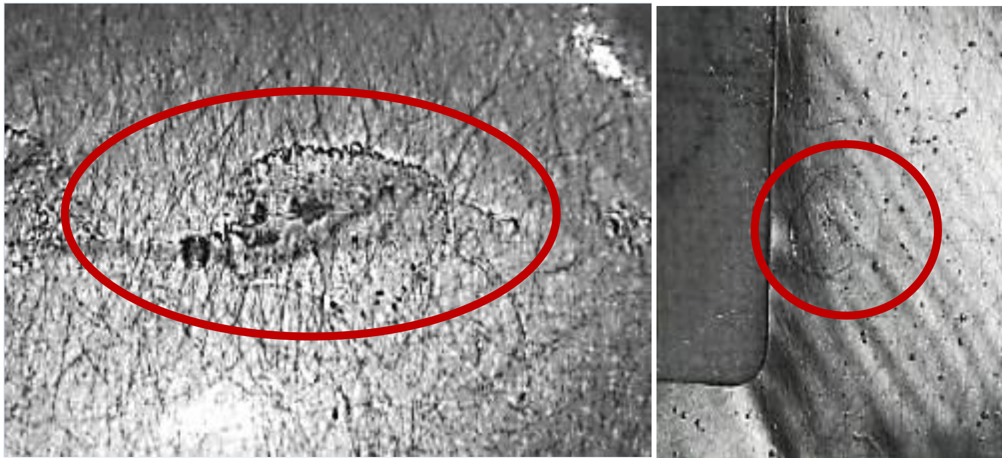

Figura 3.3 – Ejemplos de defecto de ojo de pescado.

3.3 Defecto de Rasgado

Este tipo de defecto también es causado por la manipulación del electrodo durante el proceso de ensamble. Al igual que el *defecto de ojo de pescado* también se trata de geometrías irregulares, de tamaños y localizaciones variables en el área interna del electrodo y en la superficie del gel expuesta al aire, pero a diferencia de éste, los defectos de rasgado no son causados por gel faltante sino por gel desplazado dentro del electrodo. Un ejemplo de este tipo de defecto puede observarse en la figura 3.4.

Figura 3.4 – Ejemplo de defecto de rasgado.

3.4 Defecto de Aluminio Expuesto

Este defecto es uno de los más sencillos de identificar, es causado en el proceso de ensamble al unir la placa de aluminio con la placa de gel, ya sea porque la placa de gel es más pequeña que la de aluminio, o porque no fueron alineadas correctamente. Como su nombre lo indica este defecto se caracteriza porque el aluminio no es cubierto totalmente por el gel y se aprecia expuesto en alguna zona del electrodo. Este defecto siempre se localiza en uno o más bordes del electrodo, su geometría siempre es una línea recta (con longitud y ancho variables). La figura 3.5 muestra un ejemplo de este tipo de defecto.

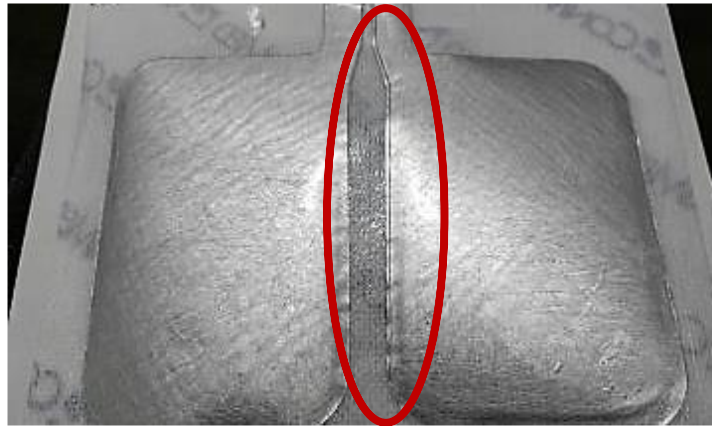

Figura 3.5 – Ejemplo de defecto de aluminio expuesto.

3.5 Defecto de Burbuja

Los defectos de burbujas son generados durante la fabricación de la superficie de gel y son causados por burbujas de aire atrapadas dentro del gel, es decir, entre la placa de aluminio y la superficie de gel. Estas burbujas pueden o no exponer el aluminio ya que en ocasiones solamente se trata de espacios huecos causados por aire, pero en otras ocasiones estos espacios o burbujas se revientan creando un orificio por el cual puede apreciarse el aluminio. Su geometría es siempre circular y su tamaño puede variar. Su localización es siempre en la superficie interna del electrodo. En la figura 3.6 se puede apreciar un ejemplo de este tipo de defectos.

Figura 3.6 – Ejemplo de defecto de burbuja.

3.6 Defecto de Contaminación

Los defectos de contaminación son generados durante la fabricación de la superficie de gel, cualquier objeto de material distinto al gel, que llegue a adherirse a esta superficie puede causar este problema. En ocasiones el material se encuentra sobre la superficie, pero también puede darse el caso, de que el objeto este atrapado dentro de la capa de gel. En ambas situaciones, esto representa un riesgo al momento de utilizar el electrodo. Su geometría es irregular y se localizan en la superficie interna del electrodo. En la figura 3.7 se puede apreciar un ejemplo de este tipo de defectos.

Figura 3.7 – Ejemplo de defecto de contaminación.

IV. SISTEMA DE ADQUISICIÓN DE IMÁGENES

Durante el desarrollo del presente trabajo se realizaron numerosas pruebas con el fin de identificar los elementos que en conjunto produjeran las imágenes con mejores características para ser procesadas posteriormente por el algoritmo de detección. Es bien sabido que, en un sistema de visión, el utilizar los elementos correctos en la etapa de la adquisición de imágenes puede producir mejores resultados en la etapa de procesamiento. Tal como se mencionó en el Capítulo 2, se pretende obtener imágenes que maximicen el contraste de los rasgos de interés, en este caso, los defectos en la superficie de gel que aseguraren el mismo resultado entre diversos objetos manteniendo una escena estable que no se vea afectada por la luz ambiental. A continuación, se presenta el proceso desarrollado y las consideraciones tomadas para la selección de cada elemento de la etapa de adquisición.

4.1 Cámara

Para la selección de la cámara se compararon tres dispositivos. La figura 4.1 muestra las cámaras revisadas, mientras que la tabla 4.1 muestra un resumen de las principales características de cada cámara.

Figura 4.1 – De izquierda a derecha, cámara Unibrain Fire-i 785b, FLIR CMLN-13S2M-CS, Basler puA2500-14cm.

Ya que dos de ellas (Unibrain & FLIR) utilizan el mismo sensor, la calidad esperada en sus imágenes es la misma, podrían considerarse equivalentes de no ser por el protocolo que utilizan para comunicarse, favoreciendo este punto a la cámara Unibrain, ya que su comunicación Firewire permite adquirir imágenes de 1.2 MP (megapíxeles) hasta 30 cuadros por segundo (fps

– *frames per second*) mientras que la comunicación USB 2.0 de la cámara FLIR se limita únicamente a 18 fps, de modo que la selección se redujo a dos opciones: Basler o Unibrain. La cámara Basler tiene un sensor de 1/2.5" y el tamaño de cada elemento sensitivo es cuadrado de 2.2 µm de lado, su principal ventaja está dada por su resolución de 5 megapixeles, comparada con 1.2 megapixeles de la cámara Unibrain. Además, puede manejar velocidades de hasta 14 fps, que representa la mitad de las velocidades máximas alcanzadas por la cámara Unibrain.

Tabla 4.1 – Comparación de características de las cámaras evaluadas.

	Unibrain	FLIR	Basler
Modelo	Fire-i 785b	CMLN-13S2M-CS	puA2500-14cm
Sensor	1/3″ CCD Sony ICX-445AL	1/3" CCD Sony ICX445	1/2.5" CMOS ON Semiconductor MT9P031
Tipo sensor	Monocromático	Monocromático	Color
Comunicación	Firewire	USB 2.0	USB 3.0
Resolución	1.2 megapíxeles (1280 x 960)	1.2 megapíxeles (1296 x 964)	5 megapíxeles (2592 x 1944)
Tamaño del elemento sensitivo	3.75µm x 3.75µm	3.75µm x 3.75µm	2.2µm x 2.2µm
Velocidad de captura	30 fps	18 fps	14 fps
Montaje de lentes	C	CS	CS

Dado que algunos de los defectos descritos en el Capítulo 3 pueden tener dimensiones pequeñas, entre mayor resolución tenga la cámara empleada, se podrán detectar defectos de menor tamaño. Esto nos llevó a inclinarnos por la cámara Basler. Por cuestiones relacionadas al tiempo de entrega de la cámara Basler, esta no estuvo disponible en etapas tempranas del

desarrollo del presente trabajo, razón por la cual, algunas de las pruebas descritas en las siguientes secciones (pruebas para la selección de lente e iluminación), se realizaron con la cámara Unibrain, pero toda las imágenes que se utilizaron para el desarrollo del algoritmo, así como para la obtención de los resultados, fueron obtenidas con el sistema final, incluyendo la cámara Basler.

4.2 Lente

Para la selección de la lente se evaluaron dos marcas distintas y 5 distancias focales. Todas las lentes evaluadas fueron de distancia focal fija. Por un lado, se tienen las lentes del fabricante *Edmund Optics*, del tipo Doble-Gauss con montaje tipo C y, por el otro, las lentes del fabricante *Evetar* con montaje tipo S. Las distancias focales consideradas fueron 75mm, 50mm, 35mm, 25mm y 18mm. La primera consideración para seleccionar la lente adecuada para la aplicación y compatible con la cámara Basler puA2500-14cm, para ello se consideraron tres características:

- El montaje debe ser tipo CS, esto no es inconveniente para ninguna de las lentes evaluadas, ya que son tipo C y S, y ambas pueden ser montadas en la cámara por medio de un adaptador.
- Deben soportar en tamaño del sensor, en este caso un sensor 1/2.5” (5.76mm x 4.29mm, 7.18mm diagonal).
- La lente no debe generar distorsión al trabajar con imágenes de alta resolución, es decir a 5 MP.

La siguiente consideración fue la distancia de trabajo necesaria para generar un campo de visión lo suficientemente grande para abarcar la superficie total del electrodo de dispersión. Incluso cuando una lente puede otorgar resultados muy buenos a determinadas distancias, si se trabaja fuera de las distancias para las cuales fue fabricado se puede observar distorsión en la imagen adquirida. La figura 4.2 muestra imágenes obtenidas con 5 lentes diferentes a la misma distancia de trabajo. Es fácil percatarse que, dependiendo de la longitud focal, se debe seleccionar la distancia de trabajo o viceversa.

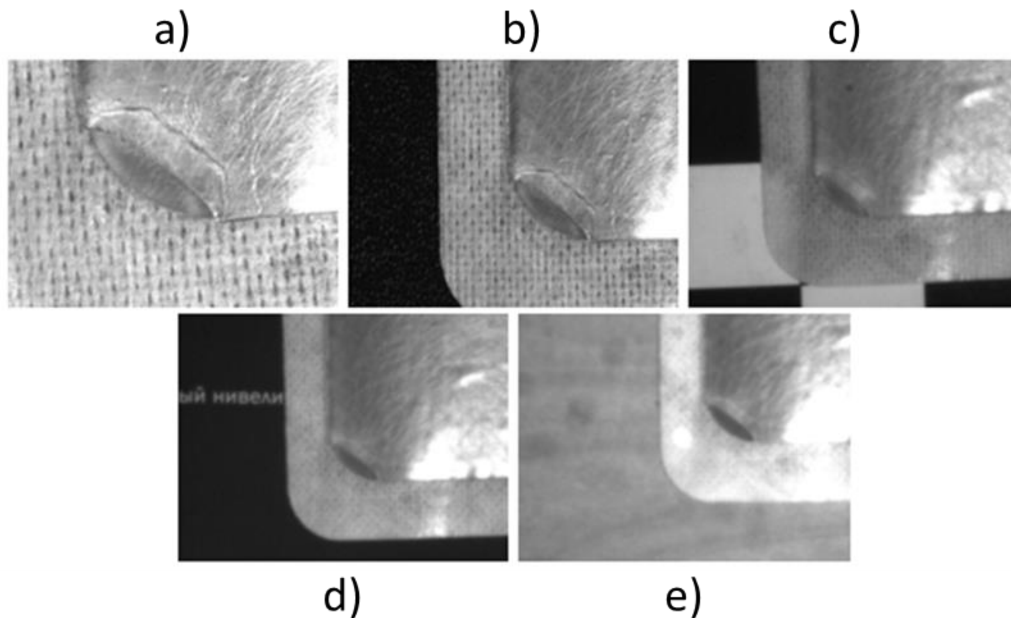

Figura 4.2 – Imágenes obtenidas a una distancia de trabajo de 60 cm con lentes de a) 75 mm, b) 50 mm, c) 35 mm, d) 25 mm y e) 18 mm.

Este fue un factor importante, ya que el sistema fue montado en una estructura de aluminio extruido que permitía montar la cámara de forma paralela al electrodo, y variar su altura respecto a este, desde 10 cm hasta aproximadamente 1 m. Los dos elementos que influyen en el campo de visión, son la distancia de trabajo y el ángulo de visión, dado que las lentes consideradas, son de longitud focal fija y, por lo tanto, ángulo de visión fijo. La única variable restante es la distancia de trabajo, de modo que el parámetro a calcular fue la distancia de trabajo.

Considerando que el ángulo de visión θ está dado por

$$\theta = 2\tan^{-1}\left(\frac{h}{2f}\right) \tag{4.1}$$

Donde h corresponde al tamaño del sensor y f a la distancia focal de la lente. Luego podemos calcular el mismo ángulo del campo de visión pero partiendo de la distancia de trabajo D y la longitud H del campo de visión, tal como lo muestra en la siguiente ecuación:

$$\theta = 2\tan^{-1}\left(\frac{H}{2D}\right) \tag{4.2}$$

Igualando las ecuaciones 4.2 y 4.1, y simplificando un poco, tenemos que:

$$\frac{h}{f} = \frac{H}{D} \tag{4.3}$$

El tamaño del sensor *h* está definido por la cámara seleccionada, 5.7mm x 4.3mm. La distancia focal *f* es una característica conocida de las lentes evaluadas (75mm, 50mm, 35mm, 25mm y 18mm). La longitud *H* del campo de visión es una restricción física, debe ser una distancia tal que permita que el electrodo de dispersión sea captado en su totalidad por la cámara. Dado que el tamaño del electrodo es 100 mm x 65 mm, se considera una longitud *H* de 105 mm, dando 5 mm de margen en la imagen. Si tomamos la ecuación 4.3 y despejamos ***D,*** podemos obtener la distancia de trabajo a partir de:

$$D = \frac{Hf}{h} \tag{4.4}$$

Dado que *H* y *h* serán constantes para todas las lentes, podemos reescribir la Ec. 4.4 de la siguiente forma:

$$D = \frac{(105\ mm)f}{(5.7\ mm)} \approx 18.4\ f \tag{4.5}$$

La tabla 4.2 muestra un resumen de las características tomadas en cuenta para la selección de la lente. Una de estas características es la distancia de trabajo necesaria para cada lente, obtenidas a partir de la Ec. 4.5, todas las unidades están expresadas en milímetros.

La distancia más adecuada para montar la cámara es de 644mm, considerando la estructura sobre la que se realizaron las pruebas. Esto reduce las opciones a los lentes de 35 mm, M12B3525M12 y #54-689. En la figura 4.3 se pueden observar ambas lentes montadas en la cámara Basler.

Tabla 4.2 – Comparación de lentas y sus distancias de trabajo.

Numero de parte	Fabricante	Longitud Focal	Distancia de trabajo	Tamaño de sensor	Tipo de Montaje	Distorsión a 5MP
#54-691	Edmund Optics	75	1380	21.6	C	Si
M12B5025-WM12	Evetar	50	920	8	S	No
#54-690	Edmund Optics	50	920	21.6	C	Si
M12B3525-WM12	Evetar	35	644	8	S	No
#54-689	Edmund Optics	35	644	16	C	Si
M12B2524-WM12	Evetar	25	460	8	S	No
#55-326	Edmund Optics	25	460	14.3	C	Si
#54-857	Edmund Optics	18	324	14.3	C	Si

Figura 4.3 – Izquierda, lente #54-689. Derecha, lente M12B3525M12.

El factor decisivo entre estas dos lentes, fue la calidad de la imagen de alta resolución. La figura 4.4 muestra las mismas escenas capturadas con ambas lentes, se puede apreciar que las imágenes capturadas con la lente #54-689, tiene menos nitidez y definición que las adquiridas con su contraparte. De modo que la lente seleccionada fue la Evetar M12B3525M12.

Figura 4.4 – Izquierda, imágenes obtenidas con la lente #54-689. Derecha, imágenes obtenidas con la lente M12B3525M12.

Considerando la cámara seleccionada, cuya resolución es de 2592 x 1944 pixeles y la lente utilizada, cuya longitud focal es de 35 mm, se tiene un campo de visión de 10.5cm x 7.9cm, se tiene entonces que cada pixel corresponde a 40.5 µm de la escena real, lo cual permite captar detalles de proporciones micrométricas.

4.3 Iluminación

El tipo de superficie de los electrodos de dispersión, es una superficie especular que, como se mencionó en el Capítulo 2, se caracteriza por reflejar la luz que incide sobre su superficie de manera puntual, es decir, el ángulo de reflexión es igual que al ángulo de incidencia. Las iluminaciones de campo brillante fueron las primeras en verificarse. Se utilizó una lámpara de anillo ubicada enfrente del electrodo de dispersión (detrás de la cámara). La figura 4.5 muestra la lámpara de anillo utilizada (izquierda) y el efecto generado en el electrodo de dispersión (derecha).

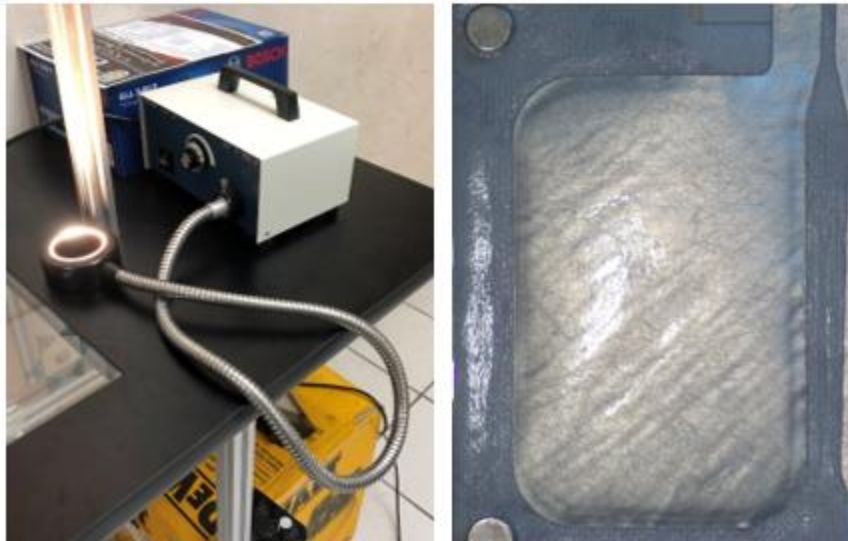

Figura 4.5 – Iluminación de anillo, imagen adquirida con iluminación de campo brillante.

Como era de esperarse, la luz de campo brillante produce fuertes reflejos en la superficie especular, esta iluminación al no ser uniforme crearía problemas en la etapa de procesamiento. La siguiente prueba de iluminación se realizó utilizando un proyector como fuente de

iluminación, con el propósito de proyectar patrones de luz sobre el electrodo de dispersión. La prueba pretendía observar las deformaciones en los patrones proyectados al incidir sobre deformaciones (defectos) en la superficie de gel. En la figura 4.6 se pueden observar los patrones utilizados para realizar las pruebas, así como el montaje del proyector.

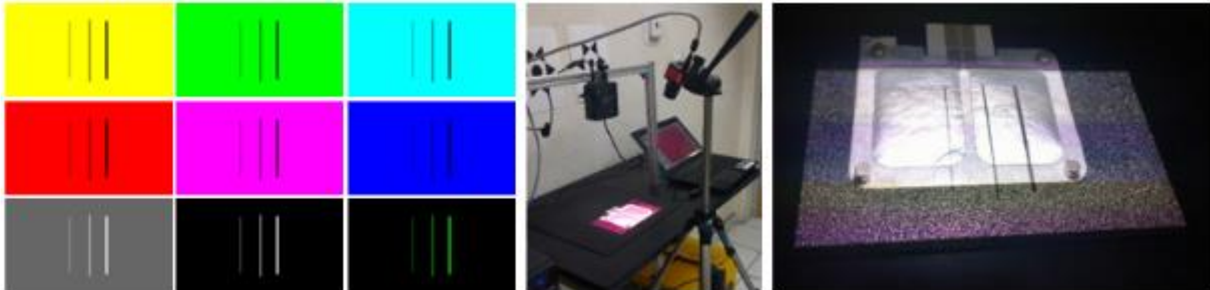

Figura 4.6 – De izquierda a derecha, patrones de iluminación utilizados, montaje de proyector y cámara, proyección de patrón sobre electrodo de dispersión.

Las imágenes utilizadas para evaluar esta iluminación, fueron adquiridas con un electrodo de dispersión, con un defecto evidente. En la figura 4.7 se puede apreciar que, pese a la evidente deformación en la superficie de gel, las líneas proyectadas no reflejaban dicha deformación, además de que la superficie de aluminio producía los mismos reflejos puntuales que se presentaban en la iluminación de campo brillante, de modo que se descartó la iluminación estructurada por medio de proyector.

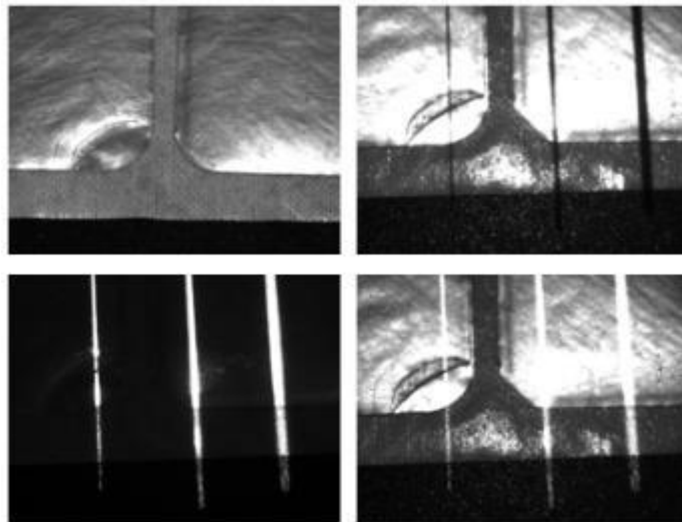

Figura 4.7 – Imágenes adquiridas con diferentes patrones de iluminación.

La tercer prueba se realizó con una fuente de iluminación difusa, utilizando un ángulo bajo. La ventaja de la iluminación difusa es que la luz no incide en un solo ángulo, por lo tanto, las reflexiones tampoco serán en un solo ángulo, evitando de esta manera puntos muy luminosos en el aluminio. El problema con esta prueba radicó en que la fuente de iluminación no era capaz de cubrir totalmente el área del electrodo, de modo que la zona que quedaba más alejada de la fuente podía percibirse oscura, mientras que la región más cercana estaba totalmente iluminada. La figura 4.8 muestra la fuente de iluminación y los resultados obtenidos con la misma.

Figura 4.8 – Iluminación difusa, lámpara de luz difusa e imagen adquirida con iluminación difusa.

Después se realizaron pruebas basadas en la técnica de campo oscuro, tal como se menciona en la Sección 2.1.4. Esta técnica utiliza fuentes puntuales de luz, con un ángulo bajo, de tal forma que las reflexiones de dicha fuente no puedan ser captadas de manera directa por la cámara. Además, con el objetivo de facilitar el proceso de segmentación, se combinó la iluminación de campo oscuro con la iluminación de fondo. Como ya se mencionó previamente la iluminación de fondo por sí sola no es de gran ayuda, ya que el aluminio no permite que la luz lo atraviese y, por lo tanto, dicha iluminación no tiene efecto alguno sobre la superficie de gel, pero en este caso el propósito de dicha iluminación no es para resaltar los rasgos en la superficie de gel, sino para separar la superficie del electrodo de la superficie del papel adhesivo que lo contiene. En la figura 4.9 puede apreciarse el área que se desea resaltar con cada tipo de iluminación.

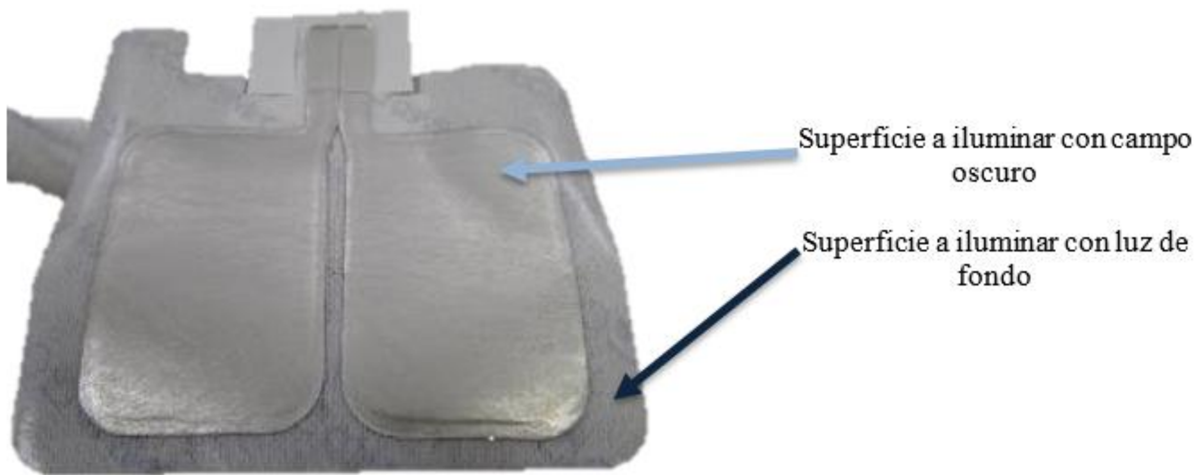

Figura 4.9 – Áreas a resaltar con cada tipo de iluminación.

De modo que la iluminación de campo oscuro es la que se enfocará en resaltar los rasgos de la superficie de gel. Para esto se realizaron 3 pruebas, la primera de ellas contemplaba dos fuentes de luz LED a una distancia muy corta del electrodo de alrededor de 10cm. La figura 4.10 muestra la manera en que se posicionan las luces de campo oscuro y la luz de fondo.

Figura 4.10 – Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo.

Como puede apreciarse en la imagen previa las fuentes de luz fueron colocadas de forma perpendicular, esto con la idea de poder resaltar tanto las imperfecciones verticales como horizontales. El problema con esta iluminación es que los rayos de luz que inciden más cerca de la lámpara, lo hacen con un ángulo mayor que los que inciden en la parte más lejana, de modo que pueden apreciarse zonas que reflejan luz tal como si se tratase de una iluminación de campo brillante. Esto se ilustra en la figura 4.11.

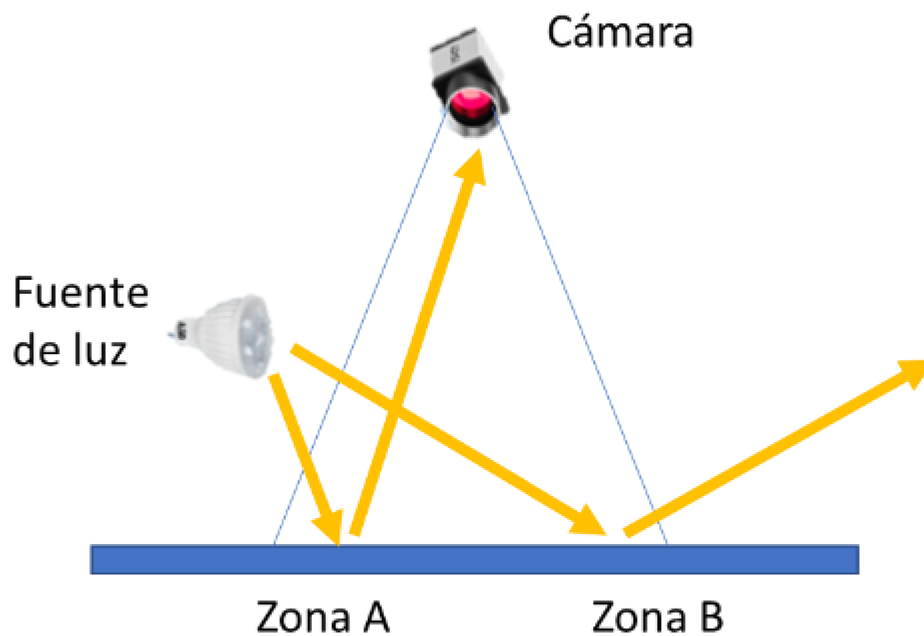

Figura 4.11 – Iluminación de campo oscuro a distancias cortas. La zona A aparece más iluminada para la cámara que la zona B.

La figura 4.12 muestra el efecto antes descrito en un electrodo capturado con esta iluminación. Si bien la iluminación de campo oscuro no resultó como se deseaba, la iluminación de fondo cumple totalmente su objetivo ya que permite segmentar sin problema alguno el electrodo del papel adhesivo.

Figura 4.12 – Imagen capturada utilizando Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo.

En la siguiente prueba se realizaron algunos cambios para intentar mitigar este efecto, primero se optó por colocar las fuentes de luz, una frente a la otra en lados opuestos del electrodo, para generar una iluminación simétrica. Además, las fuentes de luz se colocaron más lejos del electrodo (alrededor de 35 cm). La figura 4.13 muestra el acomodo de las fuentes de luz para esta prueba.

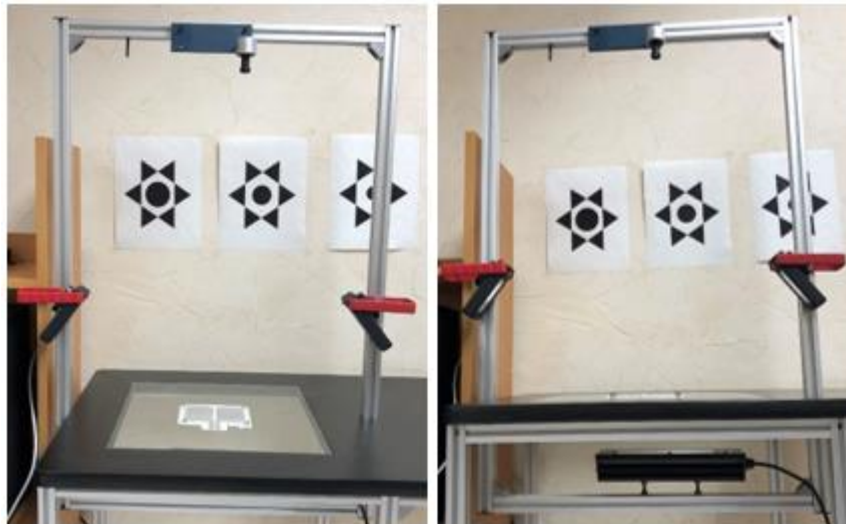

Figura 4.13 – Iluminación de campo oscuro, dos fuentes de luz simétricas a 35 cm y utilizando luz de fondo.

La figura 4.14 muestra el resultado obtenido con esta última prueba, podemos observar que los efectos de las zonas excesivamente brillantes se redujeron considerablemente, pero la iluminación sigue sin ser uniforme, las zonas cercanas a las fuentes de luz siguen siendo notorias.

Figura 4.14 – Imagen capturada utilizando Iluminación de campo oscuro, dos fuentes de luz perpendiculares a 10 cm y utilizando luz de fondo.

El problema en las pruebas anteriores radicaba en que los ángulos de incidencia de la luz no eran iguales en toda la superficie del electrodo, una forma de solucionar esto es utilizar una fuente de luz más potente, de modo que pueda alejarse más del electrodo pero seguir iluminándolo. Cada vez que la fuente de luz se aleja, los ángulos de la luz incidente en el electrodo serán más parecidos e idealmente si la fuente de luz se aleja hasta el infinito, los rayos que incidan en dos puntos de la superficie del electrodo serán iguales, aun cuando estos puntos se ubiquen en dos extremos opuestos. Esto se ilustra en la figura 4.15.

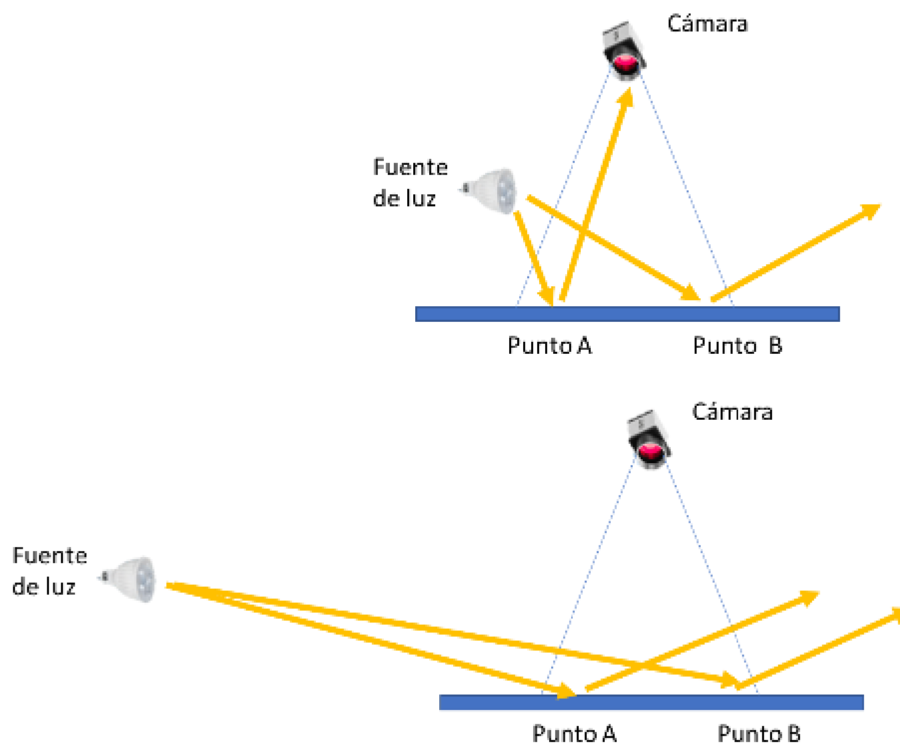

Figura 4.15 – Efecto de variar la distancia de la fuente de luz en una escena.

Considerando esto, la última prueba se realizó reemplazando las 2 fuentes de iluminación por una fuente de mayor potencia con una intensidad luminosa de 370 lúmenes y temperatura de color de 3000K. Esta fuente de iluminación se colocó con un ángulo bajo, el eje de la lámpara se colocó de forma paralela al plano del electrodo a una distancia de 80 cm del electrodo. En la figura 4.16 podemos observar los resultados obtenidos con esta configuración.

Figura 4.16 – Imagen capturada utilizando iluminación de campo oscuro con una sola fuente de luz a 80 cm y utilizando luz de fondo.

En la última prueba podemos apreciar que la iluminación es completamente uniforme en toda la superficie de gel, no se observan puntos con iluminación excesiva ni reflejos puntuales. Además, el color cálido de la fuente de campo oscuro realza aún más el contraste con el papel adhesivo, que de por si era notorio gracias a la luz de fondo. De modo que la iluminación seleccionada basada en los resultados de esta prueba fue:

Iluminación de campo oscuro con una sola fuente de luz que consiste en una lámpara *7904* de *Electrixtask* con un arreglo de 8 leds, intensidad luminosa de 370 lúmenes, temperatura de color de 3000K y una potencia de 3.5 watts. Mientras que para la luz de fondo se utilizó una lámpara *SL seriesTwin 6 Watts* conectado a un controlador universal *StockerYale* de *85Khz*.

4.4 Tarjeta Embebida

La siguiente etapa en el desarrollo consistió en la selección de la plataforma embebida que sería utilizada para desarrollar el sistema de visión. Si bien existen una gran cantidad de sistemas disponibles en el mercado, la selección de la tarjeta para este proyecto estaba condicionada para los elementos previamente definidos para el sistema, ya que la tarjeta debía ser compatible con

el resto de los elementos. La iluminación o la estructura donde el sistema se desarrolló no tienen mucha injerencia en esta decisión, pero la cámara por otro lado, está totalmente ligada a la tarjeta embebida. Como se describió en la Sección 4.1, la cámara Basler puA2500-14cm tiene una interfaz de comunicación USB 3.0 y soporta una resolución de 5 MP. Por lo tanto, la tarjeta debe cumplir con los requerimientos mínimos de contar con un puerto USB 3.0 y tener suficiente capacidad de procesamiento para manipular imágenes de 5 MP. Aunado a esto, podemos hacer uso de uno de los recursos más comunes para este tipo de aplicaciones, las comparativas de desempeño o *benchmarks*. Este tipo de reportes generalmente tiene el propósito de comparar el desempeño de determinado producto, evaluando las características que cada uno ofrece, los puntos a favor para determinar su rendimiento en diversos escenarios de uso. A cada aspecto evaluado se le otorga una puntuación de modo que es fácil llegar a una conclusión, sobre que producto es mejor para determinada aplicación. En diversas fuentes se pueden encontrar comparativas de tarjetas de embebidas, también conocidas como computadoras de una sola placa (SBCs por sus siglas en inglés, *Single Board Computers*). Tal es el caso de [35] o [36], pero el presentado en [37], resulta particularmente relevante, en primera instancia porque se realiza una comparativa enfocada al desempeño de diversas SBCs utilizadas para adquirir imágenes, además porque fue realizado por Basler, que es el fabricante de la cámara que se utilizó en el presente trabajo. En [37]se compararon seis SBCs:

- Nvidia Jetson TK1
- Nvidia Jetson TX1
- Qualcomm DragonBoard 410c
- Hardkernel Odroid XU4
- Raspberry Pi 3
- Raspberry Pi 2

Se evaluaron la capacidad de procesamiento, así como el desempeñó de los protocolos de comunicación USB 3.0 y GigE, concluyendo que la tarjeta Jetson TX1 de Nvidia tuvo el mejor resultado en procesamiento y en manejo del protocolo USB3.0, ambos aspectos críticos para

este proyecto. Por ello se decidió trabajar con dicha tarjeta, pero al momento de realizar este trabajo, la nueva generación Jetson TX2 se encontraba disponible, al verificar las especificaciones de ambas versiones (las especificaciones pueden observarse en la figura 4.17) se concluyó que la versión 2, supera las especificaciones de su predecesora, de modo que la tarjeta que se utilizó para este desarrollo fue la Jetson TX2 [38].

	NVIDIA Jetson TX1	NVIDIA Jetson TX2
CPU	ARM Cortex-A57 (quad-core) @ 1.73GHz	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	256-core Maxwell @ 998MHz	256-core Pascal @ 1300MHz
Memory	4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Storage	16GB eMMC 5.1	32GB eMMC 5.1
Encoder*	4Kp30, (2x) 1080p60	4Kp60, (3x) 4Kp30, (8x) 1080p30
Decoder*	4Kp60, (4x) 1080p60	(2x) 4Kp60
Camera†	12 lanes MIPI CSI-2 | 1.5 Gb/s per lane | 1400 megapixels/sec ISP	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Display	2x HDMI 2.0 / DP 1.2 / eDP 1.2 | 2x MIPI DSI	
Wireless	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.0	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet	
USB	USB 3.0 + USB 2.0	
PCIe	Gen 2 | 1×4 + 1 x1	Gen 2 | 1×4 + 1×1 or 2×1 + 1×2
CAN	Not supported	Dual CAN bus controller
Misc I/O	UART, SPI, I2C, I2S, GPIOs	
Socket	400-pin Samtec board-to-board connector, 50x87mm	
Thermals‡	-25°C to 80°C	
Power††	10W	7.5W
Price	$299 at 1K units	$399 at 1K units

Figura 4.17 – Imagen comparativa de las tarjetas Jetson TX1 y Jetson TX2.

La tarjeta TX2, puede observarse en la figura 4.18 [38], se trata de una tarjeta embebida de alto poder computacional, tiene un consumo de potencia de 7.5 watts, cuenta con 8 Gb de memoria RAM, tiene una unidad de procesamiento gráfico (GPU por sus siglas en inglés *Graphical Processing Unit*) y un disco de estado sólido de 32 Gb, además de incorporar un puerto USB 3.0. Estas características la hacen ideal para el sistema de visión desarrollado.

Figura 4.18 – Tarjetas Jetson TX2.

La tarjeta incluye las antenas necesarias para conectarse a una red inalámbrica, un cable de alimentación, una fuente de poder, y dos cables USB requeridos para la configuración de la misma. Por defecto, la tarjeta tiene precargada una versión del sistema operativo Ubuntu 16.04, pero es necesario realizar algunas configuraciones antes de poder utilizarla.

4.4.1 Configuración del sistema operativo

La compañía Nvidia, fabricante de la tarjeta Jetson TX2, ofrece algunas opciones recomendadas de configuración para su tarjeta. Estas opciones están orientadas a sacar el máximo provecho de la misma y utilizar todos sus recursos de manera óptima. La principal opción sugerida es un kit de desarrollo de software (SDK por sus siglas en inglés, *Software Development Kit*), que sirve como base para realizar desarrollos sobre él. El SDK contiene un sistema operativo (Ubuntu), además de todos los controladores necesarios para los componentes de hardware de la tarjeta. Este SDK lleva el nombre de **JetPack,** en el momento de desarrollo de esta tesis la versión más actual de dicho SDK es la 3.3. El SDK se puede descargar directamente en la liga https://developer.nvidia.com/embedded/jetpack.

A continuación, se muestran los pasos necesarios para configurar la tarjeta Jetson TX2 para su uso, estas instrucciones fueron tomadas de [39]:

1. Se requiere una PC con sistema operativo Ubuntu 16.04 de 64 bits, para realizar el proceso. A este PC se le denominara anfitrión, en ella se descargará la versión del SDK que se desee. En este caso la versión descargada fue la 3.3.

2. Después de realizar la descarga se otorgan permisos de ejecución al archivo, esto se hace a través de la consola de comandos de la PC anfitrión usando el comando

   chmod +x JetPack-3.3.run

3. Luego se ejecuta el programa de instalación del SDK con el comando

   ./JetPack-3.3run

   En ese aparecerá la interfaz de instalación

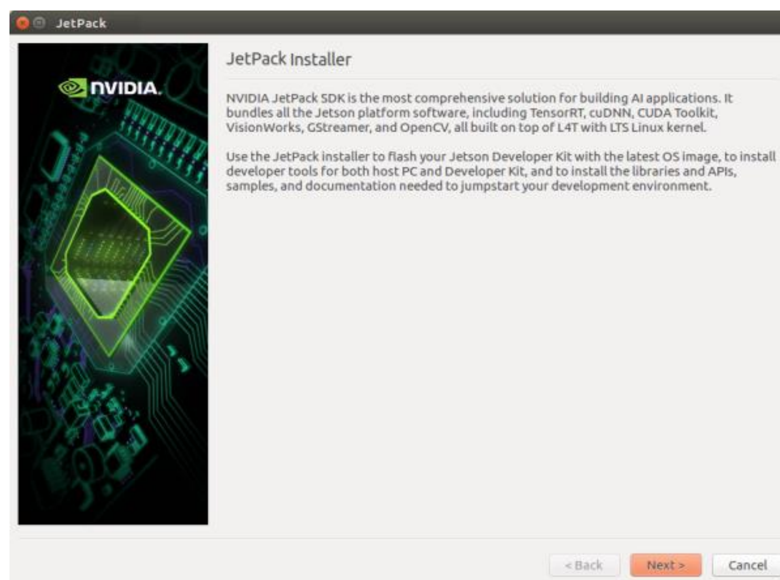

Figura 4.19 – Interfaz de instalación de JetPack-3.3. Pantalla 1.

4. La siguiente pantalla muestra las rutas en que se instalará el SDK, así como algunos avisos de privacidad, la recomendación de Nvidia, es no cambiar las rutas de instalación, ya que esto puede generar problemas en los pasos siguientes.

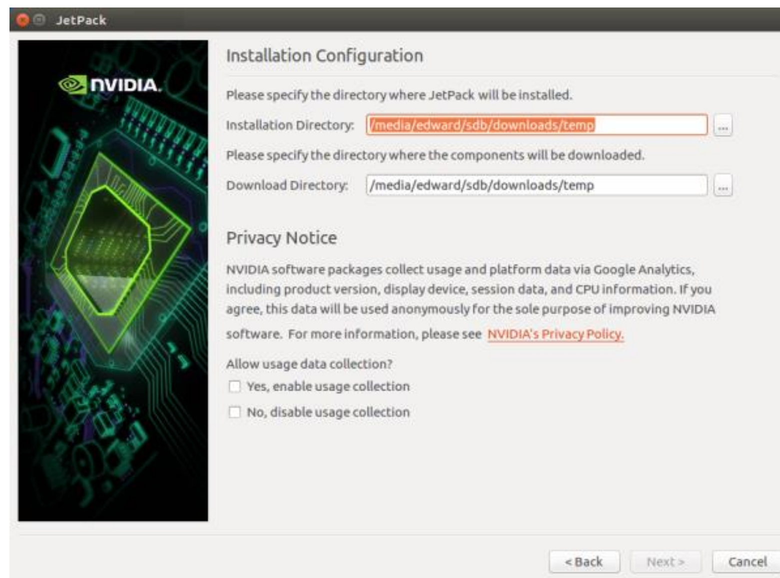

Figura 4.20 – Interfaz de instalación de JetPack-3.3. Pantalla 2.

5. En el paso siguiente se debe seleccionar la tarjeta en la cual se instalará el SDK, en este caso la Jetson TX2.

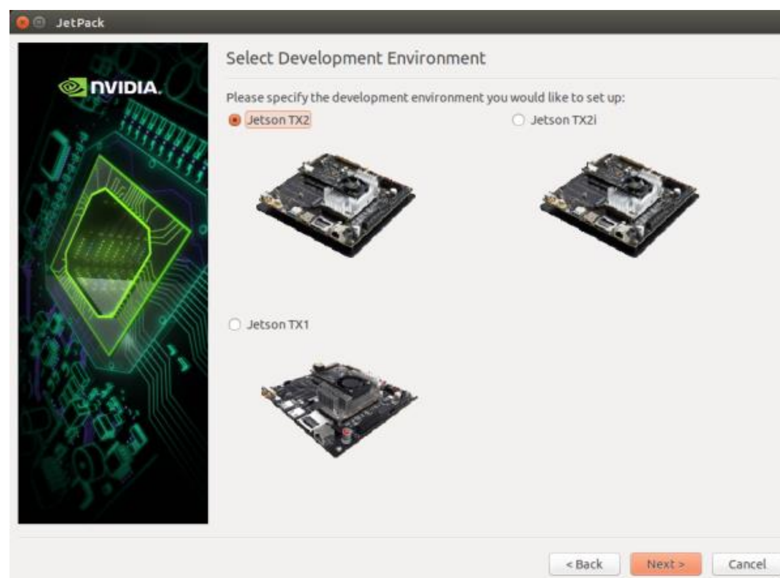

Figura 4.21 – Interfaz de instalación de JetPack-3.3. Pantalla 3.

6. Una vez seleccionada la tarjeta, el programa solicitará credenciales con permisos de súper usuario, para completar el proceso.

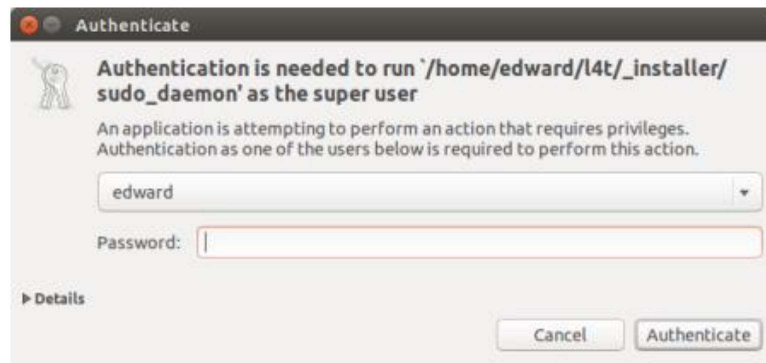

Figura 4.22 – Interfaz de instalación de JetPack-3.3. Pantalla 4.

7. En este paso aparecerá una pantalla indicando todos los componentes de software que serán instalados, se pueden agregar o eliminar según se desee, nuevamente la sugerencia de Nvidia es no realizar cambios e instalar todos los paquetes seleccionados por defecto.

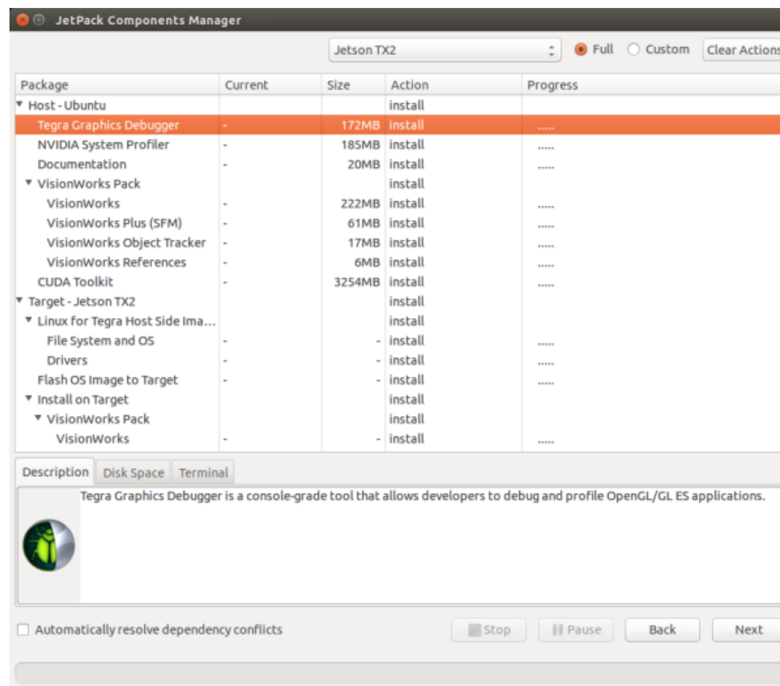

Figura 4.23 – Interfaz de instalación de JetPack-3.3. Pantalla 5.

8. En la siguiente pantalla aparecerán los acuerdos de licencias, para los diversos módulos a instalar. Debemos leer y aceptarlos de manera individual para proseguir.

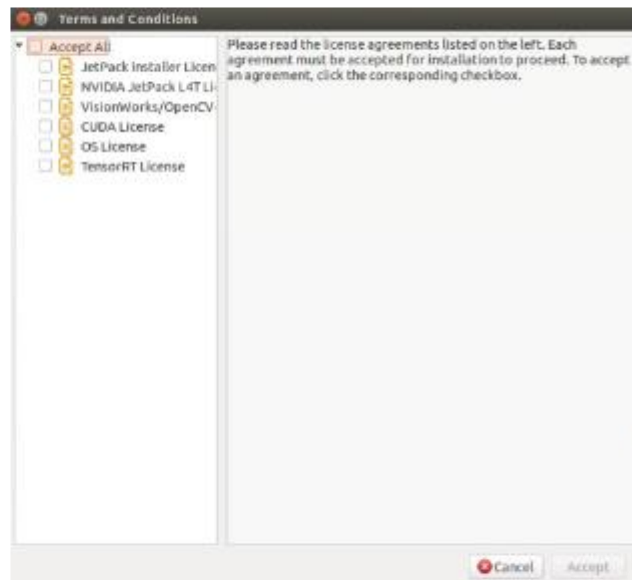

Figura 4.24 – Interfaz de instalación de JetPack-3.3. Pantalla 6.

9. Después comenzará la instalación de los componentes necesarios en el anfitrión, una barra mostrará el progreso de la instalación y una vez que termine se habilitará el botón ***next***.

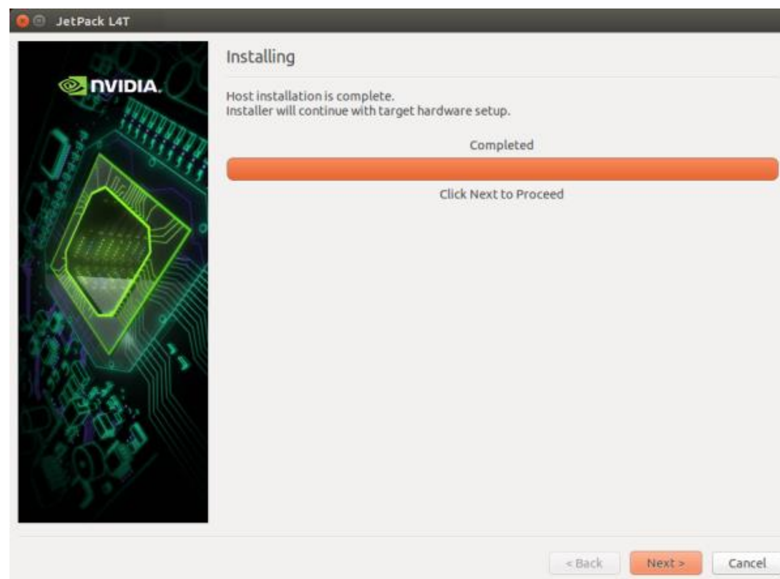

Figura 4.25 – Interfaz de instalación de JetPack-3.3. Pantalla 7.

10. En este paso, se conectó la PC anfitrión a un ruteador con conexión a internet, así mismo, la tarjeta Jetson se conectó al mismo ruteador a través de un cable de red.

En la pantalla que se muestra en la figura 4.26, se debe seleccionar la opción 1, que refleja la conexión realizada.

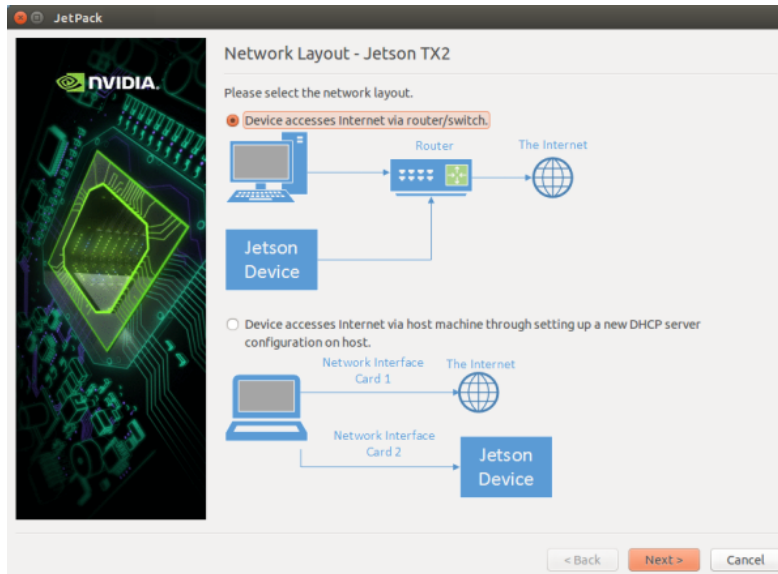

Figura 4.26 – Interfaz de instalación de JetPack-3.3. Pantalla 8.

11. Después se selecciona el puerto utilizado para conectarse al ruteador indicado en el paso previo.

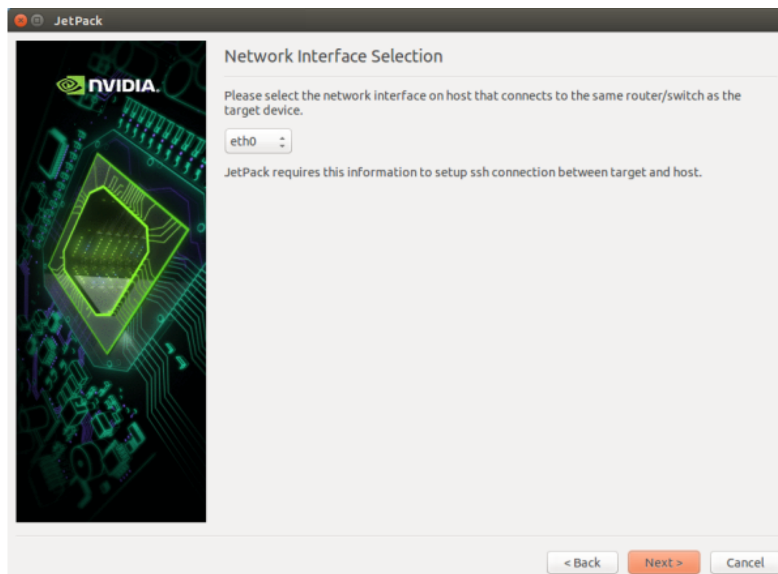

Figura 4.27 – Interfaz de instalación de JetPack-3.3. Pantalla 9.

12. La siguiente pantalla (figura 4.28), nos indicará que la instalación en el anfitrión ha terminado y está por comenzar la instalación en la tarjeta Jetson. Para poder iniciar nos solicitará que pongamos la tarjeta en modo forzado de recuperación USB, esto se logra desconectando la tarjeta de la alimentación y conectando el cable USB – micro USB entre la PC anfitrión y la tarjeta Jetson. Después de conectar el cable, se debe encender la tarjeta y una vez que haya encendido, se presionará el botón que está marcado como **Rec** (recuperación, *recovery* en inglés) y mientras se mantiene presionado se presiona momentáneamente el botón marcado como **Reset** o reinicio. Después de realizar ésto, se esperará dos segundos para liberar el botón **Rec**, en el equipo anfitrión se utiliza el comando **lsusb** para ver todos los dispositivos USBs conectados. Si los pasos anteriores fueron correctos debe aparecer en la lista algún dispositivo identificado como "Nvidia Corp". De ser así, la instalación puede proseguir al presionar *enter* en la pantalla de instalación.

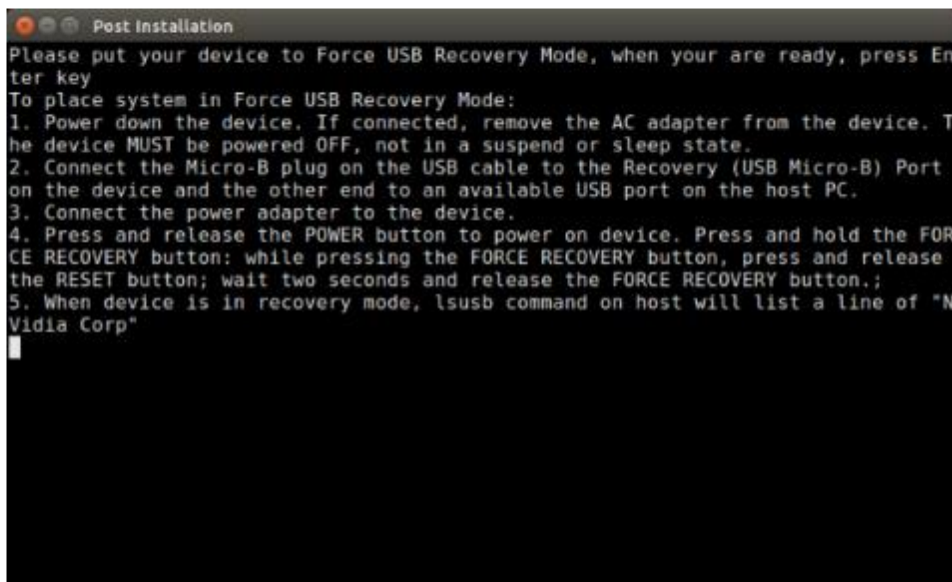

Figura 4.28 – Interfaz de instalación de JetPack-3.3. Pantalla 10.

13. El siguiente paso (figura 4.29) mostrará los paquetes que serán instalados en la tarjeta Jetson.

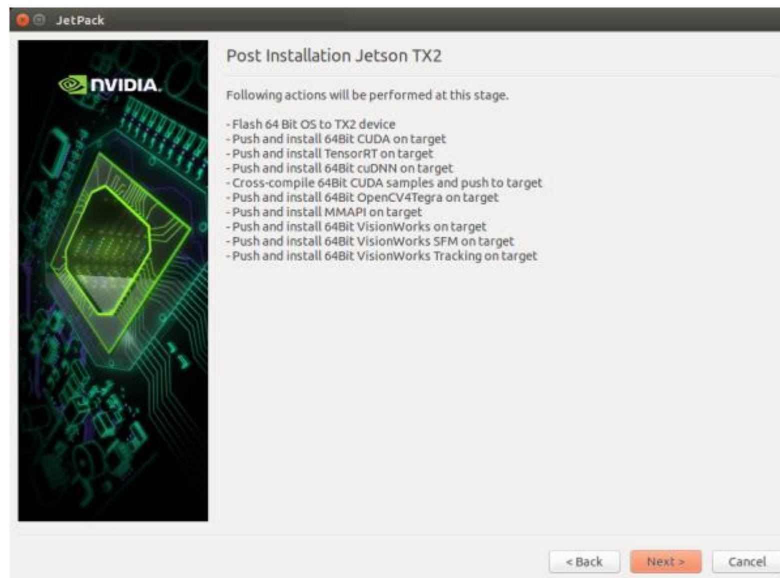

Figura 4.29 – Interfaz de instalación de JetPack-3.3. Pantalla 11.

14. La última pantalla permite eliminar los archivos temporales utilizados para la instalación (figura 4.30).

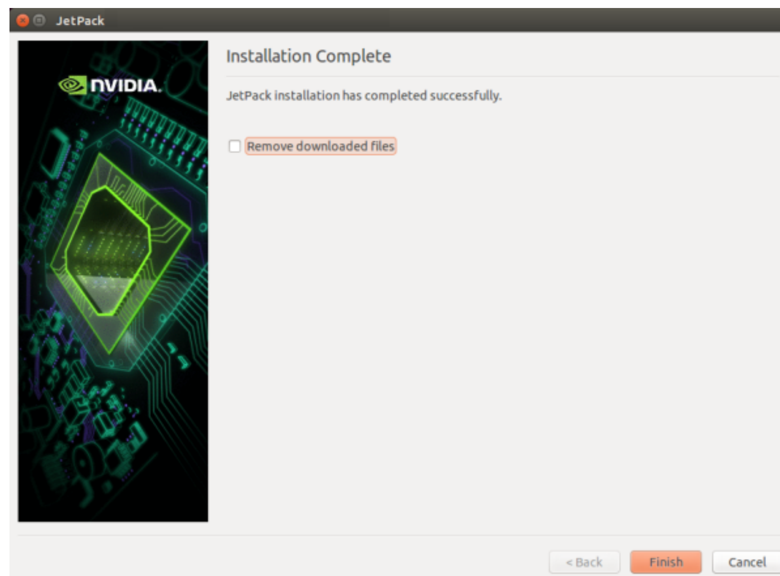

Figura 4.30 – Interfaz de instalación de JetPack-3.3. Pantalla 12.

Después de completar la instalación se reinicia la tarjeta Jetson y se enciende de manera normal. El sistema operativo Ubuntu 16.04 está totalmente configurado y puede utilizarse de forma normal. Solamente se necesita un monitor, un teclado y un mouse para poder interactuar con la

tarjeta. Es importante tener en cuenta que después de instalar el SDK en la tarjeta Jetson se genera un usuario llamado **nvidia**, este usuario será el que se empleará para trabajar en la tarjeta. En todas las secciones que se muestran a continuación en donde se utiliza el comando **sudo,** la terminal solicitará credenciales que tengan permisos de súper usuario, se deben utilizar estas credenciales:

- Nombre de usuario: **nvidia**
- Contraseña: **nvidia.**

4.4.2 Configuración de OpenCV

Una vez que el sistema operativo y los controladores de la tarjeta se configuraron a través del SDK de Nvidia fue necesario configurar el ambiente de desarrollo de software. Se instaló el intérprete de Python, que fue lenguaje seleccionado para el desarrollo de las aplicaciones del proyecto, así como la librería de OpenCV. Esta es una librería de visión por computadora de código abierto, altamente optimizada y multiplataforma. Debido a que la tarjeta Jetson no estuvo disponible desde los inicios del proyecto, el código y las pruebas se realzaron en distintas computadoras, el ambiente de desarrollo debía ser compatible entre todas, permitiendo así que el código pudiera ser modificado y ejecutado desde cualquiera de ellas. La tabla 4.3 muestra los diferentes entornos utilizados, de modo que puede asumirse que el código generado funciona, en cualquiera de las versiones de Python y OpenCV abajo mencionadas.

Tabla 4.3 – Entornos de desarrollo utilizados.

	SO	Python	OpenCV	Procesador	GPU	RAM
PC 1	Ubuntu 18.04	3.6.6	4.0.0-alpha	AMD A12-9700P	AMD Radeon R7	16 GB
PC 2	Windows 10	3.6.5	3.4.4	Intel I7-7700HQ	NVIDIA Quadro M2200	32 GB
Jetson TX2	Ubuntu 16.04	3.5.2	4.0.1	NVIDIA Denver 2 y ARM Cortex-A57 MPCore	NVIDIA Pascal	8 GB

A continuación, se describirán los pasos necesarios para instalar Python y OpenCV en la tarjeta Jetson. Las instalaciones en PC, Windows o Ubuntu no serán analizadas, pues el enfoque de este trabajo es sobre la tarjeta embebida. El procedimiento mencionado a continuación está basado en [40] y fue ejecutado en la línea de comandos de la tarjeta Jetson.

1. El primer paso es asegurar que el sistema se encuentra actualizado, mediante los comandos:

   sudo apt-get update

   sudo apt-get upgrade

2. Después se instalan algunas herramientas de desarrollo.

   sudo apt-get install build-essential cmake pkg-config

3. Se instalan las librerías necesarias para decodificar imágenes de disco, así como de video.

   sudo apt-get install libjpeg-dev libpng-dev libtiff-dev

   sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev

   sudo apt-get install libxvidcore-dev libx264-dev

4. Luego se instalan las librerías encargadas de generar gráficos y manejar matrices.

sudo apt-get install libgtk-3-dev

sudo apt-get install libatlas-base-dev gfortran

5. El siguiente paso es instalar Python, en este caso 3.5.

   sudo apt-get install python3.5-dev

6. Después se deben descargar los archivos fuentes de OpenCV, que contiene la mayor parte de la librería. Estos archivos son descargados como código fuente, ya que se compilarán con algunas opciones particulares para tomar la mayor ventaja posible del hardware de la tarjeta Jetson. También se descarga OpenCV **contrib**, que es un complemento de la librería de OpenCV en el que se incluyen otros algoritmos que no están incluidos en OpenCV.

   cd ~

   wget -O opencv.zip https://github.com/opencv/opencv/archive/4.0.1.zip

   wget -O opencv_contrib.zip https://github.com/opencv/opencv_contrib/archive/4.0.1.zip

   unzip opencv.zip

   unzip opencv_contrib.zip

7. Opcionalmente se puede instalar **pip**, que es un manejador de paquetes de Python y facilitará el resto del proceso.

   wget https://bootstrap.pypa.io/get-pip.py

   sudo python3 get-pip.py

8. Una vez instalado **pip**, lo utilizaremos para instalar dos herramientas **virtualenv** y **virtualenvwrapper**, que servirán para crear y manejar los ambientes virtuales utilizados. Es una práctica común al trabajar con Python, el crear ambientes virtuales, estos ambientes permiten tener configuraciones específicas para un proyecto con librerías o dependencias instaladas en Python sin afectar su

funcionalidad en otros proyectos, ya que se asigna un ambiente virtual a cada uno y las modificaciones permanecen aisladas a cada ambiente.

sudo pip install virtualenv virtualenvwrapper

sudo rm -rf ~/get-pip.py ~/.cache/pip

9. Luego de instalar ambas herramientas deben configurarse para que se ejecuten de manera automática en el Sistema.

echo -e "\n# virtualenv and virtualenvwrapper" >> ~/.bashrc

echo "export WORKON_HOME=$HOME/.virtualenvs" >> ~/.bashrc

echo "export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3" >> ~/.bashrc

echo "source /usr/local/bin/virtualenvwrapper.sh" >> ~/.bashrc

source ~/.bashrc

10. El siguiente paso es crear el ambiente virtual que se utilizará, en este caso llamado **cv**.

mkvirtualenv cv -p python3

11. Cada vez que se desee trabajar en el ambiente virtual deberá usarse el siguiente comando, ya que OpenCV será instalado dentro del ambiente virtual, no se podrá acceder a esta librería si Python se ejecuta desde fuera de él.

workon cv

12. Después se procede a instalar **numpy**, es una librería de Python utilizada para procesamiento numérico.

pip install numpy

13. Se configuran los parámetros de la compilación, estos determinan que componentes de OpenCV serán incluidos en la librería.

mkdir build

```
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE \
    -D CMAKE_INSTALL_PREFIX=/usr/local \
    -D INSTALL_PYTHON_EXAMPLES=ON \
    -D INSTALL_C_EXAMPLES=OFF \
    -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
    -D PYTHON_EXECUTABLE=~/.virtualenvs/cv/bin/python \
    -D BUILD_EXAMPLES=ON
    -D OPENCV_ENABLE_NONFREE=ON \
    -D WITH_CUDA=ON \
    -D ENABLE_FAST_MATH=1 \
    -D CUDA_FAST_MATH=1 \
    -D WITH_CUBLAS=1
```

14. Luego se ejecuta el proceso de compilación

```
make -j4
```

15. El paso previo puede tomar más de una hora para concluir, pero una vez que termina se puede instalar OpenCV con los siguientes comandos:

```
sudo make install
sudo ldconfig
```

16. El último paso es ligar la instalación de OpenCV al ambiente virtual, utilizando los siguientes comandos

```
cd /usr/local/lib/python3.5/site-packages/cv2/python-3.5
sudo mv cv2.cpython-35m-x86_64-linux-gnu.so cv2.so
```

cd ~/.virtualenvs/cv/lib/python3.5/site-packages/

ln -s /usr/local/lib/python3.5/site-packages/cv2/python-3.5/cv2.so cv2.so

En este punto el ambiente de desarrollo se encuentra totalmente configurado y listo para usarse, si todos los pasos anteriores fueron ejecutados de forma correcta, en una terminal se pueden ejecutar los siguientes comandos para obtener la versión de OpenCV instalada

workon cv

Python

import cv2

cv.__version__

En la figura 4.31 se puede apreciar la ejecución de estos comandos en la tarjeta Jetson, como puede confirmarse que la versión de Python corresponde a la 3.5.2 y la de OpenCV a la 4.0.1.

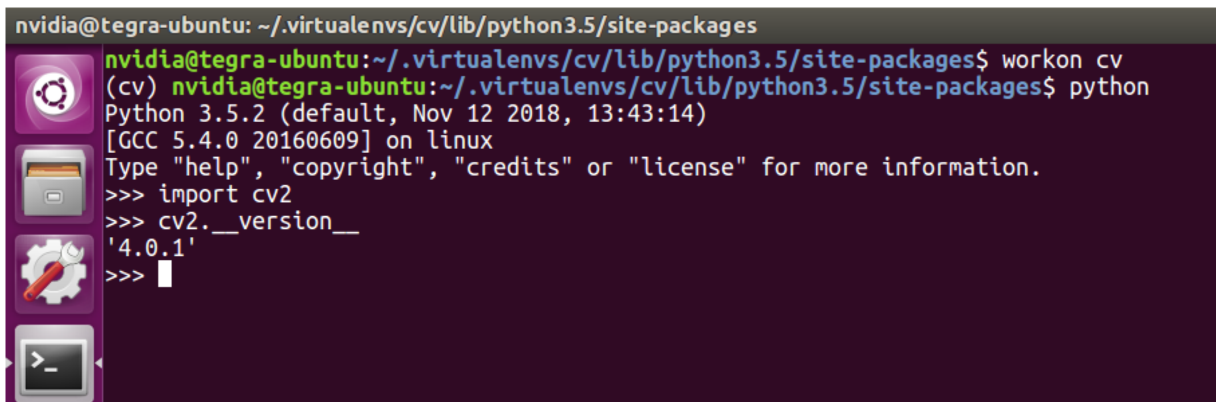

Figura 4.31 – Versión de Python y OpenCV instalada en la tarjeta Jetson TX2.

4.4.3 Configuración del procesador

Como se mención al inicio de la Sección 4.4, la tarjeta Jetson cuenta con 2 procesadores, un NVIDIA Denver 2 con 2 núcleos y un ARM Cortex-A57 MPCore con 4 núcleos, además de un procesador gráfico NVIDIA Pascal con 256 núcleos CUDA. Después de instalar el SDK JetPack, por defecto, solo el procesador ARM estará activo [41]. Esto puede verificarse al

ejecutar el comando **sudo ./tegrastats** en cualquier terminal. La figura 4.32 muestra la salida que se obtiene al usar este comando.

```
nvidia@tegra-ubuntu: ~                                                    5:03 PM
nvidia@tegra-ubuntu:~$ sudo ./tegrastats
RAM 840/7846MB (lfb 1596x4MB) CPU [0%@653,off,off,0%@653,0%@652,0%@652] EMC_FREQ 2%@665 GR3D_FREQ 7%@140 APE 150 BCPU@3
7.5C MCPU@37.5C GPU@36.5C PLL@37.5C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.4C VDD_IN 1494/1494 VDD_CPU 153/153 V
DD_GPU 153/153 VDD_SOC 306/306 VDD_WIFI 0/0 VDD_DDR 229/229
RAM 840/7846MB (lfb 1596x4MB) CPU [5%@346,off,off,2%@345,4%@345,2%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150 BCPU@
38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.3C VDD_IN 1418/1456 VDD_CPU 153/153 VDD_GP
U 153/153 VDD_SOC 306/306 VDD_WIFI 0/0 VDD_DDR 210/219
RAM 840/7846MB (lfb 1596x4MB) CPU [6%@345,off,off,2%@345,2%@343,4%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150 BCPU@
38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.5C PMIC@100C thermal@37.4C VDD_IN 1417/1443 VDD_CPU 153/153 VDD_GPU
 153/153 VDD_SOC 306/306 VDD_WIFI 0/0 VDD_DDR 191/210
RAM 840/7846MB (lfb 1596x4MB) CPU [4%@345,off,off,3%@345,3%@348,2%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150 BCPU@
38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.5C PMIC@100C thermal@37.4C VDD_IN 1417/1436 VDD_CPU 153/153 VDD_GPU
 153/153 VDD_SOC 306/306 VDD_WIFI 0/0 VDD_DDR 210/210
RAM 840/7846MB (lfb 1596x4MB) CPU [3%@345,off,off,4%@345,2%@345,2%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150 BCPU@
38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.5C PMIC@100C thermal@37.4C VDD_IN 1418/1432 VDD_CPU 153/153 VDD_GPU
 153/153 VDD_SOC 306/306 VDD_WIFI 0/0 VDD_DDR 210/210
```

Figura 4.32 – Utilización de procesadores en tarjeta Jetson TX2, configuración por defecto.

En la terminal se pueden apreciar varios indicadores del rendimiento de la tarjeta, nos centraremos en los más importantes. En primera instancia tenemos el término **RAM m/7846MB** que indica el uso de memoria en determinado instante. El termino **CPU [x0%@y0 x1%@y1 x2%@y2 x3%@y3 x4%@y4 x5%@y6]** nos indica el estatus de cada uno de los núcleos de ambos procesadores, los términos **xi** indican el porcentaje de utilización de cada núcleo, mientras que el término **yi** indica la frecuencia de reloj a la que dicho núcleo está operando en MHz. Los índices 0, 3, 4 y 5, corresponden a los 4 núcleos del procesador ARM, mientras que los índices 1 y 2 corresponden a los 2 núcleos del procesador Denver 2. Tenemos además el término **GR3D_FREQ x%@y**, en este caso, al igual que en los anteriores, **x** refiere al porcentaje de utilización, mientras que **y** indica la frecuencia de reloj, pero en este caso ambos enfocados al GPU. Basados en esto podemos confirmar en la figura 4.32, que los dos núcleos del procesador Denver 2 se encuentran apagados, mientras que los 4 núcleos del procesador ARM se encuentran trabajando a una frecuencia de 345 MHZ y el GPU a una frecuencia de 140 MHz. Si bien la tarjeta puede utilizarse con esta configuración su desempeño no sería el mejor ya que se estaría subutilizando. El SDK instalado en la tarjeta provée 5 modos de operación, cada modo utiliza una combinación de núcleos y velocidades de reloj distintas. No se revisarán los detalles de cada modo de operación, basta con mencionar que el modo 0, es el que nos permite utilizar todos los núcleos a la mayor velocidad posible. Para poder cambiar a este modo de operación se hace uso de él comando **nvpmodel -m 0**, donde el **0** corresponde al modo de

operación. Después ejecutamos nuevamente el comando **sudo ./tegrastats** para confirmar que los cambios se hayan reflejado. En la figura 4.33 podemos observar la ejecución de estos dos comandos.

```
nvidia@tegra-ubuntu: ~                                                    5:04 PM
nvidia@tegra-ubuntu:~$ sudo nvpmodel -m 0
nvidia@tegra-ubuntu:~$ sudo ./tegrastats
RAM 841/7846MB (lfb 1596x4MB) CPU [0%@345,0%@498,0%@499,0%@345,0%@345,0%@345] EMC_FREQ 3%@408 GR3D_FREQ 4%@140 APE 150
MTS fg 0% bg 0% BCPU@38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.4C VDD_IN 1609/1609 V
DD_CPU 229/229 VDD_GPU 153/153 VDD_SOC 383/383 VDD_WIFI 0/0 VDD_DDR 248/248
RAM 841/7846MB (lfb 1596x4MB) CPU [2%@345,0%@345,0%@345,2%@345,5%@345,2%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150
 MTS fg 0% bg 0% BCPU@38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.1C VDD_IN 1494/1551
VDD_CPU 229/229 VDD_GPU 153/153 VDD_SOC 306/344 VDD_WIFI 0/0 VDD_DDR 210/229
RAM 841/7846MB (lfb 1596x4MB) CPU [4%@345,0%@345,0%@345,4%@348,3%@342,1%@344] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150
 MTS fg 0% bg 0% BCPU@38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.4C VDD_IN 1456/1519
VDD_CPU 229/229 VDD_GPU 153/153 VDD_SOC 306/331 VDD_WIFI 0/0 VDD_DDR 191/216
RAM 841/7846MB (lfb 1596x4MB) CPU [2%@347,0%@344,0%@345,2%@345,3%@345,6%@345] EMC_FREQ 14%@102 GR3D_FREQ 0%@140 APE 150
 MTS fg 0% bg 0% BCPU@38C MCPU@38C GPU@36.5C PLL@38C Tboard@33C Tdiode@33.75C PMIC@100C thermal@37.4C VDD_IN 1456/1503
VDD_CPU 229/229 VDD_GPU 153/153 VDD_SOC 306/325 VDD_WIFI 0/0 VDD_DDR 191/210
```

Figura 4.33 – Utilización de procesadores en tarjeta Jetson TX2, todos los nucleas activos.

Al observar los términos dentro de **CPU []** podemos confirmar que efectivamente todos los núcleos se encuentran encendidos. Una vez que se selecciona un modo de operación, este prevalecerá a pesar de apagar la tarjeta, de modo que esta configuración debe ser realizada solo una ocasión. Si bien todos los núcleos del CPU se encuentran encendidos, aún se encuentran trabajando a una frecuencia de 345 MHZ y el GPU a una frecuencia de 140 MHz. Esta frecuencia se puede incrementar para obtener un mejor desempeño, para lograr esto se utiliza el comando **sudo ./jetson_clocks.sh**, que incrementa los relojes de cada procesador a su máxima capacidad. Nuevamente se verifican los cambios utilizando el comando **sudo ./tegrastats.** Esto se aprecia en la figura 4.24.

```
nvidia@tegra-ubuntu: ~                                                    5:04 PM
nvidia@tegra-ubuntu:~$ sudo ./jetson_clocks.sh
nvidia@tegra-ubuntu:~$ sudo ./tegrastats
RAM 843/7846MB (lfb 1596x4MB) CPU [0%@2035,0%@2034,0%@2034,0%@2034,0%@2034,0%@2036] EMC_FREQ 0%@1866 GR3D_FREQ 8%@1300
APE 150 MTS fg 0% bg 0% BCPU@39C MCPU@39C GPU@37.5C PLL@39C Tboard@34C Tdiode@34.5C PMIC@100C thermal@38.1C VDD_IN 3409
/3409 VDD_CPU 382/382 VDD_GPU 229/229 VDD_SOC 842/842 VDD_WIFI 0/0 VDD_DDR 1203/1203
RAM 843/7846MB (lfb 1596x4MB) CPU [0%@2034,0%@2035,0%@2034,0%@2034,1%@2034,0%@2034] EMC_FREQ 0%@1866 GR3D_FREQ 0%@1300
APE 150 MTS fg 0% bg 0% BCPU@39C MCPU@39C GPU@37.5C PLL@39C Tboard@34C Tdiode@34.5C PMIC@100C thermal@38.1C VDD_IN 3332
/3370 VDD_CPU 382/382 VDD_GPU 153/191 VDD_SOC 842/842 VDD_WIFI 0/0 VDD_DDR 1203/1203
RAM 843/7846MB (lfb 1596x4MB) CPU [0%@2035,0%@2035,0%@2035,0%@2034,0%@2034,0%@2034] EMC_FREQ 0%@1866 GR3D_FREQ 0%@1300
APE 150 MTS fg 0% bg 0% BCPU@38.5C MCPU@38.5C GPU@37.5C PLL@38.5C Tboard@34C Tdiode@34.5C PMIC@100C thermal@38.1C VDD_I
N 3332/3357 VDD_CPU 382/382 VDD_GPU 229/203 VDD_SOC 842/842 VDD_WIFI 0/0 VDD_DDR 1203/1203
RAM 843/7846MB (lfb 1596x4MB) CPU [1%@2035,0%@2035,0%@2035,0%@2036,0%@2034,0%@2034] EMC_FREQ 0%@1866 GR3D_FREQ 8%@1300
APE 150 MTS fg 0% bg 0% BCPU@39C MCPU@39C GPU@37.5C PLL@39C Tboard@34C Tdiode@34.5C PMIC@100C thermal@38.4C VDD_IN 3332
```

Figura 4.34 – Utilización de procesadores en tarjeta Jetson TX2, todos los nucleas activos, máxima frecuencia.

En esta ocasión todos los núcleos están encendidos, y cada núcleo del CPU está operando a 2 GHz, mientras que el GPU está operando a 1.3 GHz, ambos a su máxima frecuencia. A diferencia del modo de operación, la frecuencia de operación es una configuración que no se

conserva después de apagar la tarjeta, de modo que cada que se desee acelerar el procesamiento, se deberá ejecutar el comando **sudo ./jetson_clocks.sh** o generar una rutina que se ejecute con el arranque del sistema y se encargue de ejecutar automáticamente dicho comando.

4.4.4 Configuración de Hardware

Esta sección cubre las conexiones físicas necesarias para integra la tarjeta Jetson TX2 en el sistema de visión, algunas conexiones que se mencionan en la Sección 4.4.1 no se incluyen aquí debido a que son conexiones temporales utilizadas únicamente durante la instalación del SDK. La figura 4.35 muestra las conexiones utilizadas en este proyecto.

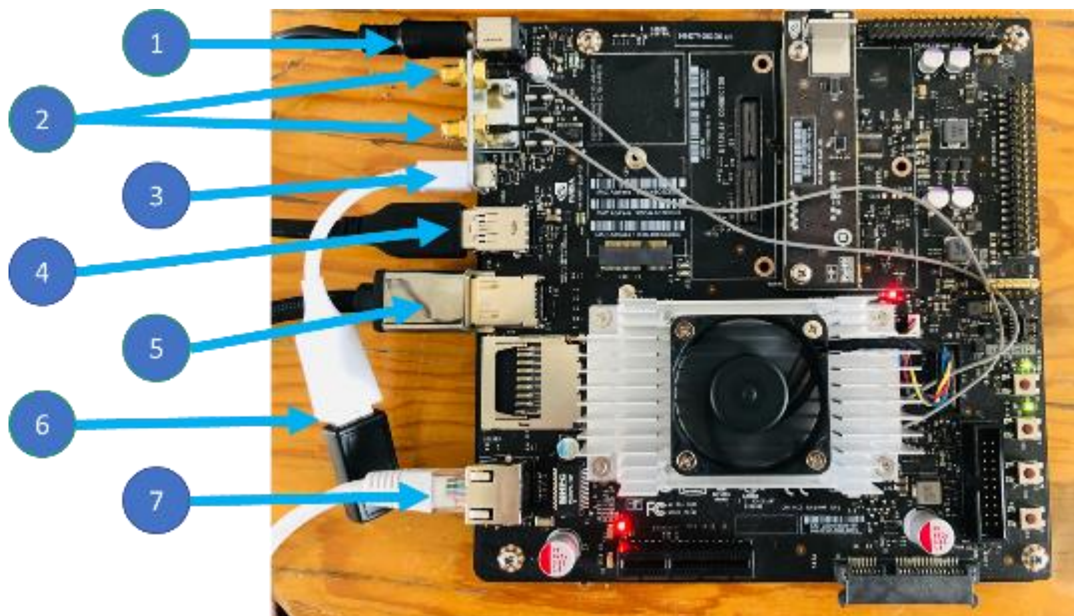

Figura 4.35 – Conexiones en la tarjeta Jetson TX2.

Los elementos que pueden observarse en la figura anterior corresponden a:

1. **Conector tipo Jack.-** Sirve para energizar la tablilla, está conectado a una fuente de alimentación de 19 V DC.
2. **Conectores axiales**.- Estos conectores se utilizan cuando se desea usar la red inalámbrica. En cada uno se conecta una antena. El uso de las antenas es opcional, en el caso de esta tesis las antenas no son requeridas porque la conexión a una red no es esencial para el funcionamiento del sistema y porque las configuraciones que

requerían la descarga de archivos de internet se realizaron utilizando la red cableada.

3. **Puerto USB 2.0 con un conector micro USB**.- Este puerto se utilizó para conectar un adaptador micro USB a USB A. En la figura 4.35 el adaptador está señalado con el número 6, que a su vez fue utilizado para conectar el receptor de un kit de teclado y ratón inalámbricos.
4. **Puerto USB 3.0 con un conector tipo A**.- Este puerto fue utilizado para conectar directamente la cámara Basler puA2500-14cm.
5. **Puerto HDMI hembra**.- A través de este puerto se conectó el monitor utilizando un cable HDMI.
6. **Adaptador micro USB a USB tipo A**.- Como se mencionó en el punto 3, este adaptador se utilizó para conectar el receptor de un kit de teclado y ratón inalámbricos.
7. **Puerto Ethernet**.- Este puerto se utilizó para conectarse a un enrutador y tener acceso a internet. Esta conexión no es necesaria para el funcionamiento del sistema, pero si fue necesaria durante el desarrollo para la descarga de librerías y archivos de internet.

En la figura 4.36 se puede observar un diagrama a bloques que muestra los diferentes elementos que interactúan con la tarjeta Jetson.

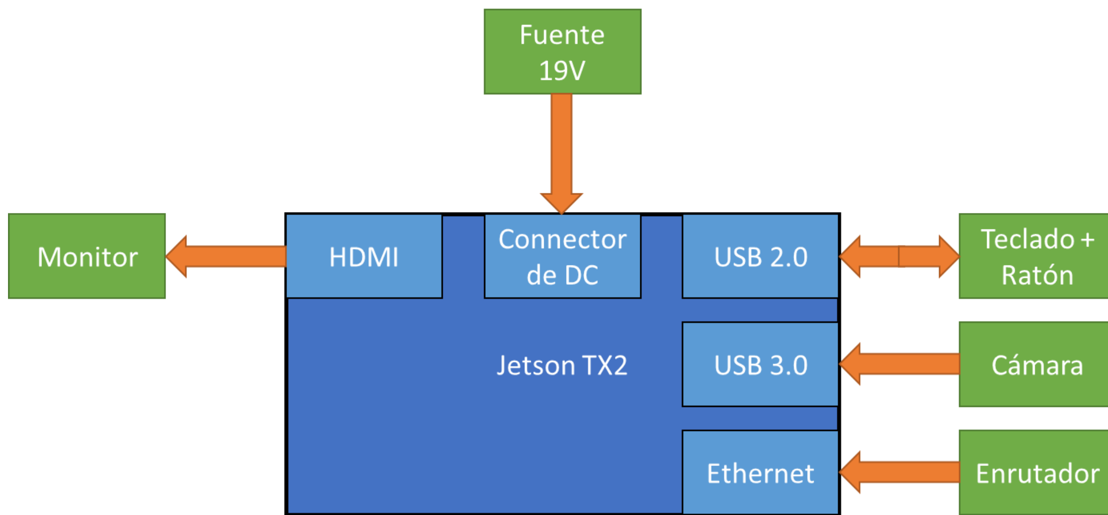

Figura 4.36 – Diagrama a bloques de las conexiones en la tarjeta Jetson TX2.

V. SOFTWARE DE DETECCIÓN DE DEFECTOS

En capítulos anteriores se mencionó que el código desarrollado para este proyecto, se realizó en Python, utilizando la librería OpenCV. Tal como se resume en la tabla 4.3, el código está estructurado de manera tal que puede ser ejecutado en cualquiera de los sistemas operativos en que se desarrolló (Windows 10, Ubuntu 16.04 o Ubuntu 18.04). La única condición para que el programa funcione de manera correcta es que se tenga instalada alguna de las versiones compatibles de Python (3.5.2, 3.6.5, o 3.6.6), junto con alguna de las versiones compatibles de OpenCV (3.4.4, 4.0.0 o 4.0.1). Cumpliendo con estos requisitos el programa puede ser ejecutado en cualquier plataforma de hardware, aunque el tiempo de ejecución puede cambiar drásticamente de una plataforma a otra.

También es importante considerar la necesidad de tener la cámara conectada a la plataforma que se esté utilizando, para la ejecución del programa completo. Esto no es del todo cierto, pues por razones de desarrollo se habilitó un modo de operación que permite ejecutar todo el código utilizando imágenes guardadas en el disco duro. Por lo que la cámara no es un requisito indispensable para el funcionamiento, los detalles sobre la habilitación y funcionamiento de este modo se detallan en la Sección 5.2.

5.1 Estructura de Archivos

El código del proyecto se dividió en 5 archivos, para una mayor claridad. Estos archivos se encuentran en la sección de anexos. Cada archivo se encarga de ejecutar una sección específica del algoritmo de detección. A continuación se expone el propósito de cada uno.

- **main.py**. Este es el archivo principal, desde donde se ejecuta la función principal **main**( ). El listado se encuentra en el Anexo 1. Para iniciar el programa se debe ejecutar este archivo desde Python. En este archivo también se maneja la entrada de información, existen dos entradas de información distintas, una de ellas proviene del usuario, ya que puede interactuar con el programa presionando diferentes teclas, la segunda entrada de información se refiere a la fuente de imágenes, ya sea desde la cámara o desde archivos almacenados en el disco duro. Desde este programa se

ejecutan las funciones de detección. La lógica completa del algoritmo se aborda a detalle en la Sección 5.3. Como todos los demás archivos se incluyen aquí como bibliotecas, todas las funciones de todos los archivos son accesibles desde aquí. Otra parte fundamental de este archivo, es que se encarga de entregar la salida de información al usuario. Esto se realiza a través de dos medios, el primero es la consola de comandos, que mediante texto se puede apreciar el estado de ejecución del programa y el segundo medio es a través de una interfaz gráfica. En dicha interfaz se despliega la imagen de salida. Existen diferentes opciones para la imagen de salida, que se abordarán en la Sección 5.2.

- **common.py**. Este archivo funciona como un recurso común entre todos los archivos. El listado se encuentra en el Anexo 5. En él se declaran todas las variables y funciones globales, es decir, aquellas que necesitan ser utilizadas en todos los archivos. Este archivo además sirve para realizar configuraciones previas al inicio del programa, algunas de estas configuraciones son: la ruta desde donde se leen las imágenes en el disco duro, el contenido de la imagen de salida, así como la configuración de las teclas para interactuar con la interfaz.
- **segmentation.py**. Este archivo contiene las funciones necesarias para realizar la primera etapa del algoritmo que es la segmentación del electrodo. El listado se encuentra en el Anexo 2. Desde el archivo **main.py** se hace un llamado a la función **Segmentation**, ella se encarga de separar el electrodo del fondo, generando así una región de interés y permitiendo que el resto del procesamiento trabaje solamente sobre el área del electrodo y no sobre el área completa de la imagen.
- **outterDefects.py**. En este archivo se encuentran las funciones necesarias para identificar los defectos que se ubican en los bordes de los electrodos, defectos de de-laminación y aluminio expuesto. El listado se encuentra en el Anexo 4. Al igual que el caso anterior, desde el archivo **main.py** se realizan las llamadas necesarias a esta biblioteca.

- **innerDefects.py**. Este archivo contiene las funciones encargadas de localizar los defectos que se producen en la superficie interna del electrodo, defectos de contaminación, burbuja, ojo de pesado y rasgado. De igual manera, las llamadas necesarias para esta biblioteca, son llevadas a cabo por el archivo principal **main.py**. El listado se encuentra en el Anexo 3.

Estos cinco archivos pueden guardarse en cualquier ruta dentro de la tarjeta Jetson, la única condición que presentan es que deben localizarse todos dentro de la misma carpeta. En el presente proyecto, los archivos se ubicaron en la ruta **~/Documents/tesis/segmentación/.**

5.2 Configuraciones para la Ejecución del Software

Para ejecutar el programa desde la tarjeta Jetson, es necesario acceder al ambiente virtual en el que se instaló la librería de OpenCV. Una vez dentro del ambiente virtual se debe navegar hasta la ruta donde se encuentran guardados los 5 archivos del proyecto y finalmente usando Python, se debe ejecutar el archivo **main.py**. Esto se puede lograr utilizando los siguientes comandos:

workon cv

cd /Documents/tesis/segmentation

python main.py

Una vez que se ejecuta el programa, existen algunas interacciones que se pueden realizar por medio del teclado, por defecto las teclas que pueden usarse y sus funciones son las siguientes:

- **“esc”**. Permite terminar la ejecución del programa, mientras no se presione esta tecla, el programa permanecerá ejecutando un ciclo infinito.
- **“n”**. Cuando se trabaja con archivos cargados desde el disco duro, esta tecla provoca que se lea y procese el siguiente archivo. Por otra parte, si se trabaja con la cámara esta tecla provoca que una nueva imagen sea adquirida.

- **“r”**. Utilizando esta tecla se fuerza a que el algoritmo vuelva a procesar la última imagen que se adquirió a través de la cámara o la última imagen que se leyó del disco duro.
- **“d”**. Al presionar esta tecla se alternan los modos de **normal** y **desarrollo**, cuando trabaja en modo **desarrollo** es posible observar en la pantalla muchas imágenes que se generaron, ya que eran útiles durante el desarrollo del programa, pero realmente no forma parte del algoritmo de detección. Por otro lado, en el modo **normal** permitirá visualizar solo algunas imágenes correspondientes a etapas del algoritmo de procesamiento. Cabe destacar que la ejecución del modo **desarrollo** toma mayor tiempo que la del modo **normal,** ya que se ejecutan operaciones adicionales.
- **“z”**. Esta tecla selecciona la etapa de **Segmentación** para mostrar en la interfaz, en la Sección 5.3 se presentan los detalles de las etapas de procesamiento. Solo una etapa de procesamiento puede mostrarse en pantalla a la vez, por lo que debe usarse esta tecla si se desea mostrar esta etapa.
- **“x”**. Esta tecla selecciona la etapa de **Defectos en bordes** para mostrar en la interfaz, en la Sección 5.3 se presentan los detalles de las etapas de procesamiento. Solo una etapa de procesamiento puede mostrase en pantalla a la vez, por lo que debe usarse esta tecla si se desea mostrar esta etapa.
- **“c”**. Esta tecla selecciona la etapa de **Defectos en superficie** para mostrar en la interfaz, en la Sección 5.3 se presentan los detalles de las etapas de procesamiento. Solo una etapa de procesamiento puede mostrarse en pantalla a la vez, por lo que debe usarse esta tecla si se desea mostrar esta etapa.
- **“v”**. Esta tecla selecciona la etapa final del programa, es decir, muestra los resultados de todo el procesamiento realizado marcando los defectos encontrados en la imagen de entrada.
- **“0-9”**. Cada tecla numérica, permite visualizar algún paso de la etapa de procesamiento que esta seleccionada en determinado momento, por ejemplo, la etapa

de segmentación consiste de 7 distintos pasos, solo uno de ellos se puede mostrar a la vez en pantalla, por lo que se debe usar las teclas numéricas si se desea cambiar el paso que se está desplegando.

Si bien, todas estas teclas han sido establecidas por defecto, pueden ser modificadas, para que una tecla distinta realice alguna de las funciones descritas anteriormente. Esto se puede realizar antes de ejecutar el programa, accediendo al archivo **common.py**, mostrado en el Anexo 5. Dentro del archivo se encuentra una sección identificada como:

```
####################################################################################################
## Inicia sección configurable por usuario
####################################################################################################
```

Dentro de esta sección será posible configurar varios parámetros:

- **Modo de desarrollo**. Cambiando la definición de **DEBUG = True** se puede lograr que el programa al ser ejecutado inicie en modo **desarrollo** o en modo **normal** según se desee. Debe considerarse que este parámetro solamente establece el modo en el que el programa inicia, pero es posible cambiar este modo mientras el programa se encuentra ejecutándose. Al cerrar y abrir el programa, siempre iniciará de acuerdo a la definición establecida aquí.

- **Uso de la cámara**. El programa tiene capacidad de funcionar con la cámara adquiriendo imágenes a través de ella, o sin la misma, cargando imágenes previamente guardadas en el disco duro. Para decidir si la cámara será utilizada o no, se debe modificar la definición **USE_CAMERA = True**. Cuando este parámetro es igual a **true**, la cámara debe estar conectada al sistema, cuando este parámetro es **False**, la cámara no será utilizada y en su lugar se buscarán imágenes en el disco duro.

- **Ubicación de imágenes en disco duro**. Este parámetro es relevante solamente cuando se ejecuta el programa, configurado para no utilizar la cámara. Si este es el caso, las imágenes serán cargadas desde el disco duro. Al modificar las definiciones

PATH='./imagenes/' y **EXTENSION='*.jpg'**, podemos indicar al programa en que ruta y con qué extensión debe buscar los archivos. Considere que la ruta puede ser absoluta a la carpeta que contiene las imágenes o relativa partiendo de la ruta en la que se encuentra el archivo **main.py**.

- **Teclas de selección**. Cada una de las teclas establecidas por defecto puede ser cambiada por otra. Es importante mencionar que debe evitarse seleccionar una misma tecla para varias funciones y que esto podría ocasionar problemas en la ejecución del programa.

- **Etapa y paso**. Se pueden modificar las definiciones **stageToDisplay = STAGE0** y **processToDisplay = FINAL_IMAGE** para definir que etapa y que paso se desea desplegar en la interfaz cuando el programa inicie. Estos dos parámetros son configurables durante la ejecución usando las teclas descritas anteriormente. Al igual que el modo de operación, a pesar de que este parámetro se cambie durante la ejecución del software, cada vez que se reinicie la ejecución se tomarán las definiciones del archivo **common.py**

El final de la sección configurable está identificado de la siguiente forma:

```
################################################################################
## Termina sección configurable
################################################################################
```

Debe tenerse en cuenta, que ningún parámetro que se encuentre fuera de la sección definida como configurable, debe ser modificado.

5.3 Flujo del Programa

Como se mencionó anteriormente, el código está compuesto por 5 archivos, de los cuales el principal es **main.py**. Este se encarga de controlar el flujo del programa, el resto de los archivos funcionan como bibliotecas a las cuales se llama para ejecutar tareas específicas del

procesamiento. La figura 5.1, muestra un diagrama de flujo donde se describe el funcionamiento del programa de detección de defectos.

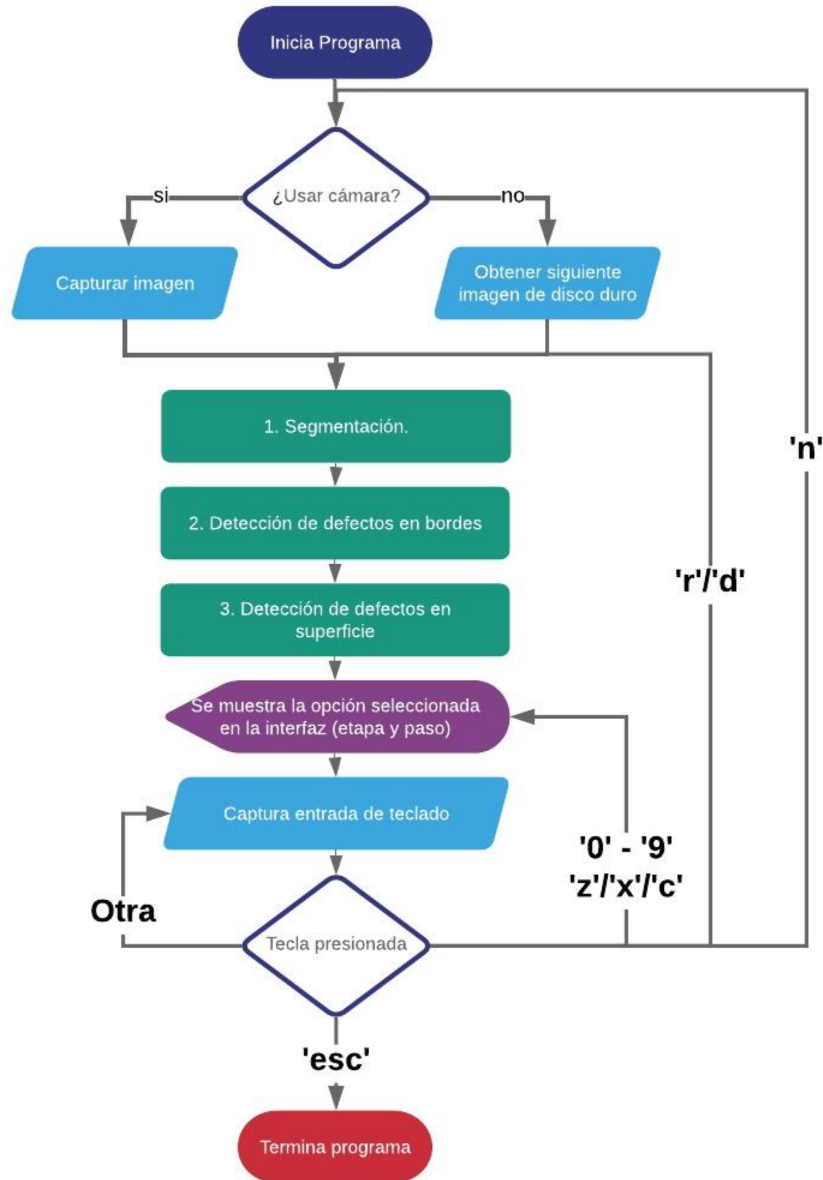

Figura 5.1 – Diagrama de flujo de la función **main( )**.

Lo primero que el diagrama de flujo indica, es que existen dos posibilidades para obtener la imagen a procesar, una es por medio de la cámara, y la otra por medio de imágenes previamente guardadas, estas dos opciones son mutuamente excluyentes. Una vez que se tiene la imagen a

procesar el flujo es realmente simple y la principal tarea de la función **main( )** es manejar la interacción con el usuario. Es también evidente que el procesamiento de la imagen de entrada ocurre en tres etapas: segmentación, detección de defectos en bordes y detección de defectos en superficie. Estas etapas son secuenciales. Cuando se ejecuta el programa aparece una de las ventanas que se muestran en la figura 5.2, dependiendo del modo de ejecución.

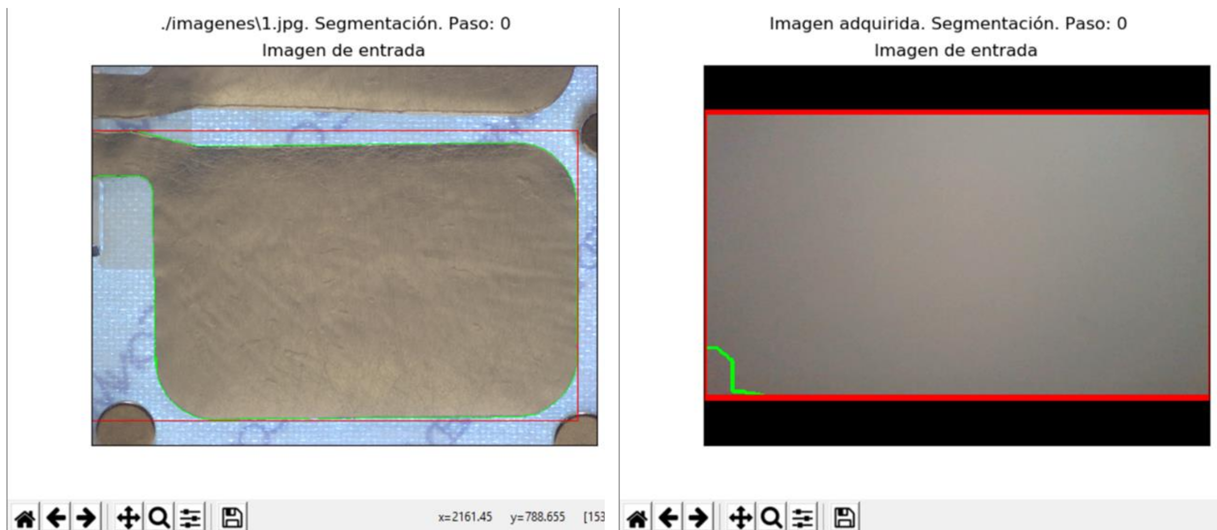

Figura 5.2 – Programa en ejecución. Izquierda: sin utilizar cámara. Derecha: utilizando cámara.

En la esquina inferior izquierda, se pueden observar una serie de botones que permiten manipular la imagen que se despliega en pantalla. Estas herramientas permiten mover, realizar zoom, seleccionar o guardar la imagen. Arriba de la imagen puede verse el título, este se compone de cuatro campos. El primer campo es el nombre y ruta desde donde se abrió la imagen, esto solo en caso de proceder del disco duro (figura 5.2 izquierda). En caso de que se esté utilizando la cámara, en lugar de nombre se observará el texto "Imagen adquirida" (figura 5.2 derecha). El segundo y tercer campo indican la etapa de procesamiento y el paso seleccionado para desplegar en pantalla. Finalmente, el último campo muestra el nombre de imagen desplegada.

En las siguientes secciones se describe detalladamente cada etapa de procesamiento, así como los pasos que la componen. Cabe señalar que la función **main( )** no es buena referencia para estos detalles, ya que cada etapa es ejecutada en una biblioteca distinta y desde la función

principal solamente se hace una llamada a las funciones. En su lugar deben consultarse las bibliotecas **segmentation.py, outterDefects.py** e **innerDefects.py**.

5.3.1 Segmentación

La primera etapa de procesamiento es la segmentación. Esta se lleva a cabo utilizando la función **Segmentation( )**, que es parte de la librería **segmentation.py**. Esta función tiene como argumento de entrada la imagen a ser procesada en formato Verde-Rojo-Azul (GRB por sus siglas en inglés, *Green-Red-Blue*), ya que este formato es el que utiliza OpenCV por defecto para cualquier imagen. Los argumentos de salida son:

- La ROI, es decir, el área mínima que contiene al electrodo en formato RGB.
- El contorno del electrodo.

La finalidad de esta etapa es aislar el electrodo removiendo el fondo y reduciendo el área de la imagen, de tal forma que solo se conservan aquellos pixeles que contienen información del objeto de interés. El objetivo es que las siguientes etapas puedan trabajar solamente con los datos que son relevantes. En la figura 5.3 se puede observar un diagrama flujo que refleja la lógica de esta etapa de procesamiento. Como se ha mencionado previamente, se tienen dos modos de ejecución: **normal** y **desarrollo**. En esta etapa, al igual que en el resto, ambos modos entregaran los mismos resultados, la única diferencia es que el modo de **desarrollo** genera información adicional que posteriormente podrá ser desplegada y fue utilizada principalmente durante el periodo de pruebas. Por esta razón, solo se describirá el flujo del modo de ejecución **normal**.

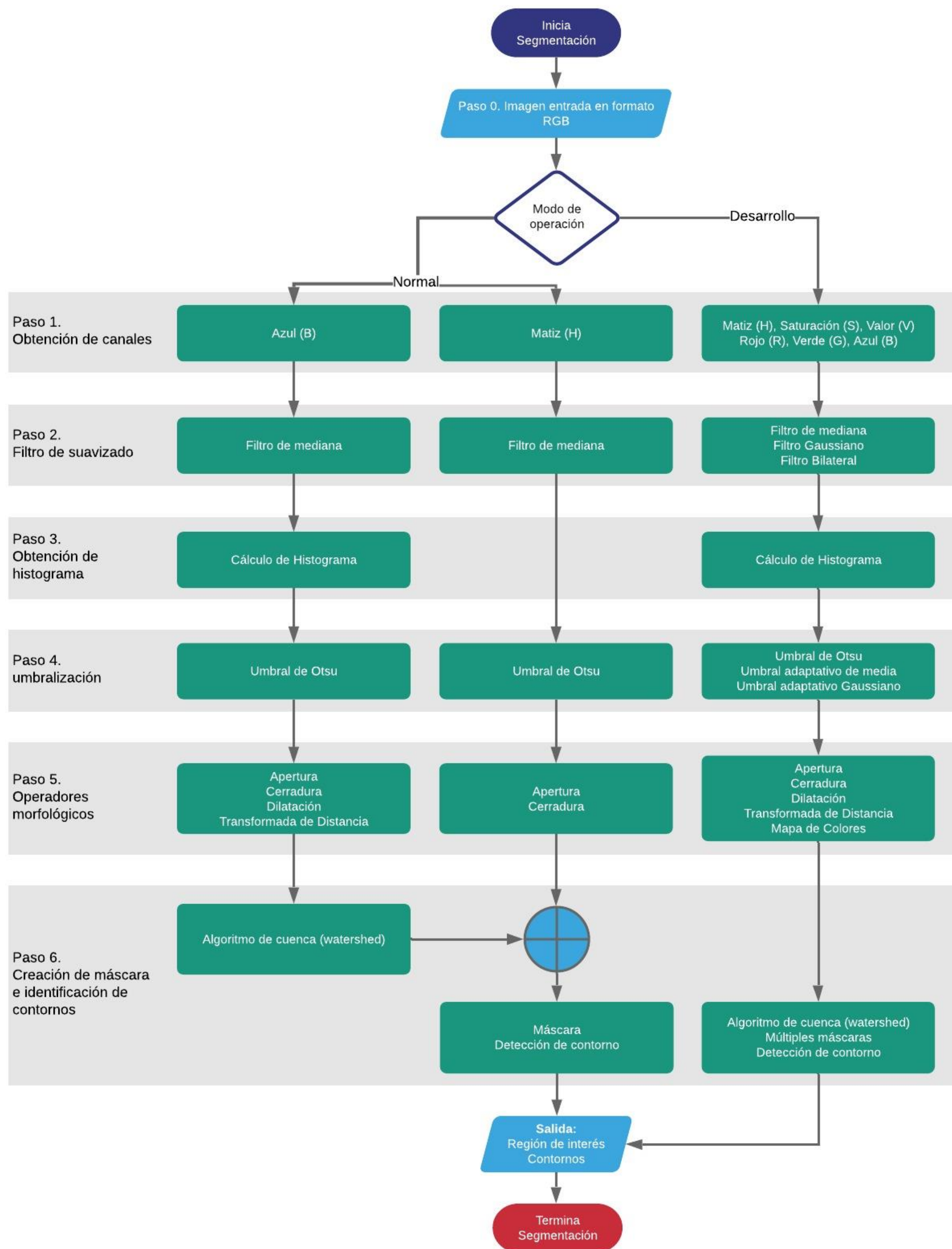

Figura 5.3 –Diagrama de flujo de la función **segmntation( )**.

El procesamiento se ejecuta de manera secuencial en seis pasos:

1. En el primer paso, la imagen de entrada en formato GRB se convierte al espacio de color Matiz-Saturación-Valor (HSV, por sus siglas en ingles *Hue-Saturation-Value*). Luego ambos espacios de color se descomponen en los canales que los conforman. Los canales Azul (B) y Matiz (H) serán utilizados en los pasos siguientes, ya que son los que muestran un mayor contraste entre el electrodo y el papel adhesivo del fondo, tal como puede observarse en la figura 5.4. Esto se debe a la combinación de iluminación de campo oscuro generada con una luz cálida y la luz de fondo con un color más frio. A partir de este paso, siguiendo la estrategia planteada en [25], se ejecutan dos procesamientos paralelos, ambos con la finalidad de aislar al electrodo del papel adhesivo. Pero uno de ellos trabaja sobre la imagen obtenida del canal B, a la que nos referiremos como **imgA**, mientras que el otro utiliza la imagen obtenida del canal H a la que nos referiremos como **imgH.**

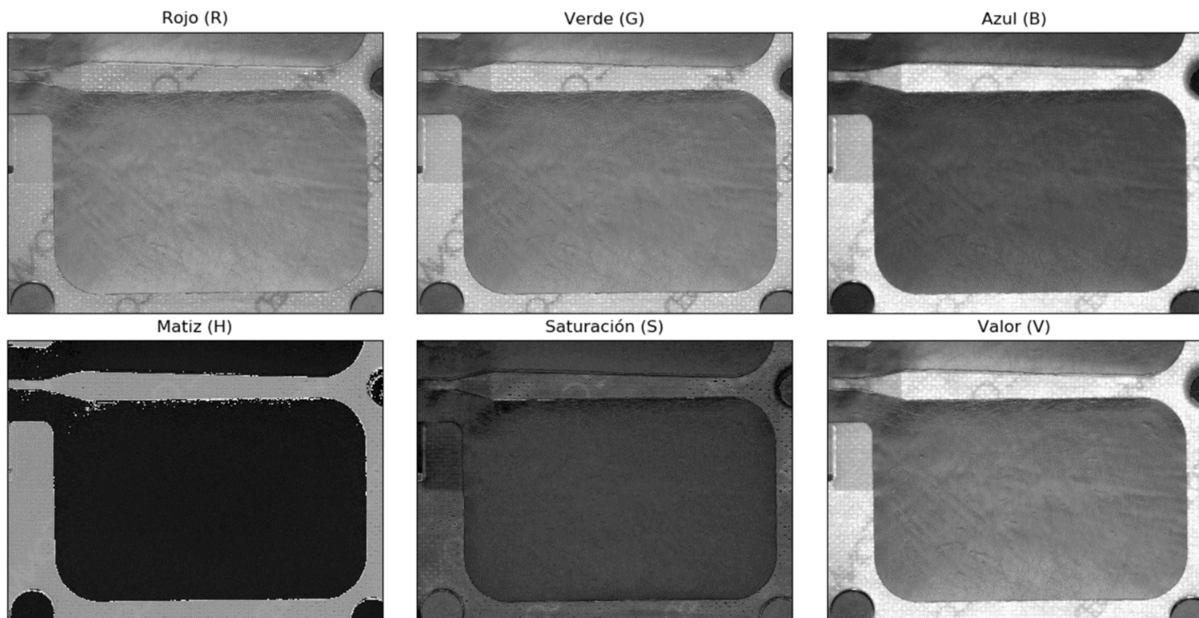

Figura 5.4 – Imagen de Electrodo, vista en cada canal de los espacios RGB y HSV.

2. El segundo paso aplica un filtro de suavizado, específicamente de mediana, sobre las imágenes obtenidas en el paso previo con el fin de remover el ruido. Se evaluaron otras opciones de filtros como el filtro Gaussiano, filtro de medias y filtro bilateral,

siendo el filtro de mediana el que mejor se desempeñó. La figura 5.5 muestra los efectos de los filtros de suavizado aplicados sobre las imágenes **imgA** e **imgH.** Nos referiremos a las imágenes filtradas como **imgAfiltrada** e **imgHfiltrada**, respectivamente.

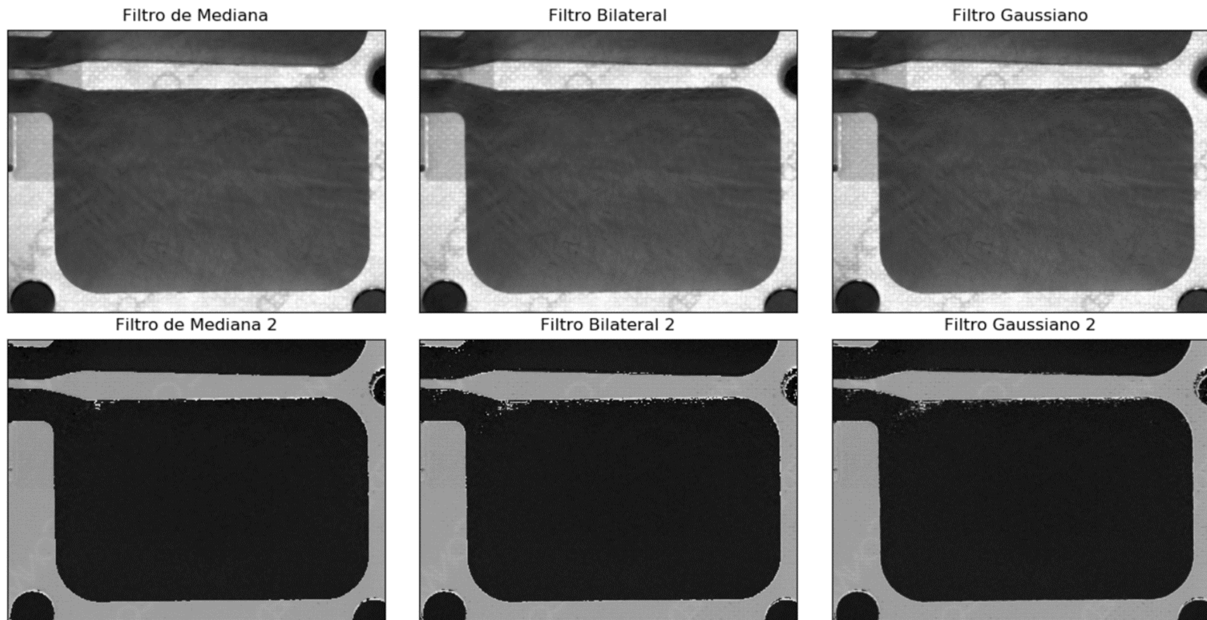

Figura 5.5 – Primera fila: Filtros de suavizado aplicados sobre **imgA**. Segunda fila: filtros de suavizado aplicados sobre **imgH**.

3. El paso tres consiste en obtener el histograma de **imgAfiltrada**. Si bien el histograma no se utiliza en pasos posteriores del procesamiento, fue esencial analizarlo para determinar que método de umbralizado sería el más adecuado. La figura 5.6 muestra el histograma de **imgAfiltrada**, se puede apreciar que existen dos picos en la distribución. Estos dos picos corresponden a dos clases claramente separadas, una corresponde al fondo o papel adhesivo y otra al objeto de interés o electrodo. En este tipo de distribuciones, tal como comprueban en [23], el método de Otsu es sumamente confiable. La línea roja que se muestra en el histograma corresponde al valor óptimo de umbral encontrado con el método de Otsu para separar ambas clases.

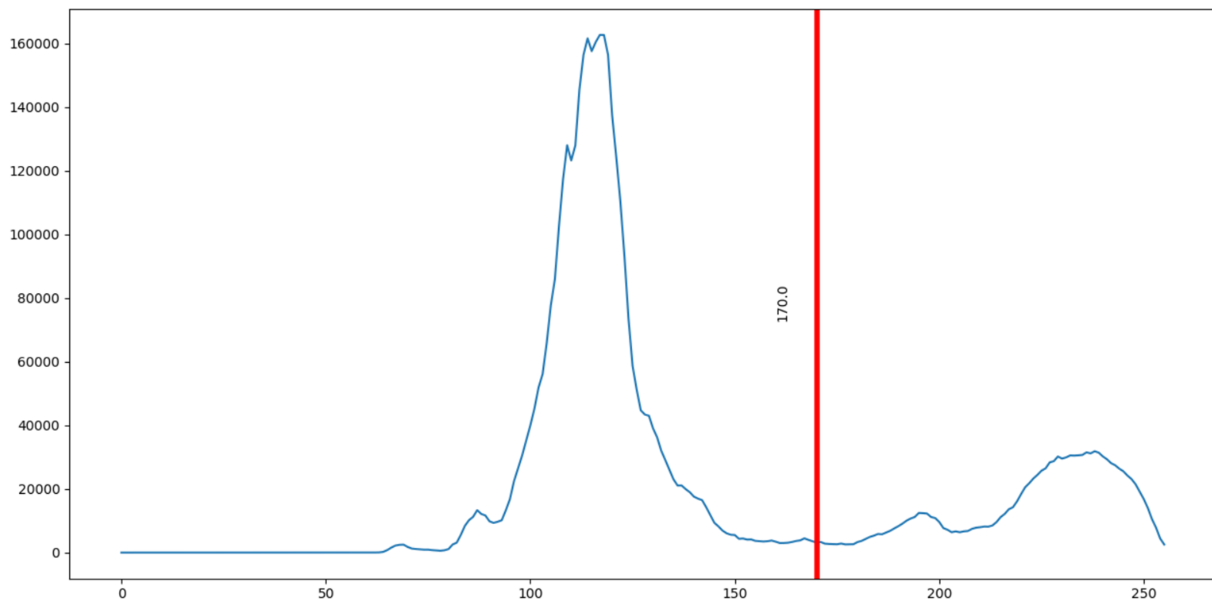

Figura 5.6 – Histograma de **imgAFiltrada**

4. El cuarto paso consiste en binarizar las imágenes **imgAfiltrada** e **imgHfiltrada** por medio del método de umbralizado de Otsu. Como se mencionó en el paso anterior, este método resulta bastante efectivo. Se evaluaron además los métodos de umbralizado adaptativo de mediana y Gaussiano. La figura 5.7 muestra el resultado de las distintas técnicas de umbralizado aplicadas en **imgAfiltrada**. En adelante nos referiremos a las imágenes binarizadas como **binA** y **binH**, respectivamente.

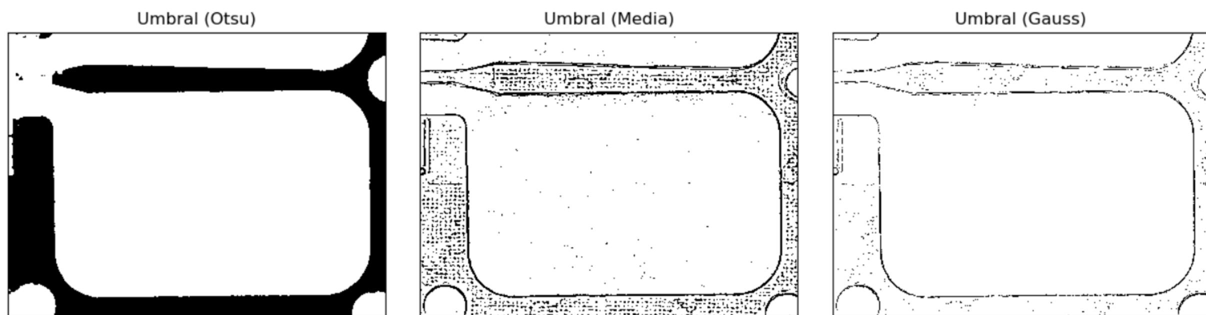

Figura 5.7 – Imagen **imgAFiltrada** binarizada utilizando distintos métodos de umbralizado.

5. En el quinto paso se aplican operaciones morfológicas a las imágenes **binA** y **binH** con el propósito de eliminar los pequeños puntos negros o blancos que se encuentran dispersos en dichas imágenes. Las operaciones que se utilizan son de apertura y

cierre. Al aplicar la apertura se eliminan todos los pequeños puntos blancos que no están contenidos en su totalidad en las áreas más grandes, mientras que la operación de cierre sirve para rellenar o eliminar los pequeños huecos o puntos negros que se localizan en dichas áreas. Las primeras dos imágenes de la figura 5.8 muestran los resultados de aplicar estos operadores a **binA.** Después de las operaciones morfológicas se puede considerar a **binH** como la primera máscara denominada **máscara 1**.

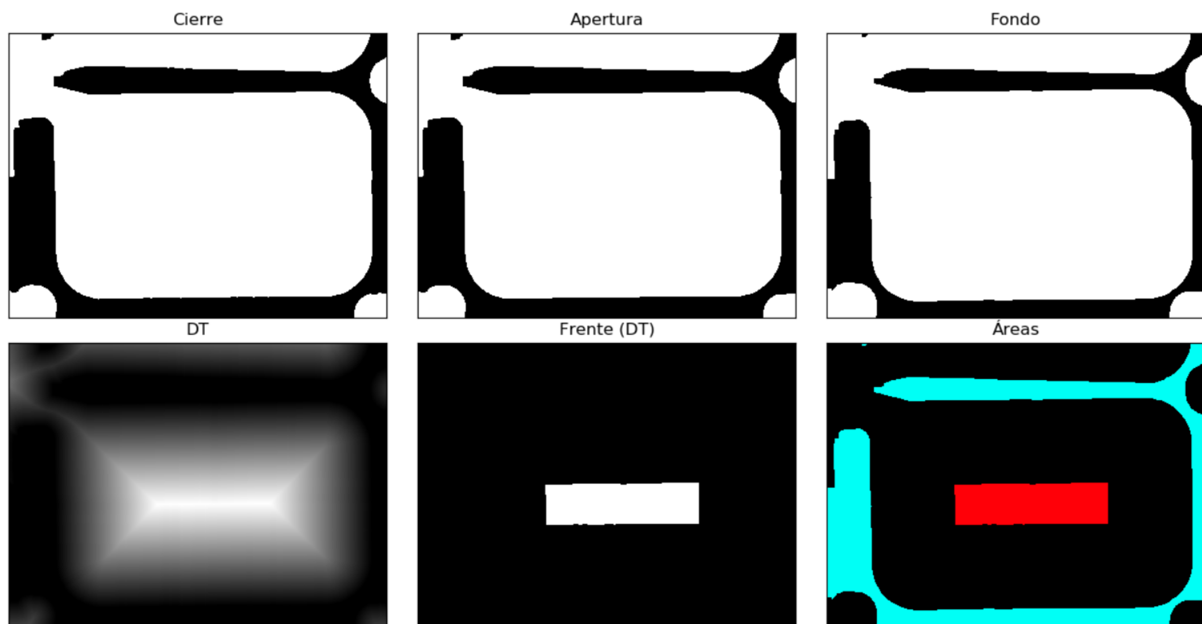

Figura 5.8 – Primera fila: Imagen **binA** después de operaciones morfológicas. Segunda fila: Transformada de distancia de **binA**, umbralizado de transformada de distancia y mapa de colores de áreas.

Si se observa el diagrama de flujo de la figura 5.3 se puede notar que en este paso, el procesamiento para **binA** y para **binH** es distinto. Las operaciones morfológicas son aplicadas en ambas imágenes, pero **binA** es además sometida a una operación de **dilatación**, utilizada para agrandar el área del electrodo. Al hacer esto se sabe que el área blanca abarca totalmente al electrodo y probablemente también una porción del fondo cercana a los bordes, pero también se puede deducir con tener total certeza que el área negra pertenece al fondo, de modo que a esta área se le denomina como el **fondo**. Posteriormente se aplica una transformación de distancia a la imagen dilatada.

La transformación de distancia es una operación que se aplica a imágenes binarias y genera una imagen en escala de grises donde cada pixel que se encuentre dentro de un objeto (área blanca) será reemplazado por un valor de intensidad que será proporcional a la distancia que existe entre el pixel y el borde más cercano, de modo que los pixeles cercanos a los bordes tendrán valores bajos y los pixeles de las zonas centrales tendrán valores altos. Esto puede apreciarse en la primera imagen de la segunda fila de la figura 5.8 que muestra la transformación de distancia de la imagen **binA**. Después de aplicar las operaciones morfológicas se procede a binarizar la transformada de distancia, esta vez con el propósito de conservar solamente los pixeles centrales del electrodo, los de valor mayor, que podemos asegurar pertenecen al electrodo. Al final de este paso se han identificado tres áreas: pixeles que pertenecen al fondo, pixeles que pertenecen al electrodo y pixeles inciertos, que pueden pertenecer al fondo o al electrodo. La última imagen de la figura 5.8 muestra un mapa de colores donde cada área es representada con un color distinto, el fondo aparece en azul, el electrodo en rojo y el área incierta en negro.

6. El último paso de esta etapa consiste en utilizar las tres áreas identificadas en el paso anterior para ejecutar el algoritmo de cuencas hidrográficas (*watershed* en inglés), para generar una segunda máscara. Este algoritmo se basa en el crecimiento de regiones pero necesita semillas o marcadores para identificar las zonas o áreas que crecerán. Por lo que, las áreas identificadas con certeza en el paso anterior, el fondo y el electrodo, son usadas como marcadores para este algoritmo. Al terminar este paso se tendrá una máscara para el área del electrodo, denominada **máscara 2**. Finalmente se suman las dos máscaras obtenidas de modo que toda área identificada como electrodo por cualquiera de las dos máscaras es considerada electrodo. A esta combinación se le llama **máscara combinada**, que es más precisa que cualquiera de las dos máscaras que la componen. La figura 5.9 muestra un ejemplo claro de como la combinación de ambas máscaras puede producir un resultado bastante certero.

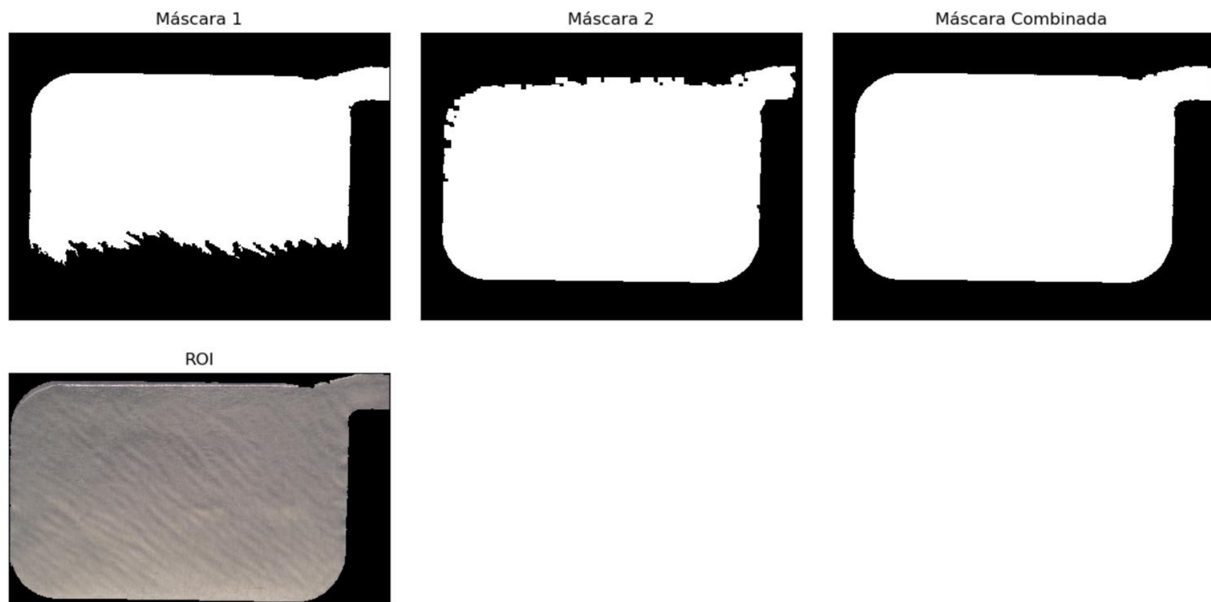

Figura 5.9 – Primera fila: máscaras utilizadas para segmentación. Segunda fila: Imagen segmentada.

5.3.2 Detección de Defectos en Bordes

La segunda etapa de procesamiento corresponde a la detección de defectos en bordes. Esta etapa se ejecuta utilizando la función **EdgeDefects( ),** que se encuentra en la biblioteca **innerDefects.py**. Esta etapa está enfocada en localizar los defectos que pueden producirse en los bordes del electrodo. Estos defectos son: aluminio expuesto y de-laminación. Si bien, algunos otros defectos que se presentan en el área interna del electrodo también pueden ser localizados en esta etapa, su objetivo principal es la localización de los dos defectos mencionados. La figura 5.10 muestra un diagrama de flujo que ilustra el procesamiento realizado por esta función. Como la función **EdgeDefects( )** se ejecuta después de la etapa de segmentación, la ROI y el contorno del electrodo ya fueron identificados previamente. Ambos datos serán argumentos de entrada para esta función.

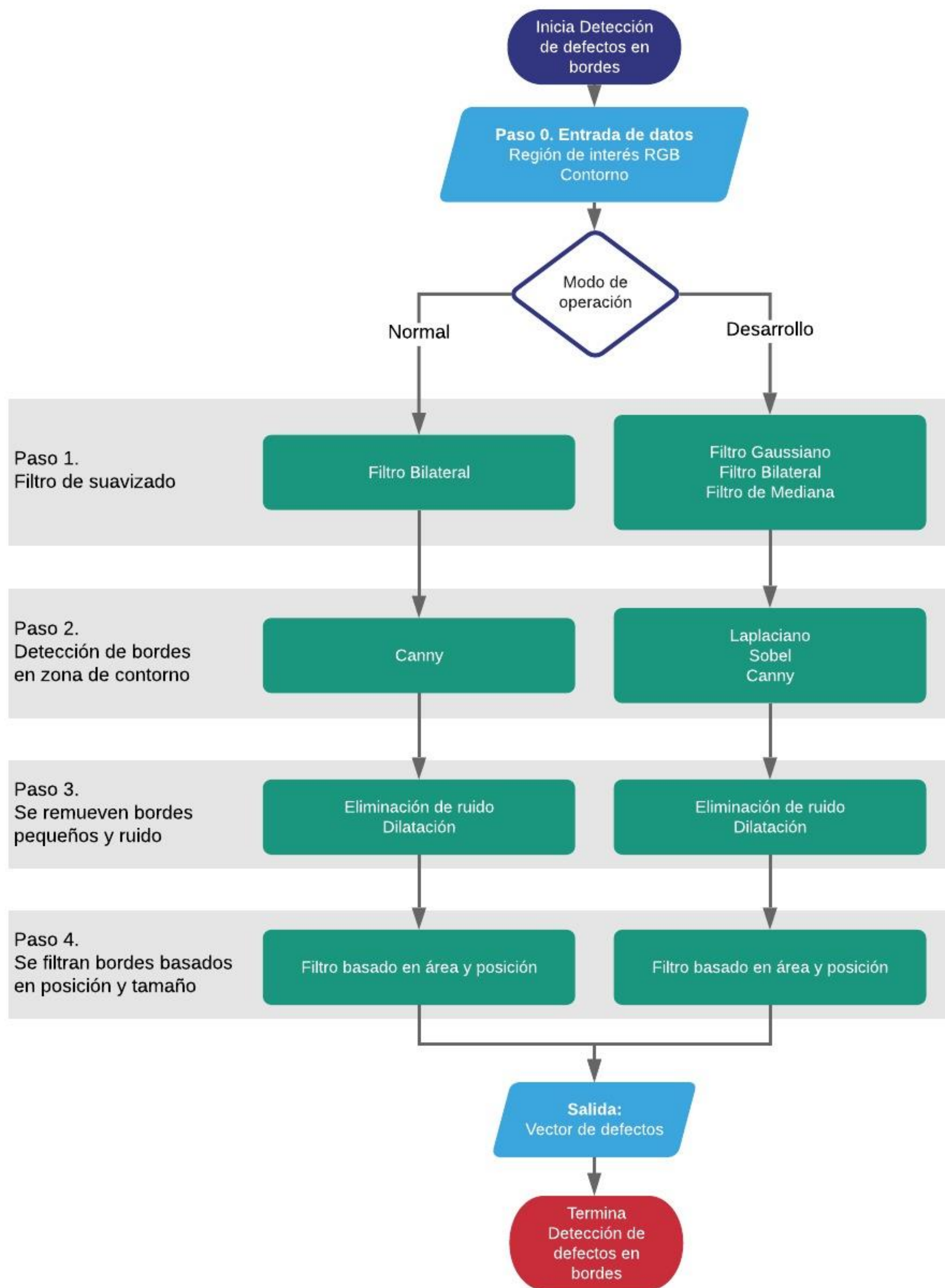

Figura 5.10 –Diagrama de flujo de la función **EdgeDefects ( )**.

Esta etapa es realizada en cuatro pasos, cada uno de estos pasos se describirá enfocado únicamente al modo de operación **normal**.

1. El primer paso sirve para remover el ruido que pueda estar presente en la imagen, para ello primero se toma la imagen de entrada en formato RGB y se convierte a escala de grises, posteriormente se utiliza un filtro de suavizado bilateral, el cual resulta bastante útil cuando se desea eliminar el ruido sin perder el detalle de los bordes. La figura 5.11 muestra el resultado de este paso aplicado en una imagen, como se puede observar en dicha imagen. También fueron evaluados otros filtros de suavizado como el filtro de mediana y el filtro gaussiano.

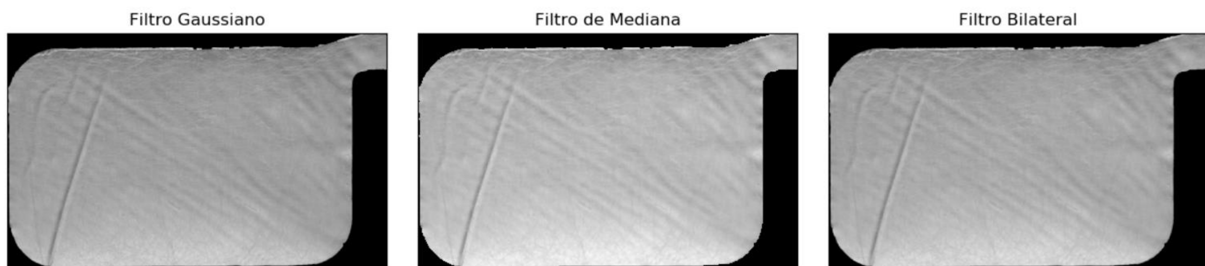

Figura 5.11 –Filtros de suavizado aplicados a la imagen de entrada en la función **EdgeDefects** ( ).

2. El siguiente paso consiste en la aplicación de un detector de bordes sobre la imagen suavizada en escala de grises. Para este paso se evaluaron los detectores de bordes del Laplaciano, Sobel y Canny. El detector de bordes de Canny probó ser el más eficiente de los tres debido a las etapas de filtrado de ruido y la detección de bordes basado en máximos locales. En la figura 5.12 se puede apreciar los bordes detectados con cada uno de estos algoritmos evaluados. El resultado de este paso es una máscara binaria que contiene los bordes en la imagen, a la que se hará referencia como **canny**.

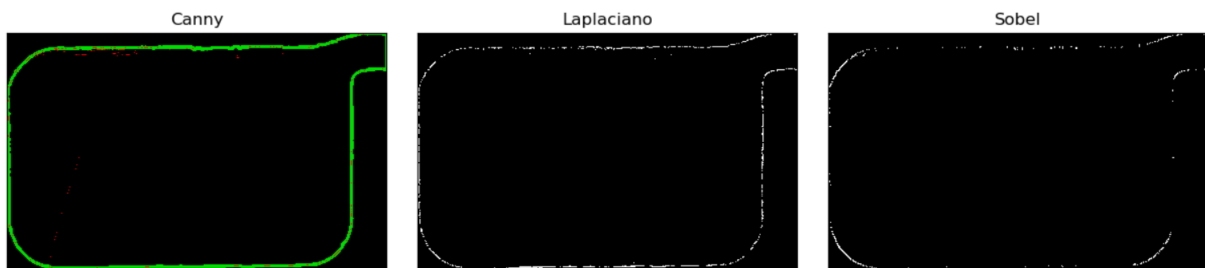

Figura 5.12 –Detectores de bordes aplicados sobre imagen suavizada.

3. En el tercer paso se toma la máscara **canny** y se procede a eliminar los bordes detectados que son causados por ruido, como la textura fibrosa del gel. La mayoría de estos bordes son de dimensiones pequeñas, de modo que se verifican todos los bordes y se eliminan todos aquellos que puedan ser contenidos por un cuadrado de 10 pixeles por 10 pixeles. Después, de los bordes restantes se extraen los que se sabe pertenecen al contorno del electrodo mismo. Para esto se hace uso de la imagen de contorno pasada como argumento de entrada a la función **EdgeDefects( )**. Una vez eliminados los bordes causados por el electrodo y los que son causados por ruido, es altamente probable que los bordes restantes sean algún tipo de defecto. De modo que se procede a resaltarlos mediante la operación morfológica de dilatación con el objetivo de manipularlos más fácilmente en los pasos posteriores. La figura 5.13 muestra un ejemplo de la aplicación de estas operaciones. A la máscara generada en este paso se le llama **dilationMask.**

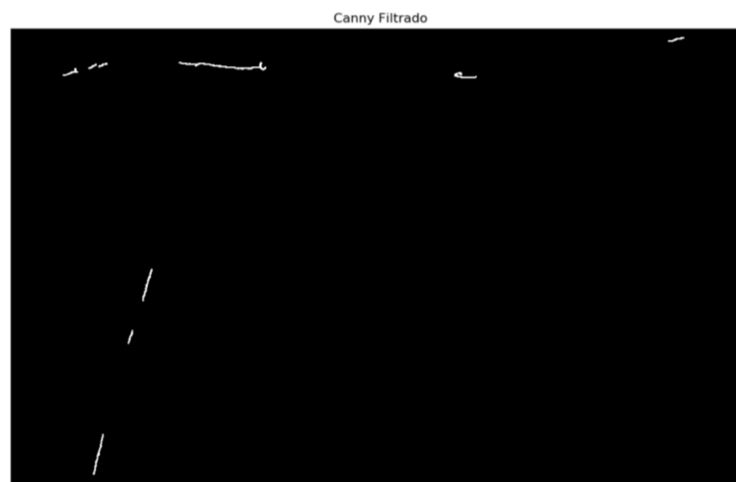

Figura 5.13 – Máscara **canny** después del proceso de filtrado de bordes y dilatación.

4. En el último paso se realiza un último filtrado de los bordes restantes para asegurar que solo aquellos que son defectos se conserven. Después de analizar los bordes detectados producidos por defectos reales y compararlos con aquellos ocasionados por otras causas se detectaron dos rasgos distintivos: la forma y el tamaño.

   Se determinó que los defectos reales tienden a ser de dimensiones grandes, mayores a 1500 pixeles. Usualmente estos bordes lucen como líneas rectas bien definidas. Por lo que, todo borde con un área mayor a 1500 pixeles es considerado un defecto. También se identificó que algunos defectos tienen áreas pequeñas, menores a 1500 pixeles. Cuando se trata de estos defectos se puede identificar el rectángulo más pequeño que pueda contener en su totalidad al borde detectado para dicho defecto, las dimensiones de los lados del rectángulo serán **A** y **B**, siendo siempre **A>B**. La relación de **A/B** describe que tanto se asemeja ese rectángulo a un cuadrado, cuanto más cercano sea este valor a 1 será más parecido al cuadrado. Por otro lado, entre mayor sea el valor de la relación **A/B** el rectángulo será menos parecido al cuadrado. Los defectos de áreas pequeñas tienden a estar contenidos en rectángulos cuya relación **A/B** es menor a 3.5, de modo que todo borde que pueda contenerse en un rectángulo cuya relación **A/B** sea menor a 3.5 será considerado un defecto.

   Basado en los rasgos anteriores se procede a filtrar la máscara **dilationMask** para conservar solo aquellos bordes que cumplan con alguno de los dos rasgos correspondientes a los defectos en el borde del electrodo. Finalmente, a partir de la máscara resultante se obtienen las coordenadas de cada borde y se procede a dibujar un rectángulo rojo en la imagen original para identificar cada defecto encontrado. La figura 5.14 muestra el resultado de la ejecución de este paso. La salida de esta etapa de procesamiento es un vector que contiene los defectos encontrados en la imagen de entrada.

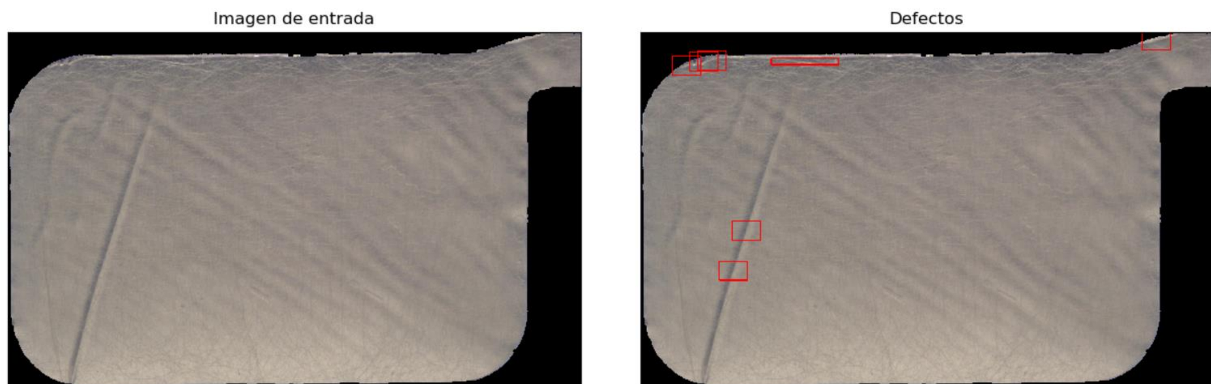

Figura 5.14 – Resultado del proceso de detección de defectos en el borde del electrodo.

5.3.3 Detección de Defectos en Superficie

Esta etapa se lleva acabo al ejecutar la función **SurfaceDefects( )**, que se encuentra en la librería **outterDefects.py**. La figura 5.15 muestra el diagrama de flujo de la esta función. Esta etapa es ejecutada después de la segmentación, por lo que recibe como argumentos de entrada la ROI de y los contornos del electrodo. El propósito de esta etapa de procesamiento es identificar los defectos que se presentan en la superficie interna del electrodo. Estos defectos son: **rasgado**, **contaminación**, **burbuja** y **ojo de pescado**.

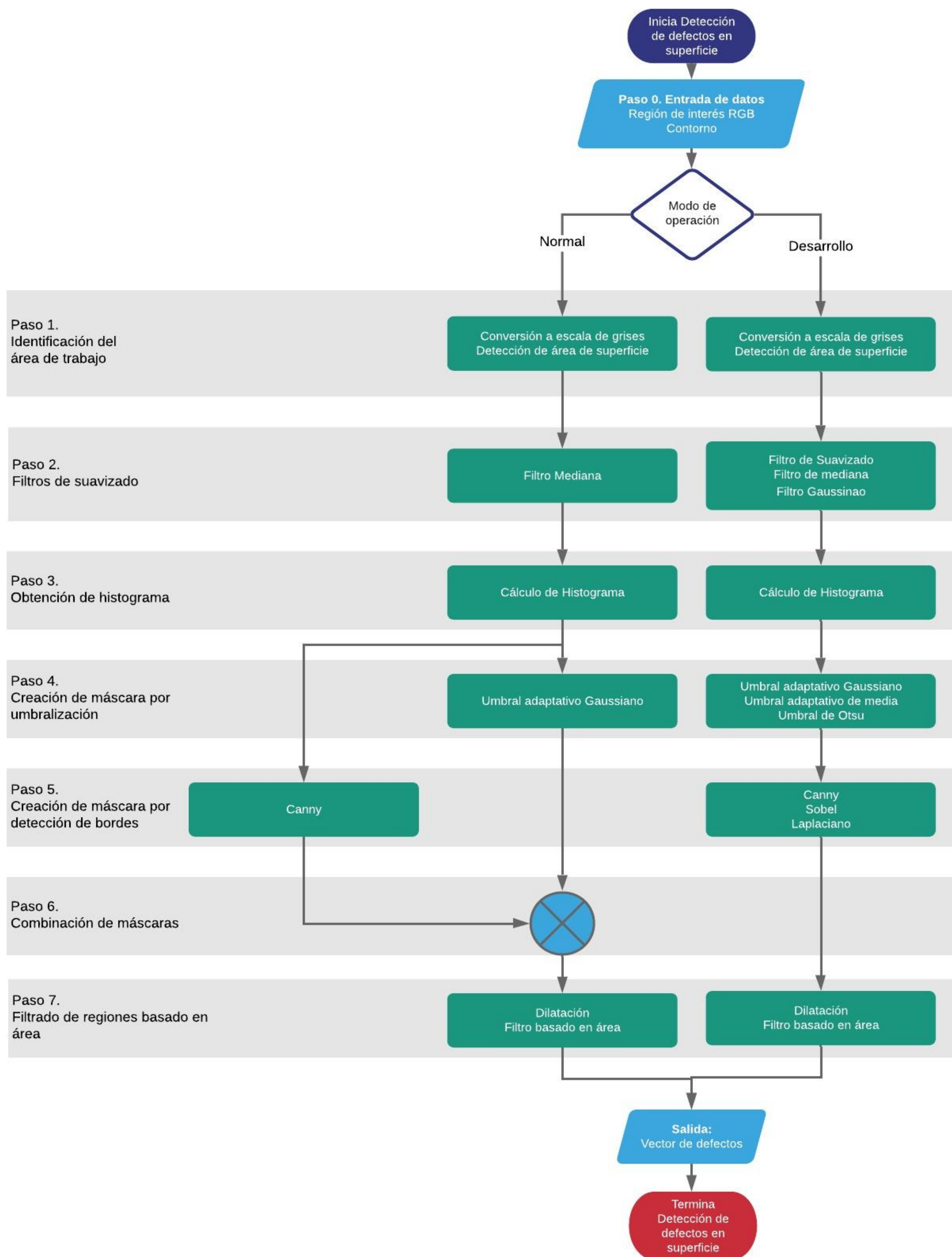

Figura 5.15 –Diagrama de flujo de la función **SurfaceDefects( )**.

A diferencia de la etapa previa que se enfocaba en los defectos del borde del electrodo, aunque detectando defectos en otras áreas, en esta etapa únicamente se detectan defectos en el área interna del electrodo. Los siete pasos necesarios para el procesamiento de esta etapa son:

1. A pesar de que uno de los argumentos de entrada de esta función es la región de interés (ROI) obtenida en la etapa de segmentación, esa región aun contiene información que no es de utilidad para esta etapa, como la información de los bordes y el fondo. El primer paso consiste en ajustar la ROI de modo que solo contenga la superficie interna del electrodo. Para ésto se convierte la imagen de entrada a escala de grises, luego se identifica el punto central de cada extremo de la imagen y se realiza un barrido desde cada punto hacia el centro buscando el primer pixel que no sea negro, es decir, el primer pixel que no pertenezca al fondo. Una vez que se encuentran los cuatro puntos se puede dibujar un rectángulo donde cada arista pase por estos puntos, pero este rectángulo aun contendría información de los bordes, por lo que se agrega un factor de compensación fijo de 100 pixeles, de modo que el rectángulo dibujado contenga únicamente la superficie interna. La ROI utilizada en esta etapa será el área contenida en ese rectángulo. Esto puede apreciarse en la figura 5.16.

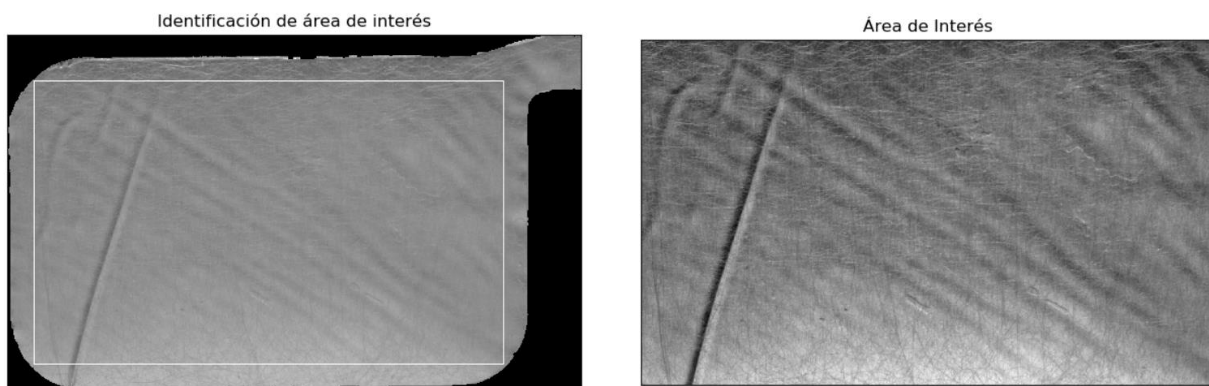

Figura 5.16 –Izquierda: Imagen de entrada con la ROI marcada. Derecha: ROI.

2. En el paso número dos se aplica un filtro de suavizado a la ROI. Al igual que en etapas anteriores, se evaluaron los filtros de mediana, bilateral y gaussiano. Para el

propósito de esta etapa, el filtro de mediana resultó ser el más apropiado. En la figura 5.17 se pueden apreciar los resultados de utilizar los distintos filtros.

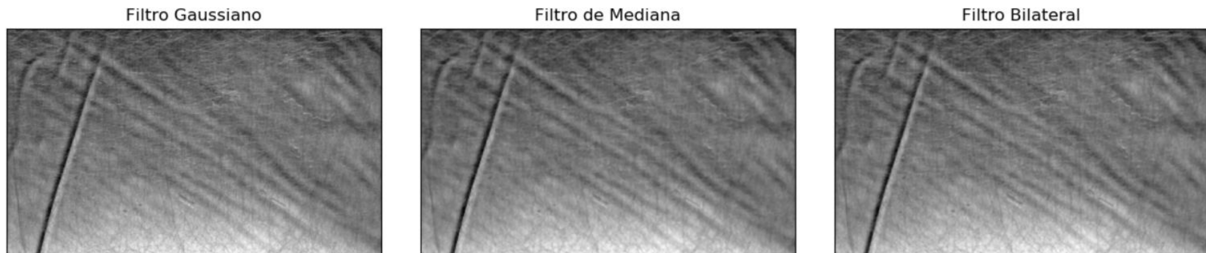

Figura 5.17 –Filtros de suavizado aplicados en la ROI.

3. El paso tres en esta etapa consiste en obtener el histograma de la imagen suavizada. El histograma no se utiliza en pasos posteriores del procesamiento, pero fue necesario analizarlo para determinar que método de umbralizado sería el más adecuado. La figura 5.18 muestra el histograma obtenido donde se puede apreciar que en este caso no existe una separación notoria de clases, por lo que el método de Otsu no es viable. Los métodos adaptativos resultan más convenientes, puesto que utilizan distintos valores de umbral para diferentes regiones en una misma imagen, encontrando el valor óptimo para un determinado vecindario. Se evaluaron dos variantes de este método, el gaussiano y el de media. Finalmente, el gaussiano permitió obtener mejores resultados, por lo que fue seleccionado para el procesamiento.

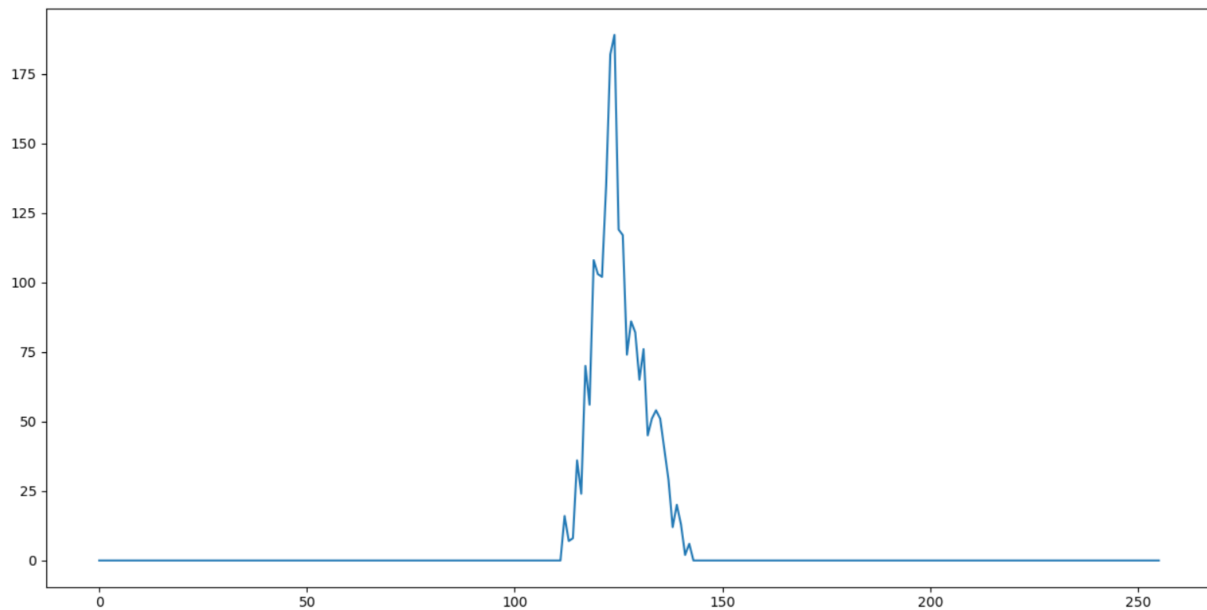

Figura 5.18 – Histograma de imagen suavizada.

4. En este paso se realiza la binarización de la imagen utilizando el método adaptativo gaussiano, creando así la primera máscara definida como **mascara 1**. Además del método adaptativo gaussiano se evaluaron otros métodos, la figura 5.19 muestra los resultados obtenidos con cada uno.

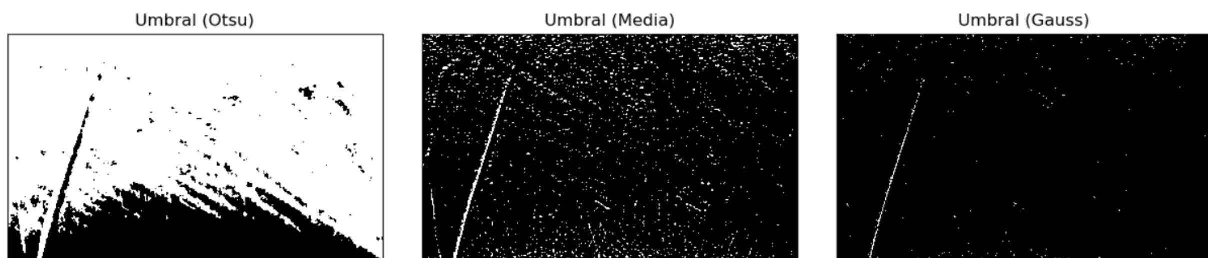

Figura 5.19 – Binarización de imagen suavizada.

5. En el quinto paso se obtiene la segunda máscara, denominada **mascara** 2. Esto al aplicar el detector de bordes de Canny a la imagen suavizada del segundo paso. La figura 5.20 muestra el resultado de esta operación, así como la de otros detectores de bordes que también fueron evaluados.

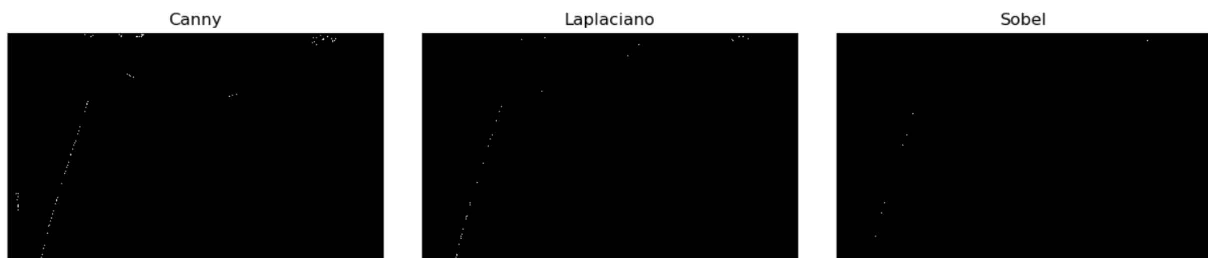

Figura 5.20 – Detectores de bordes aplicados sobre imagen suavizada.

6. En este paso se utilizan la **máscara 1** obtenida por umbralizado y la **máscara 2** obtenida por detección de bordes para crear una máscara combinada. Solamente aquellos pixeles que aparezcan en ambas máscaras serán incluidos en la máscara combinada. Esta operación da la seguridad de que esos pixeles fueron encontrados como irregularidades por dos métodos distintos. Este método es semejante al propuesto por [26], pero en lugar de utilizar distintos umbrales, se utilizan distintas técnicas de umbralizado. Una vez que se crea la máscara combinada se somete a una operación morfológica de dilatación con el fin de manipular más fácilmente las regiones detectadas. Esto podemos observarlo en la figura 5.21.

Figura 5.21 – Combinación de la máscara por umbral y máscara de bordes.

7. El último paso de esta etapa de procesamiento consiste en identificar el rasgo característico de los defectos y filtrar todas las regiones tomando como base ese rasgo. Al analizar las regiones correspondientes a defectos, se pudo observar que todas ellas tenían áreas mayores a 200 pixeles, mientras que las regiones generadas por ruido generalmente tenían áreas menores. De modo que el rasgo utilizado para filtrar fue el área de cada región. Todas las regiones con áreas superiores a 200 pixeles serán consideradas defectos. Una vez identificados los defectos, se procede a marcarlos en color amarillo en la imagen de entrada, generando así la imagen de salida, tal como se puede apreciar en la figura 5.22. La salida de esta etapa es un vector que contiene los defectos encontrados.

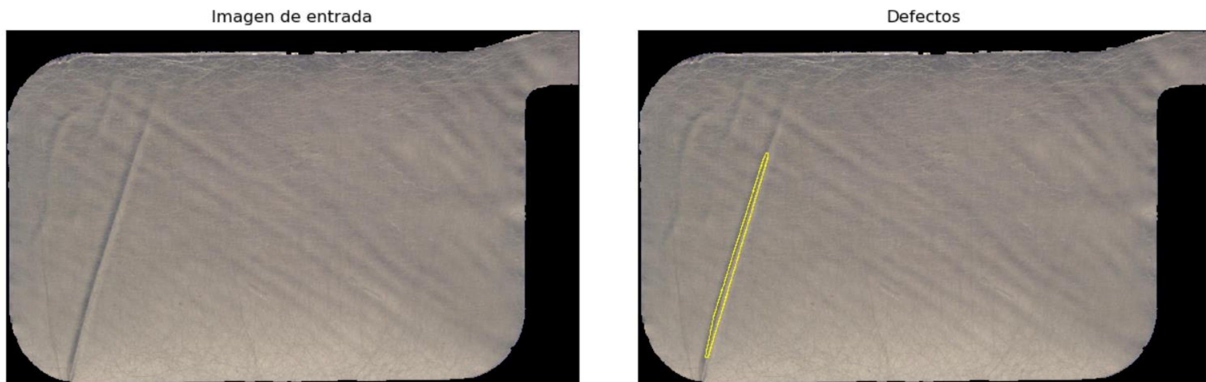

Figura 5.22 – Imagen de entrada e Imagen de salida con localización de defectos.

VI. RESULTADOS

Para la evaluación del algoritmo desarrollado se utilizaron en total 82 muestras, de las cuales 29 no tenían defectos y 53 tenían al menos un defecto. El algoritmo aplicado a las muestras consta de tres etapas principales:

- La de segmentación, encargada de ubicar y extraer el electrodo.
- La de detección de defectos en bordes, enfocada en localizar los defectos de aluminio expuesto y de-laminación.
- La de detección de defectos en superficie, encargada de localizar el resto de los defectos de contaminación, rasgado, burbuja y aluminio expuesto.

Ya que cada etapa abarca múltiples tipos de defectos, los resultados serán analizados por etapas y no por tipos de defectos. Evaluando primeramente el desempeño de cada etapa, y finalmente el desempeño del algoritmo de detección en su totalidad.

El Anexo 6 incluye la tabla A6.1 que contiene los detalles de cada muestra, que tipo de defectos contiene y que resultado se obtuvo en cada etapa del procesamiento para dicha muestra.

Se utiliza la tabla 6.1 para definir los cuatro posibles resultados del proceso de detección de defectos y que pueden ser utilizados para la evaluación del desempeño de cada etapa.

Tabla 6.1 – Cuatro posibles resultados del proceso de detección.

Resultado \ Defecto	**Presente**	**No presente**
Detectado	Verdadero positivo (VP)	Falso positivo (FP)
No detectado	Falso negativo (FN)	Verdadero negativo (VN)

Los cuatro posibles resultados para la evaluación de una imagen están dados por las combinaciones del resultado obtenido: defecto detectado y defecto no detectado; y de la imagen de entrada: defecto presente y defecto no presente. De modo que cada combinación produce un resultado:

- Verdadero positivo (VP). La imagen de entrada contenía defectos y el algoritmo los detectó correctamente. Este sería el escenario real para toda imagen con defectos.
- Falso positivo (FP). La imagen no contenía defectos pero el algoritmo ha detectado defectos que no son reales. Si bien este escenario se trata de un error, sería el menos grave, ya que no implica el escape de defectos reales. Sin embargo, puede representar pérdidas monetarias si algún electrodo sin defectos es tomado como un electrodo defectuoso.
- Falso negativo (FN). La imagen de entrada sí contenía defectos, pero el algoritmo no los ha detectado. Este sería el más crítico de los errores, pues puede suponer que algún electrodo con defectos reales sea tomado como un electrodo bueno y, por ende, que sea empacado y vendido.
- Verdadero negativo (VN). La imagen de entrada no contenía defectos y el algoritmo no ha detectado defectos. Este sería el escenario ideal para toda imagen sin defectos.

6.1 Resultados de la Etapa de Segmentación

Al analizar los resultados obtenidos en la primera etapa de segmentación. El programa que fue capaz de identificar y aislar de manera correcta el electrodo en 76 de las 82 imágenes de entrada, lo que representaría una eficiencia del 93%. Esto se ve reflejado en la figura 6.1, en azul se indican las muestras para las que el algoritmo obtuvo un resultado correcto, mientras que en naranja se indican aquellas para las que falló.

En las 6 imágenes en las que la segmentación del electrodo falló, se podría considerar que una superficie de alrededor del 95% del electrodo fue ubicada, pero las regiones de los bordes se perdieron. De modo que se consideran fallas en la etapa de segmentación. Dado que esta etapa genera la información de entrada para las siguientes etapas, solamente 76 de las 82 imágenes, aquellas correctamente segmentadas, pudieron ser procesadas por el resto del algoritmo, de modo que el universo de muestras considerado en el resto de los resultados será de 76 muestras.

Figura 6.1 – Grafica de los resultados obtenidos en la etapa 1 de procesamiento.

6.2 Resultados la Etapa de Detección de Defectos en Bordes

En esta etapa y las posteriores a analizar se hará uso del término “procesar las imágenes de forma correcta” para referirse al hecho de identificar y localizar los defectos en aquellas imágenes que presentan alguno, pero también, asegurar que el algoritmo de detección no identifique defectos en aquellas imágenes que no contienen ninguno.

La segunda etapa de detección de defectos en bordes logró procesar de manera correcta 46 de las 76 imágenes de entrada, lo que representa una eficiencia del 61%, tal como se muestra en la figura 6.2. Este número sirve para dar una idea general de la eficiencia de esta etapa y si bien puede parecer bajo, por sí solo no brinda suficiente información del desempeño.

Figura 6.2 – Grafica de los resultados obtenidos en la etapa 2 de procesamiento.

Si se separan los resultados obtenidos en la etapa 2 en dos categorías, una ellas para las imágenes que no contiene defectos y otra para aquellas que sí los contienen, se obtendrían los resultados que se muestra en la figura 6.3.

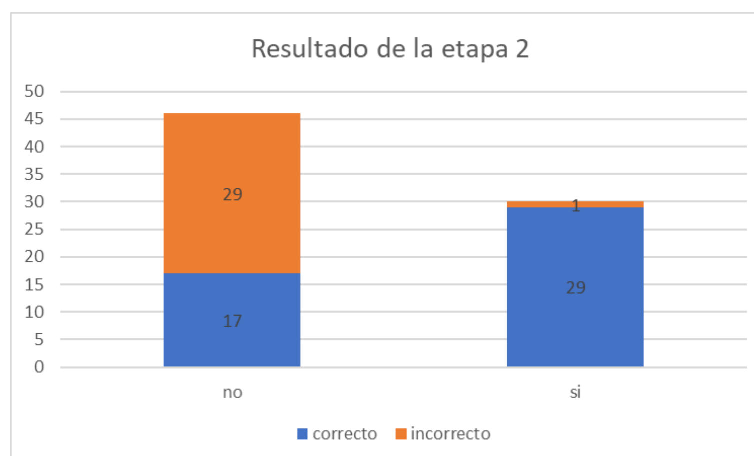

Figura 6.3 – Grafica de los resultados obtenidos en la etapa 2 de procesamiento separados por tipo de muestra.

Se puede observar que en color azul se tienen los resultados VN y VP, mientras que en anaranjado tenemos los FP y FN. Si obtenemos los porcentajes de cada uno, obtendríamos:

- 96.7% de VP.
- 3.3% de FN.
- 37 % de VN.
- 63% de FP.

Dado que el porcentaje de VP es realmente alto, se puede deducir que el algoritmo tiene una gran capacidad de detección de defectos en bordes. Por esta razón los FN, que son el tipo de error más peligroso, se encuentran en un rango bastante bajo, de 3.3%. Esto indica una baja probabilidad de no detectar un defecto. El alto porcentaje de FP constituye el mayor problema de esta etapa, pues indica la sensibilidad al ruido y una tendencia a detectar como defecto cosas que no lo son. Si bien, se mencionó previamente que este tipo de errores son los que tienen menor impacto, ya que no representan riesgos de uso de un electrodo defectuoso, pero sí el descartar un electrodo que posiblemente no tenía problemas. Idealmente se desea que sean cercanos a cero.

6.3 Resultados de la Etapa de Detección de Defectos en Superficie

La tercera etapa de detección de defectos en superficie logró procesar de manera correcta 54 de las 76 imágenes de entrada, lo que representa una eficiencia del 71%, como puede apreciarse en la figura 6.4.

Figura 6.4 – Gráfica de los resultados obtenidos en la etapa 3 de procesamiento.

Tal como se realizó en la etapa previa, se pueden desglosar los resultados de esta etapa apoyándonos en la tabla 6.1. La figura 6.5 muestra los resultados clasificados en imágenes con y sin defectos, además de indicar en azul las imágenes que fueron procesadas correctamente para cada grupo y en naranja las que fueron procesadas de manera incorrecta.

Resumiendo el resultado de cada grupo en porcentajes se obtendría lo siguiente:

- 90.3% de VP.
- 9.7% de FN.
- 57.8 % de VN.
- 42.2% de FP.

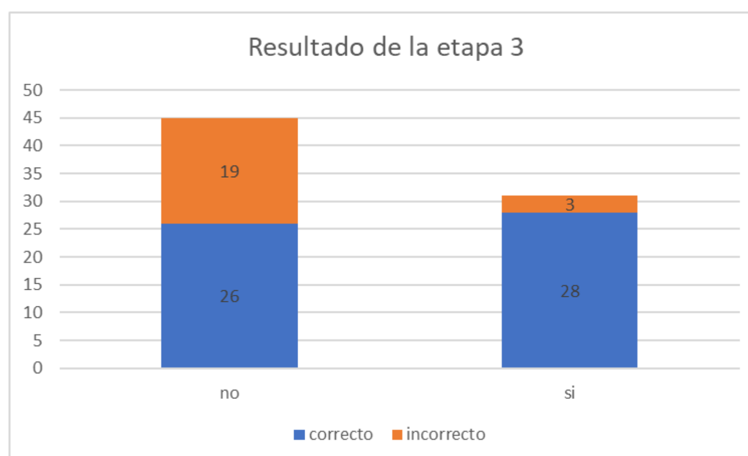

Figura 6.5 – Grafica de los resultados obtenidos en la etapa 3 de procesamiento, separados por tipo de muestra.

Si se camparan estos porcentajes, con los alcanzados en la etapa previa, se puede apreciar que la cantidad de VP y FN es prácticamente la misma, indicando una alta capacidad de detección de defectos internos y un bajo riesgo de escapes en la detección. Por otro lado, el porcentaje de VN es 20% mayor que en la etapa previa y, por ende, el porcentaje de FP es 20% menor, indicando que esta etapa es menos sensible al ruido y tiende a realizar menos falsas detecciones.

6.4 Resultados Generales

Además de analizar los resultados proporcionados por cada etapa, se puede considerar el algoritmo de detección como un conjunto y evaluar el resultado que entrega, sin importar la naturaleza de los defectos y la etapa en la que hayan sido detectados. Solamente juzgando si la imagen de entrada contenía un defecto, o no, y si el algoritmo detecto algún defecto, o no. Basados en esta premisa, 49 de las 76 imágenes, fueron procesadas correctamente, tal como lo indica la figura 6.6, esto corresponde a un 64% de eficiencia.

Figura 6.6 – Grafica de los resultados generales obtenidos por el algoritmo de detección.

De igual manera que en cada etapa de procesamiento, el porcentaje de eficiencia por sí solo brinda una idea general sobre el desempeño del algoritmo, pero realmente contiene poca información específica, de modo que se analizaron los resultados desglosados. Este desglose se realizó basados en la tabla 6.1 y se utilizaron porcentajes para cada grupo. Esto puede apreciarse en la figura 6.7, donde cada columna corresponde a un tipo de imagen (con o sin defecto), donde el color azul indica las imágenes en las que el resultado obtenido por el algoritmo fue correcto, mientras que el color anaranjado indica las imágenes en las que el resultado fue erróneo.

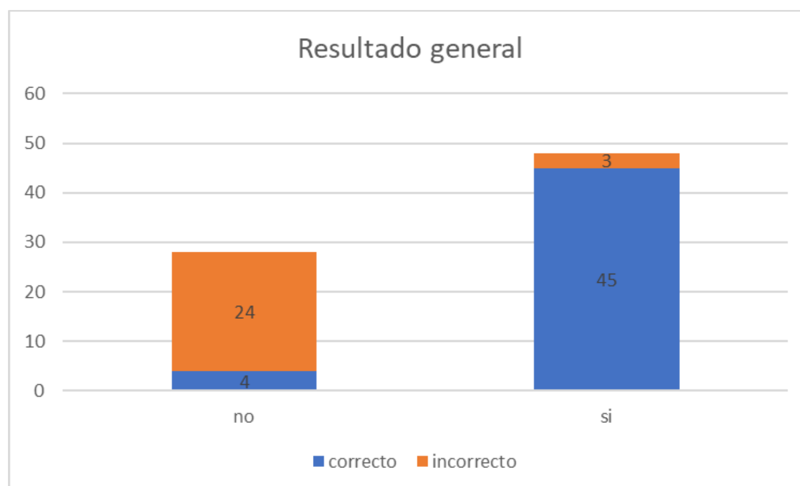

Figura 6.7 – Gráfica de los resultados generales obtenidos por el algoritmo de detección, separados por tipo de muestra.

Los porcentajes correspondientes a cada grupo de resultados son:

- 93.8% de VP.
- 6.2% de FN.
- 14.3% de VN.
- 85.7% de FP.

En estos porcentajes se puede apreciar la misma tendencia que en cada etapa de procesamiento, es decir, un alto nivel de detección con bajas probabilidades de no detectar un defecto. Pero un alto número de falsas detecciones con muy alta probabilidad de detectar un defecto donde no lo hay. Este porcentaje en los resultados generales es incluso más alto que en los resultados observados en cada etapa y se debe a la combinación de las susceptibilidades a ruido de cada etapa.

VII. CONCLUSIONES

Con el presente trabajo se comprobó la factibilidad de desarrollar un sistema embebido de visión para la localización de defectos en la superficie de gel de electrodos de dispersión, pues como se menciona en la Sección 6.4, el sistema desarrollado fue capaz de detectar distintos tipos de defectos en diversas muestras con una eficiencia de 93.8%. Es importante mencionar, que aun cuando el sistema cumple con el objetivo planteado, es decir, es capaz de detectar los defectos y presenta un alto grado de detección, debe considerarse que tiene una gran área de oportunidad, referente al alto porcentaje de falsos positivos, actualmente 85.7%.

Durante la elaboración del presente proyecto y durante la etapa de análisis de resultados se detectaron varios factores que influyen de manera directa sobre este alto porcentaje. Algunos de ellos están relacionados entre sí, algunos impactan en mayor medida que otros, pero con seguridad, el abordarlos puede conducir a una mejora considerable en el sistema.

Múltiples cámaras o múltiples imágenes. En algunas de las imágenes adquiridas pudo observarse que el algoritmo detectó defectos en bordes cuando estos no estaban presentes en la muestra física. Esto se debió a la forma en la que la cámara captaba los bordes de la imagen. La figura 7.1 ilustra un escenario de un electrodo con aluminio expuesto. Como puede apreciarse, la cámara toma la imagen desde arriba y en la vista frontal podemos observar que existen dos bordes, uno dado por el aluminio y otro por el gel.

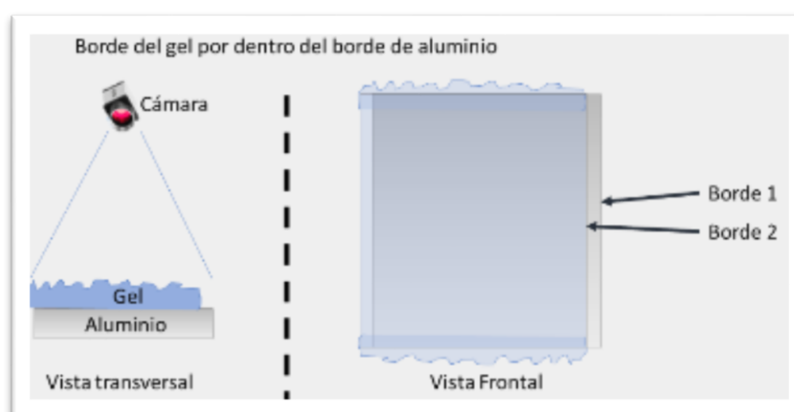

Figura 7.1 – Borde de gel por dentro del borde de aluminio.

El algoritmo de detección hace uso de estos bordes para identificar este tipo de defectos. El problema de los falsos positivos se da en muchos de los electrodos que no tiene ningun tipo de defecto, pues los bordes del gel tienden a terminar junto con los bordes de alumino. En ocasiones lo sobrepasa, de modo que la cámara, ubicada en el centro del electrodo, sigue percibiendo dos bordes, esto se ilustra en la figura 7.2.

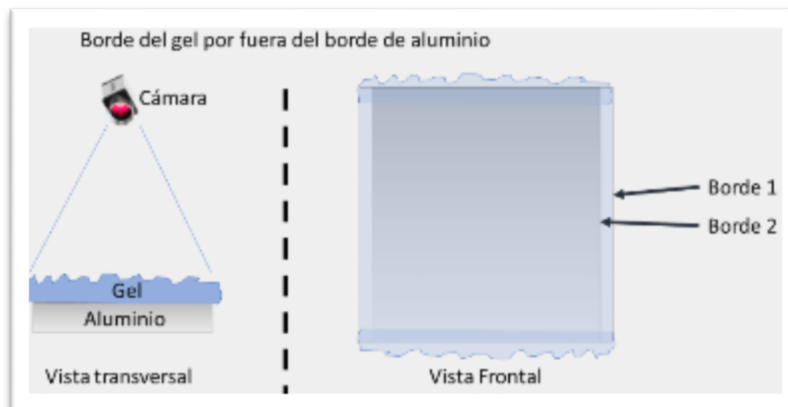

Figura 7.2 – Borde de gel por fuera del borde de aluminio.

A pesar de tratarse de un electrodo sin un defecto real, en las etapas de extracción de bordes el efecto producido es el mismo que producirá un electrodo con aluminio expuesto, se detectan dos bordes paralelos y, por ende, el algoritmo tiende a detectar esto como un defecto. Dos opciones que pueden ser evaluadas para mitigar este problema son: la adquisición de múltiples imágenes con una misma cámara, pero utilizando diferentes posiciones, semejante al método propuesto en [33] y la segunda opción es utilizar múltiples cámaras y localizarlas por encima del electrodo, pero por fuera de su superficie. La idea que se persigue con ambas propuestas es adquirir las imágenes de modo que el ángulo de visión de la cámara no genere el efecto de doble borde en los electrodos que no tiene defectos.

Método de sujeción de la muestra durante la adquisición de la imagen. Este fue probablemente el punto que más efecto tuvo en los falsos positivos detectados durante el proyecto. Debido a la naturaleza flexible de los electrodos de dispersión, resulta complicado mantenerlos totalmente planos en alguna superficie durante la adquisición de las imágenes.

También hay que agregar el hecho de que la superficie de gel no puede ser tocada, por lo que la sujeción debe realizarse a través del papel adhesivo, tal como se muestra en la figura 7.3.

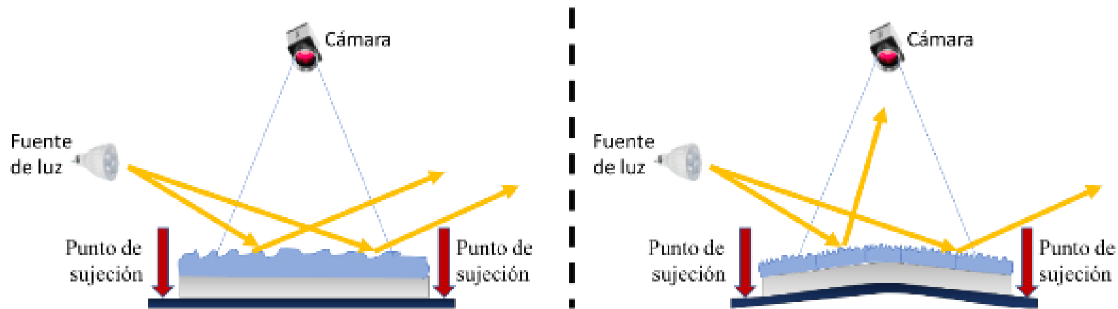

Figura 7.3 – Efecto de un electrodo sin sujetar correctamente.

Por más que se intente mantener el electrodo paralelo al plano sobre el cual se está tomando la imagen, resulta imposible utilizando solamente los puntos de sujeción en el papel adhesivo y esto genera variaciones en la luz que incide sobre la superficie y, por lo tanto, respuestas incorrectas por parte del algoritmo de detección. En este proyecto se utilizaron imanes para sujetar las cuatro esquinas del electrodo sobre un cristal. Este método fue efectivo más no infalible, ya que no se tuvo ninguna sujeción en el área central. Otras opciones, como la sujeción con vacío o succión, pueden ser evaluadas para mitigar este problema.

Iluminación simétrica. Como se mencionó en la Sección 4.3, se realizaron múltiples pruebas para identificar el mejor tipo de iluminación. El método empleado de iluminación de campo oscuro, con una luz cálida, un ángulo bajo y una distancia de trabajo relativamente grande, funcionó bien en determinados escenarios. Pero tal como se describe en el punto anterior y en la figura 7.2, cuando la sujeción del electrodo no es la adecuada, la iluminación deja de ser uniforme en la superficie. Una posible solución a este problema sería utilizar una iluminación simétrica, es decir, dos fuentes de luz una frente a la otra, de modo que si se presenta alguna variación en la superficie del electrodo el efecto sea igual en ambos lados. El inconveniente de esta solución es que solo se mitigarían los cambios en un eje de la superficie del electrodo, aquel eje que se ubique entre las dos lámparas.

Aumentar la cantidad de muestras. La cantidad de muestras utilizadas en este proyecto (82), está lejos de ser la ideal. El reducido número de muestras que se pudo obtener limitó la

información disponible de algunos de los tipos de defectos. Entre mayor sea la cantidad de muestras mejor será la caracterización que se realice de los defectos. Y entre más acertada sea la caracterización mejor será la respuesta del algoritmo. De modo que este punto sin duda puede mejorar el resultado obtenido.

VIII. REFERENCIAS BIBLIOGRÁFICAS

[1] B. L. Hainer y R. B. Usatine, «Electrosurgery for the Skin,» *AMERICAN FAMILY PHYSICIAN,* pp. 1259-1266, 2002.

[2] A. De La Torre y J. Cordova, Principios Fisicos de la Unidad Electroquirurgica, Mexico: Editorial Medica Panamericana, 2009.

[3] J. Lewis, «Introduction to Machine Vision,» 2014.

[4] B. Dipert, «Industrial Automation and Embedded Vision: A Powerful Combination,» *InTech Magazine,* 2014.

[5] Basler, «Embedded Vision,» 2017.

[6] E. Gadelmawla, «A Vision system for Surface Roughness Characterization Using the Gray Level Co-occurrence Matrix,» *NDT & E International,* pp. 577-588, 2004.

[7] S. Satorres, J. Gómez, J. Gámez y A. Sánchez, «A Machine vision System for Defect Characterization on Transparent Parts with Non-plane Surfaces,» *Machine Vision and Applications,* pp. 1-13, 2012.

[8] J. LI, M. Parker y Z. Hou, «An Intelligent System for Real Time Automatic Defect Inspection on Specular Coated Surfaces,» *Proceedings of SPIE - The International Society for Optical Engineering,* 2005.

[9] M. G. M. Fuentes, «Electrocirugía: Fundamentos,» *Servicio de Obstetricia y Ginecología, Hospital Universitario Virgen de las Nieves, Granada,* 2011.

[10] J. M. Juran y A. B. Godfrey, Juran's Quality Handbook, McGraw-Hill, 1998.

[11] K. Demaagd, A. Oliver, N. Oostendorp y K. Scott, Practical Computer Vision, O'Reilly Media, 2012.

[12] T. Noergaard, Embedded Systems Architecture, Massachusetts, US: Newnes, 2005.

[13] Argon Design, «Embedded vision systems set to revolutionise electronics,» 22 Enero 2014. [En línea]. Available: http://www.argondesign.com/news/2014/jan/22/embedded-vision-systems/. [Último acceso: 20 Marzo 2018].

[14] F. Hunter, S. Biver y P. Fuqua, Light - Science & Magic, Burlington: Elsevier, 2007.

[15] Panavision, «Sensor Size & Field of View,» 2015.

[16] The Pyhton Software Foundation, «What is Python? Executive Summary,» The Python Software Foundation, [En línea]. Available: https://www.python.org/doc/essays/blurb/. [Último acceso: 07 Junio 2019].

[17] J. MIllman y M. Aivazis, «Python for Scientists and Engineers,» *Computing in Science & Engineering,* pp. 9-12, 2011.

[18] OpenCV Team, «Open Source Computer Vision,» OpenCV, [En línea]. Available: https://opencv.org/about/. [Último acceso: 2019 Junio 07].

[19] K. Pulli y A. Baksheev, «Real-Time Computer Vision with OpenCV,» *Communications of The ACM,* vol. 55, nº 6, 2012.

[20] OpenCV, «Introduction,» OpenCV, 07 Abril 2019. [En línea]. Available: https://docs.opencv.org/4.1.0/d1/dfb/intro.html. [Último acceso: 10 Junio 2019].

[21] J. Ke, C. Duan, W. Yi y C. Yan, «Application of an adaptive two-wave mixing interferometer for detection of surface defects,» de *Progress In Electromagnetic Research Symposium (PIERS)*, Shanghai, China, 2016.

[22] Y. Shimizu, Y. Matsuno, Y. Ohba y W. Gao, «Micro Thermal Sensor for Nanometric Surface Defect Inspection,» de *16th International Conference on Nanotechnology* , Sendai, Japon, 2016.

[23] M. Win, A. R. Bushroa, M. A. Hassan, N. M. Hilman y A. Ide-Ektessabi, «A Contrast Adjustment Thresholding Method for Surface Defect Detection Based on Mesoscopy,» *IEEE Transactions on Industrial Informatics,* pp. 642-649, 2015.

[24] L. Yuan, Z. Zhang y X. Tao, «The Development and Prospect of Surface Defect Detection Based on Vision Measurement Method,» de *2016 12th Congress on Intelligent Control and Automation (WCICA)*, Guilin, China, 2016.

[25] X.-Q. Huang, X.-B. Luo y R.-Z. Wang, «A Real-time Parallel Combination Segmentation Method for Aluminum Surface Defect Images,» de *2015 International Conference on Machine Learning and Cybernetics*, Guangzhou, China , 2015.

[26] A. K. Dubey y Z. A. Jaffery, «Maximally Stable Extremal Region Marking-Based Railway Track Surface Defect Sensing,» *IEEE Sensors Journal,* vol. 16, nº 24, pp. 9047-9052, 2016.

[27] J. Tan, L. Li, Y. Wang, F. Mo, J. Chen, L. Zhao y Y. Xu, «The Quality Detection of Surface Defect in Dispensing Dack-End Based on HALCON,» de *2016 International Conference on Cybernetics, Robotics and Control*, Hong Kong, China, 2016.

[28] S. Merlo, F. Carpignano, D. Riccardi, G. Rigamonti y M. Norgia, «Infrared Structured Light Generation by Optical MEMS and Application to Depth Perception,» de *2017 IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, Padua, Italia, 2017.

[29] H. Jiang y Z. Song, «A Robust Feature Detector Algorithm for the Binary Encoded Single-Shot Structured Light System,» de *IEEE International Conference on Information and Automation*, Ningbo, China, 2016.

[30] H.-G. Maas, «Robust Automatic Surface Reconstruction with Structured Light,» *International Archives of Photogrammetry and Remote Sensing,* vol. 29, p. 709–713, 1993.

[31] R. Yang, S. Cheng, W. Yang y Y. Chen, «Robust and Accurate Surface Measurement Using Structured Light,» *IEEE Transactions on Instrumentation and Measurement,* vol. 57, nº 6, pp. 1275-1280, 2008.

[32] F. Li, S. Gao, G. Shi, Q. Li, L. Yang, R. Li y X. Xie, «Single Shot Dual-Frequency Structured Light Based Depth Sensing,» *IEEE Journal of Selected Topics in Signal Processing,* vol. 9, nº 3, pp. 384-395, 2015.

[33] S. Zhou, D. Liang y Y. Wei, «Automatic Detection of Metal Surface Defects Using Multi-angle Lighting Multivariative Image Analysis,» de *IEEE International Conference on Information and Automation*, Ningbo, China, 2016.

[34] K. Liu, H. Wang, H. Chen, E. Qu, Y. Tian y H. Sun, «Steel Surface Defect Detection Using a New Haar–Weibull-Variance Model in Unsupervised Manner,» *IEEE Transactions on Instrumentation and Measurement,* vol. 66, nº 10, pp. 2585-2596, 2017.

[35] glidernet, «CPU Boards - performance comparison,» 07 Febrero 2018. [En línea]. Available: http://wiki.glidernet.org/cpu-boards. [Último acceso: 11 Agosto 2018].

[36] E.M.Smith, «Interesting Benchmark Of SBC Boards,» 01 Diciembre 2018. [En línea]. Available: https://chiefio.wordpress.com/2018/12/01/interesting-benchmark-of-sbc-boards/. [Último acceso: 15 Enero 2019].

[37] Basler, «Using Single Board Computers (SBCs) with Basler USB3 Vision and GigE VIsion Cameras,» Basler, 2017.

[38] D. Franklin, «NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge,» NViDIA, 07 Marzo 2017. [En línea]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/. [Último acceso: 20 Octubre 2018].

[39] Nvidia Developer, «Download and Install JetPack,» Nvidia Developer, 2018. [En línea]. Available: https://docs.nvidia.com/jetson/archives/jetpack-archived/jetpack-33/index.html#jetpack/3.3/install.htm%3FTocPath%3D_____3. [Último acceso: 18 Enero 2019].

[40] A. Rosebrock, «Ubuntu 16.04: How to install OpenCV,» pyimagesearch, 24 Octubre 2016. [En línea]. Available: https://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/. [Último acceso: 30 Enero 2019].

[41] JetsonHacks, «NVPModel – NVIDIA Jetson TX2 Development Kit,» JetsonHacks, 25 Marzo 2017. [En línea]. Available: https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/. [Último acceso: 05 Febrero 2018].

ANEXO 1. ARCHIVO MAIN.PY

Este archivo contiene la función principal del proyecto.

```
##################################################################################
#### Librerías
##################################################################################
    import cv2
    import numpy as np
    import glob
    from matplotlib import pyplot as plt
    import threading
    import time
    import math
    import common
    import segmentation
    import outterDefects
    import innerDefects
##################################################################################
#### Definición de funciones
##################################################################################
    #
    # Función para detectar la presión de teclas en los objetos plt
    #
    def Plt_KeyPress(event):
```

```
    global keyPressed
    if(len(str(event.key)) == 1 or event.key == 'escape'):
        keyPressed = event.key
    else:
        keyPressed = ' '


#
# Esta función se encarga de redibujar las imágenes en pantalla
#
def ReDrwaImages():
    # Se obtienen las imágenes para desplegar, dependiendo de la opción
    # seleccionada (stageToDisplay)
    if(common.stageToDisplay >= len(common.imagesToDisplay)):
        _stageToDisplay = common.STAGE0
    else:
        _stageToDisplay = common.stageToDisplay
    if (common.DEBUG):
        _imagesToDisplay = common.imagesToDisplay[_stageToDisplay]
    else:
        _key = common.keys[_stageToDisplay]
        _value = common.imagesToDisplay[_stageToDisplay][_key]
        _imagesToDisplay = {_key:_value}
```

```
# Se verifica el número de columnas y renglones para el arreglo de imágenes
# en pantalla
qtyImages = len(_imagesToDisplay)
if(int(qtyImages/3)>=1):
    cols = 3
else:
    cols=qtyImages%3
if (qtyImages%3):
    rows = int(qtyImages/3) + 1
else:
    rows = int(qtyImages/3)


fig.clear()
# Revisamos imágenes, una a la vez
for count, (key, image) in enumerate(_imagesToDisplay.items(),1):
    # Se recibió un histograma, se agrega de forma distinta
    if(image.size == 256):
        plt.subplot(rows,cols,count)
        plt.plot(image)
        if(common.threshold != 0):
            plt.axvline(x=common.threshold,linewidth=4,color='r')
            plt.text(common.threshold-10,max(image)/2,str(common.threshold), \
                rotation=90)
```

```
        # Se agregan las imágenes
        else:
            plt.subplot(rows,cols,count)
            plt.imshow(image, 'gray')
            plt.title(key)
            plt.xticks([]),plt.yticks([])

    # Finalmente se despliegan
    if(common.processToDisplay == common.SEGMENTATION):
        process = 'Segmentación. '
    elif(common.processToDisplay == common.OUTTER_DEFECTS):
        process = 'Defectos en bordes. '
    elif(common.processToDisplay == common.INNER_DEFECTS):
        process = 'Defectos en superficie. '
    elif(common.processToDisplay == common.FINAL_IMAGE):
        process = 'Resultado final. '
    else:
        process = 'ERROR. '
    mainTitle = str(imageName) +'. ' + process + 'Paso: '+ str(_stageToDisplay)
    fig.suptitle(mainTitle)
    fig.canvas.draw()

#
```

```
# Esta función ejecuta el procesamiento, haciendo uso de las librerías externas
#
def UpdateImages():
    contour, imgROI = segmentation.Segmentation(imgBGR)
    imgOutterDeffects = outterDefects.EdgeDefects(contour, imgROI)
    imgInnerDeffects = innerDefects.SurfaceDefects(contour, imgROI)

    imgOut = imgROI.copy()
    imgOut[imgOutterDeffects[:,:] == 255] = (255,0,0)
    imgOut[imgInnerDeffects[:,:] == 255] = (255,255,0)

    if(common.processToDisplay == common.FINAL_IMAGE):
        common.keys = ['Imagen de entrada']
        common.imagesToDisplay = \
            [{'Imagen de entrada':imgROI, \
                'Imagen de salida':imgOut}]

    ReDrwaImages()

#
# Función principal
#
if __name__ == "__main__":
```

```
# Inicialización de variables
common.init()
exit = False
keyPressed = ' '
if (common.USE_CAMERA):
    # Adquiere el control de la cámara
    camera = cv2.VideoCapture(0)
else:
    # Obtiene el nombre de todas las imágenes a procesar
    images = glob.glob(common.PATH+common.EXTENSION)
    count = 0

# Configura la figura que desplegara las imágenes en pantalla
fig = plt.gcf()
fig.canvas.mpl_connect('key_press_event', Plt_KeyPress)

# Ciclo principal de ejecución, mientras no se presione la tecla de
# terminación, el programa seguirá ejecutándose
while (not(exit)):
    # Adquisición de imagen
    if (common.USE_CAMERA):
        # Adquiere un frame de la cámara
        print("Tomando imágenes de cámara")
```

```
    ret, imgBGR = camera.read()
    imageName = 'Imagen adquirida'
else:
    # Adquiere una imagen del disco duro
    imageName = images[count]
    count += 1
    if(count >= len(images)):
        count = 0
    print("Tomando imágenes de disco duro: ", imageName)
    imgBGR = cv2.imread(imageName)

# Procesa la imagen
UpdateImages()


# Ejecuta un ciclo esperando entradas del usuario
continueRunning = True
updateRequired = False
while (continueRunning):
    # Pausa requerida para interactuar con objetos plt
    try:
        plt.pause(0.2)
    except:
        exit = True
```

```
    break

# Se procesa la entrada del teclado

# Opción para termina la ejecución del programa
if (keyPressed == common.KEY_INPUTS['exit']):
    continueRunning= False
    exit = True

# Opción para procesar siguiente imagen
elif (keyPressed == common.KEY_INPUTS['next']):
    continueRunning= False

# Opción para procesar nuevamente ultima la imagen
elif (keyPressed == common.KEY_INPUTS['refresh']):
    updateRequired = True

# Opción para encender/apagar modo de desarrollo
elif (keyPressed == common.KEY_INPUTS['debug']):
    common.DEBUG = not(common.DEBUG)
    print("Debug: ", common.DEBUG)
    updateRequired = True
```

```
        # Opciones para desplegar algún paso del procesamiento
        for stage, key in common.KEY_INPUTS['stages'].items():
            if (keyPressed == key):
                common.stageToDisplay = int(stage)
                ReDrwaImages()
            else:
                pass

        # Opciones para desplegar alguna etapa procesamiento
        for process, key in common.KEY_INPUTS['processes'].items():
            if (keyPressed == key):
                common.processToDisplay = int(process)
                updateRequired = True
            else:
                pass
        keyPressed = ' '
        if (updateRequired):
            # De ser necesario reprocesa la imagen
            UpdateImages()
            updateRequired = False
    if (common.USE_CAMERA):
        # Libera la cámara
        camera.release()
```

ANEXO 2. ARCHIVO SEGMENTATION.PY

Este archivo contiene la función de segmentación.

```
################################################################################
#### Librerías
################################################################################
    import cv2
    import numpy as np
    import glob
    from matplotlib import pyplot as plt
    import threading
    import time
    import math
    import common
################################################################################
#### Definición of funciones
################################################################################
    #
    # Esta función recibe un arreglo con etiquetas correspondientes a blobs
    # cada uno tiene uno etiqueta distinta. Regresa la etiqueta correspondiente al
    # blob más cercano al centro
    #
    def GetCenterLabel(labelsArray, backgroundLabel = 0):
        output = 0
```

```
for label in np.unique(labelsArray):
    # La etiqueta -1 corresponde a los bordes
    if (label < 0 or label == backgroundLabel):
        continue

    # Se crea una máscara para cada región
    mask = np.zeros(labelsArray.shape, dtype="uint8")
    mask[(labelsArray == label)] = 255

    # Se calcula el centroide de cada region
    moments = cv2.moments(mask)
    x = moments['m10'] / moments['m00']
    y = moments['m01'] / moments['m00']

    # Se calcula a la distancia del cendroide al punto central de la imagen
    distance = \
        math.sqrt((x - common.CENTER.X) ** 2 + (y - common.CENTER.Y) ** 2)

    # Se selecciona la región si es cercana al centro
    if ( distance < common.CENTER.DISTANCE_TRESHOLD):
            output = label
            break
```

```
        return output


    #

    # Esta función se encarga de realizar el proceso de segmentación,

    # @contours regresa un arreglo con los puntos del contorno del electrodo

    # @ROI sección de la imagen que contiene el electrodo

    def Segmentation(inputImage):

        # Tiempo de inicio de procesamiento

        startTime = time.time()

################################################################################

        # Paso 0. Imagen de entrada

################################################################################

        imgRGB = cv2.cvtColor(inputImage,cv2.COLOR_BGR2RGB)

        imgHSV = cv2.cvtColor(imgRGB,cv2.COLOR_RGB2HSV)


        _keys = ['Imagen de entrada']

        _imagesToDisplay = [{_keys[common.STAGE0]:imgRGB}]

################################################################################

        # Paso 1. Transformar a espacio de color RGB, y HSV, para visualizar

        # los canales de forma independiente

################################################################################
```

```
    imgB = imgRGB[:,:,2].copy()
    imgH = imgHSV[:,:,0].copy()

    if(common.DEBUG):
        imgR = imgRGB[:,:,0].copy()
        imgG = imgRGB[:,:,1].copy()
        imgS = imgHSV[:,:,1].copy()
        imgV = imgHSV[:,:,2].copy()
    else:
        imgR = imgG= imgS = imgV = []

    _keys += ['Matiz (H)']
    _imagesToDisplay += [{'Rojo (R)':imgR, 'Verde (G)':imgG, 'Azul (B)':imgB, \
        'Matiz (H)':imgH, 'Saturacion (S)':imgS, 'Valor (V)':imgV}]

    imgOutS1 = []
    imgOutS1.append(imgB)
    imgOutS1.append(imgH)

################################################################################
    # Paso 2. Se aplica un filtro de suavizado a los canales H y B

################################################################################
```

```
_size = 15
imgMedian = cv2.medianBlur(imgOutS1[common.OPTION1],_size)
imgMedian2 = cv2.medianBlur(imgOutS1[common.OPTION2],_size)

if(common.DEBUG):
    imgBilateral = cv2.bilateralFilter(imgOutS1[common.OPTION1],_size,75,75)
    imgGaussian = cv2.GaussianBlur(imgOutS1[common.OPTION1],(_size,_size), 0)
    imgBilateral2 = cv2.bilateralFilter(imgOutS1[common.OPTION2],_size,75,75)
    imgGaussian2 = cv2.GaussianBlur(imgOutS1[common.OPTION2],(_size,_size), 0)
else:
    imgBilateral = imgGaussian = imgBilateral2 = imgGaussian2 = []

_keys += ['Filtro de Mediana']
_imagesToDisplay += [{'Filtro de Mediana':imgMedian, \
    'Filtro Bilateral':imgBilateral, 'Filtro Gaussiano':imgGaussian, \
    'Filtro de Mediana 2':imgMedian2, 'Filtro Bilateral 2':imgBilateral2, \
    'Filtro Gaussiano 2':imgGaussian2}]

imgOutS2 = []
imgOutS2.append(imgMedian)
imgOutS2.append(imgMedian2)
```

```
################################################################################
    # Paso 3. Se obtiene el histograma de la imagen

################################################################################
    histogram =cv2.calcHist([imgOutS2[common.OPTION1]], [0], None, [256], [0,256])


    _keys += ['Histograma']
    _imagesToDisplay += [{_keys[common.STAGE3]:histogram}]

################################################################################
    # Paso 4. Se aplica un umbralizado a la imagen obtenida en el paso previo

################################################################################
    _size = 21
    _c = 3
    common.threshold,imgBinOtsu = cv2.threshold(imgOutS2[common.OPTION1],0,255,\
      cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
    threshold2,imgBinOtsu2 = cv2.threshold(imgOutS2[common.OPTION2],0,255,\
      cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)


    if(common.DEBUG):
      imgBinMedian = cv2.adaptiveThreshold(imgOutS2[common.OPTION1],255,\
              cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY,_size,_c)
      imgBinGauss = cv2.adaptiveThreshold(imgOutS2[common.OPTION1],255,\
```

```
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY,_size,_c)
  else:
    imgBinMedian = imgBinGauss = []

  _keys += ['Umbral (Otsu)']
  _imagesToDisplay += [{'Umbral (Otsu)':imgBinOtsu, \
    'Umbral (Media)':imgBinMedian, \
    'Umbral (Gauss)':imgBinGauss}]

  imgOutS4 = []
  imgOutS4.append(imgBinOtsu)
  imgOutS4.append(imgBinOtsu2)

################################################################################
  # Paso 5. Se aplica operadores morfológicos a la imagen resultante
  # del paso anterior

################################################################################
  _m = cv2.getStructuringElement(cv2.MORPH_RECT,(17,17))

  imgClosing = cv2.morphologyEx(imgOutS4[common.OPTION1],
cv2.MORPH_CLOSE, _m, \
    iterations = 2)
  imgOpening = cv2.morphologyEx(imgClosing, cv2.MORPH_OPEN, _m,iterations = 2)
```

```
imgClosing2 = cv2.morphologyEx(imgOutS4[common.OPTION2],
cv2.MORPH_CLOSE, _m, \
    iterations = 1)
imgOpening2 = cv2.morphologyEx(imgClosing2, cv2.MORPH_OPEN, _m, \
    iterations = 2)
ret, markers2 = cv2.connectedComponents(imgOpening2)

# Se identifica el área que pertenece al fondo
imgBackground = cv2.dilate(imgOpening,_m, iterations = 2)

# Se identifica el área que pertenece al objeto
imgDistanceTransform = cv2.distanceTransform(imgClosing,cv2.DIST_L2,5)

ret, imgForeground = cv2.threshold(imgDistanceTransform, \
    0.8*imgDistanceTransform.max(),255,cv2.THRESH_BINARY)
imgForeground = np.uint8(imgForeground)

# Se identifican el área de los bordes, puede pertenecer al fondo u objeto
imgBorders = cv2.subtract(imgBackground,imgForeground)

# Se crea un marcador para cada componente identificado
ret, markers = cv2.connectedComponents(imgForeground)
```

```
    markers = markers+1
    markers[imgBorders==255] = 0

    # Se genera mapa de colores para las áreas identificadas
    if(common.DEBUG):
        imgHMarkers = np.uint8(179*markers/np.max(markers))
        imgHSVMarkers = cv2.merge([imgHMarkers, \
            np.ones(imgHMarkers.shape, dtype="uint8")*255, \
            np.ones(imgHMarkers.shape, dtype="uint8")*255])
        imgRGBMarkers = cv2.cvtColor(imgHSVMarkers, cv2.COLOR_HSV2RGB)
        imgRGBMarkers[imgHMarkers==0] = 0
    else:
        imgRGBMarkers = []

    _keys += ['Frente (DT)']
    _imagesToDisplay += [{'Cierre':imgClosing, 'Apertura':imgOpening, \
        'Fondo':imgBackground, 'DT':imgDistanceTransform, \
        'Frente (DT)':imgForeground, 'Areas':imgRGBMarkers}]

################################################################################
    # Paso 6. Se crean una máscara utilizando el algoritmo de cuencas hidrográficas

################################################################################
```

```
input1  = cv2.cvtColor(imgOutS2[common.OPTION1],cv2.COLOR_GRAY2RGB)
labels1 = cv2.watershed(input1,markers)
labels2 = markers2

_imagesToDisplay += [{}]

if(common.DEBUG):
    # Se crea una máscara, originada desde el canal B
    mask = np.zeros(imgOutS2[common.OPTION1].shape, dtype="uint8")
    mask[(labels1 == GetCenterLabel(labels1,1))] = 255
    _imagesToDisplay[common.STAGE6][('Máscara 1')] = mask.copy()

    # Se crea una máscara, originada desde el canal H
    mask = np.zeros(imgOutS2[common.OPTION1].shape, dtype="uint8")
    mask[(labels2 == GetCenterLabel(labels2))] = 255
    _imagesToDisplay[common.STAGE6][('Máscara 2')] = mask.copy()

# Se crea una máscara combinando las máscaras creadas del canal azul y el
# canal de matiz
mask = np.zeros(imgOutS2[common.OPTION1].shape, dtype="uint8")
mask[(labels1 == GetCenterLabel(labels1,1))] = 255
mask[(labels2 == GetCenterLabel(labels2))] = 255
```

```
_imagesToDisplay[common.STAGE6][('Máscara Combinada ')] = mask

# Se aplica la máscara de mayor área, que corresponde al electrodo
imgRGBMasked = cv2.bitwise_and(imgRGB, imgRGB, mask = mask)

# Se obtiene el contorno de la máscara
if(common.Is_cv4()):
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL , \
        cv2.CHAIN_APPROX_NONE)
else:
    _, contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL , \
        cv2.CHAIN_APPROX_NONE)
cv2.drawContours(imgRGB, contours, -1, (0,255,0), 3)

# A partir del contorno se identifica le región de interés (ROI)
# Se ajusta la imagen y contornos a las coordenadas de la ROI
x,y,w,h = cv2.boundingRect(contours[0])
cv2.rectangle(imgRGB,(x,y),(x+w,y+h),(255,0,0), 6)
contour = contours[0] - np.array([np.array([np.array([x,y])])])
imgROI = imgRGBMasked[y:y+h,x:x+w].copy()

_keys += ['ROI']
_imagesToDisplay[common.STAGE6]['ROI'] = imgROI
```

```
################################################################################
    # Prepara imagen para mostrarla en pantalla
################################################################################
    if(common.processToDisplay == common.SEGMENTATION):
        common.keys = _keys
        common.imagesToDisplay = _imagesToDisplay

    # Tiempo de procesamiento
    print("Termina procesamiento Segmentacion. ", time.time()-startTime, " seg transcurridos.")

    # Regresa contornos y región de interés
    return contour, imgROI
```

ANEXO 3. ARCHIVO INNERDEFECTS.PY

Este archivo contiene las funciones para la detección de defectos internos.

```
################################################################################
#### Librerías
################################################################################
    import cv2
    import numpy as np
    import glob
    from matplotlib import pyplot as plt
    import threading
    import time
    import math
    import common
################################################################################
#### Definición de funciones
################################################################################
    #
    # Esta función verifica si existe algún defecto en los bordes del electrodo
    #
    def SurfaceDefects(contour, imgRGB):
        # Tiempo de inicio de procesamiento
        startTime = time.time()
```

```
################################################################################
    # Paso 0. Imagen de entrada
################################################################################
    _keys = ['Imagen de entrada']
    _imagesToDisplay = [{_keys[common.STAGE0]:imgRGB}]


################################################################################
    # Paso 1. Se obtienen el área de trabajo, a partir del rectángulo que
    # contiene la superficie del electrodo. Se transforma la imagen de entrada
    # a escala de grises.
################################################################################
    imgGray = cv2.cvtColor(imgRGB, cv2.COLOR_RGB2GRAY)

    _heigth, _width = imgGray.shape
    _centerY = int(_heigth/2)
    _centerX = int(_width/2)
    _offset = 100

    for _x in range(_centerX):
      if( imgGray[_centerY][_x] > 0):
```

```
        left = _x
        break

for _x in range(_width-1,_centerX,-1):
    if( imgGray[_centerY][_x] > 0):
        right = _x
        break

for _y in range(_centerY):
    if( imgGray[_y][_centerX] > 0):
        top = _y
        break

for _y in range(_heigth-1,_centerY, -1):
    if( imgGray[_y][_centerX] > 0):
        bottom = _y
        break

top += _offset
bottom -= _offset
left += _offset
right -= _offset
```

```
        imgInput = imgGray.copy()
        cv2.rectangle(imgInput, (left,top), (right,bottom),255,4)
        imgROI = imgGray[top:bottom,left:right].copy()

        _keys += ['Área de Interés']
        _imagesToDisplay += [{'Identificación de área de interés':imgInput, \
            'Área de Interés':imgROI}]


##############################################################################
        # Paso 2. Se aplican filtros de suavizado a la ROI.
##############################################################################
        _size = 11

        imgMedian = cv2.medianBlur(imgROI,_size)

        if(common.DEBUG):
            imgBilateral = cv2.bilateralFilter(imgROI,_size, 150, 150)
            imgGaussian = cv2.GaussianBlur(imgROI,(_size,_size),cv2.BORDER_CONSTANT)
        else:
            imgBilateral = imgGaussian = []
```

```
    _keys += ['Filtro de Mediana']
    _imagesToDisplay += [{'Filtro Gaussiano':imgGaussian, \
       'Filtro de Mediana':imgMedian, 'Filtro Bilateral':imgBilateral}]


################################################################################
    # Paso 3. Se obtiene el histograma de la imagen
################################################################################
    histogram =cv2.calcHist(imgMedian, [0], None, [256], [0,256])
    if(common.processToDisplay == common.INNER_DEFECTS):
       common.threshold = 0
    _keys += ['Histograma']
    _imagesToDisplay += [{'Histograma':histogram}]


################################################################################
    # Paso 4. Se binariza la imagen
################################################################################
    _size = 21
    _c = 3

    imgBinGauss = cv2.adaptiveThreshold(imgMedian,255,\
```

```
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,_size,_c)

        if(common.DEBUG):
            imgBinMedian = cv2.adaptiveThreshold(imgMedian,255,\

cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY_INV,_size,_c)
            _,imgBinOtsu = cv2.threshold(imgMedian,0,255,\
            cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
        else:
            imgBinMedian = imgBinOtsu = []

        _keys += ['Umbral (Gauss)']
        _imagesToDisplay += [{'Umbral (Otsu)':imgBinOtsu, \
            'Umbral (Media)':imgBinMedian, \
            'Umbral (Gauss)':imgBinGauss}]


##############################################################################
        # Paso 5. Se aplican detectores de bordes

##############################################################################
        canny = cv2.Canny(imgMedian,0,40,apertureSize=3, L2gradient=True)
        if(common.DEBUG):
```

```
        laplacian64 = cv2.Laplacian(imgMedian,cv2.CV_64F)
        laplacian = np.uint8(np.absolute(laplacian64))
        _,laplacianBin = cv2.threshold(laplacian,5,255,cv2.THRESH_BINARY)

        sobel64 = cv2.Sobel(imgMedian,cv2.CV_64F,1,1,-1)
        sobel = np.uint8(np.absolute(sobel64))
        _,sobelBin = cv2.threshold(sobel,5,255,cv2.THRESH_BINARY)
    else:
        laplacianBin=sobelBin=[]

    _keys += ['Canny']
    _imagesToDisplay += [{'Canny':canny, 'Laplaciano':laplacianBin, \
        'Sobel':sobelBin}]


###############################################################################
    # Paso 6. Se combinan las máscaras creadas por umbralización y por bordes

###############################################################################
    _m = cv2.getStructuringElement(cv2.MORPH_RECT,(3,3))
    maskCanny  = cv2.dilate(canny,_m, iterations = 2)
    maskGauss  = cv2.dilate(imgBinGauss, _m, iterations=2)
```

```
    imgMask = cv2.bitwise_and(maskGauss,maskCanny)


    _keys += ['Mascara final']

    _imagesToDisplay += [{'Mascara final':imgMask}]



################################################################################
    # Paso 7. Se filtran los elementos detectados en base a su área, y se

    # obtienen los contornos de cada uno

################################################################################
    num_labels, markers, stats, centroids = \

        cv2.connectedComponentsWithStats(imgMask , connectivity=8)


    thdAreas = (stats[:,cv2.CC_STAT_AREA]>200)

    for label in range(num_labels):

        # La etiqueta 0 corresponde a al fondo

        if (label == 0):

            continue

        if(thdAreas[label] != True):

            imgMask[(markers == label)] = 0


    imgMask = cv2.dilate(imgMask, _m, iterations=3)
```

```
mask = np.zeros(imgGray.shape, dtype="uint8")
mask[top:bottom,left:right] = imgMask

# Se obtienen los contornos de la máscara
if(common.Is_cv4()):
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL , \
        cv2.CHAIN_APPROX_SIMPLE)
else:
    _, contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL , \
        cv2.CHAIN_APPROX_SIMPLE)

maskOut = np.zeros(imgGray.shape,dtype="uint8")

for i in range(len(contours)):
    hull = cv2.convexHull(contours[i], False)
    cv2.drawContours(maskOut, [hull], 0, 255, 3)

imgOut = imgRGB.copy()
imgOut[maskOut[:,:] == 255] = (255,255,0)

_keys += ['Defectos']
_imagesToDisplay += [{'Imagen de entrada':imgRGB,'Defectos':imgOut}]
```

```
################################################################################
        # Prepara imagen para mostrarla en pantalla

################################################################################
        if(common.processToDisplay == common.INNER_DEFECTS):
            common.imagesToDisplay = _imagesToDisplay
            common.keys = _keys

        # Tiempo de procesamiento
        print("Termina procesamiento Superficie. ", time.time()-startTime, " seg transcurridos.")

        return maskOut
```

ANEXO 4. ARCHIVO OUTTERDEFECTS.PY

Este archivo contiene las funciones para la detección de defectos en los bordes.

```
##################################################################################
#### Librerías
##################################################################################
    import cv2
    import numpy as np
    import glob
    from matplotlib import pyplot as plt
    import threading
    import time
    import math
    import common
##################################################################################
#### Definición de funciones
##################################################################################
    #
    # Esta función verifica si existe algún defecto en los bordes del electrodo
    #
    def EdgeDefects(contour, imgRGB):
        # Tiempo de inicio de procesamiento
        startTime = time.time()
```

```
################################################################################
        # Paso 0. Imagen de entrada

################################################################################
        _keys = ['Imagen de entrada']
        _imagesToDisplay = [{_keys[common.STAGE0]:imgRGB}]

################################################################################
        # Paso 1. Se obtienen la región cercana a las orillas del electrodo,
        # esta área se determina a partir de los contornos (argumento de entrada).
        # Se aplica un filtro de suavizado a esta región.

################################################################################
        _size = 11
        imgGray = cv2.cvtColor(imgRGB,cv2.COLOR_RGB2GRAY)


        imgGaussian = cv2.GaussianBlur(imgGray,(_size,_size),cv2.BORDER_CONSTANT)
        imgMedian = cv2.medianBlur(imgGray,_size)
        imgBilateral = cv2.bilateralFilter(imgGray,_size, 150, 150)


        _keys += ['Filtro Bilateral']
        _imagesToDisplay += [{'Filtro Gaussiano':imgGaussian, \
            'Filtro de Mediana':imgMedian, 'Filtro Bilateral':imgBilateral}]
```

```
################################################################################
    # Paso 2. Se aplican detectores de bordes en la región cercana al contorno
################################################################################
    _contourWidth = 20
    canny = cv2.Canny(imgBilateral,0,40,apertureSize=3, L2gradient=True)

    if(common.DEBUG):
        laplacian64 = cv2.Laplacian(imgGaussian,cv2.CV_64F)
        laplacian = np.uint8(np.absolute(laplacian64))
        _,laplacianBin = cv2.threshold(laplacian,5,255,cv2.THRESH_BINARY)

        sobel64 = cv2.Sobel(imgGaussian,cv2.CV_64F,1,1,-1)
        sobel = np.uint8(np.absolute(sobel64))
        _,sobelBin = cv2.threshold(sobel,5,255,cv2.THRESH_BINARY)
    else:
        laplacianBin=sobelBin=[]

    cannyRGB = np.zeros(imgRGB.shape, dtype="uint8")
    cannyRGB = cv2.drawContours(cannyRGB, contour, -1, (0,220,0), _contourWidth)
    cannyRGB[canny==255] = (255,0,0)
```

```
    _keys += ['Canny']
    _imagesToDisplay += [{'Canny':cannyRGB, 'Laplaciano':laplacianBin, \
        'Sobel':sobelBin}]


################################################################################
    # Paso 3. Se filtran bordes por tamaño y posición
################################################################################
    mask = np.zeros(imgGray.shape, dtype="uint8")

    canny = cv2.drawContours(canny, contour, -1, 0, _contourWidth)
    num_labels, markers, stats, centroids = \
        cv2.connectedComponentsWithStats(canny , connectivity=8)

    thdAreas = (stats[:,cv2.CC_STAT_HEIGHT]>10) &
(stats[:,cv2.CC_STAT_WIDTH]>10)
    for label in range(num_labels):
        # La etiqueta 0 corresponde al fondo
        if (label == 0):
            continue
        if(thdAreas[label] == True):
            mask[(markers == label)] = 255
```

```
    _m = cv2.getStructuringElement(cv2.MORPH_RECT,(3,3))
    mask  = cv2.dilate(mask,_m, iterations = 2)

    _keys += ['Canny Filtrado']
    _imagesToDisplay += [{'Canny Filtrado':mask}]


################################################################################
    # Paso 4. Identificación de defectos
################################################################################
    _squarenessFacor = 3.5
    num_labels, markers, stats, centroids = \
      cv2.connectedComponentsWithStats(mask , connectivity=8)

    maskOut = np.zeros(imgGray.shape,dtype="uint8")

    for label in range(num_labels):
      # La etiqueta 0 corresponde al fondo
      if (label == 0):
        continue
```

```
_x = stats[label][cv2.CC_STAT_LEFT]
_y = stats[label][cv2.CC_STAT_TOP]
_deltaX = stats[label][cv2.CC_STAT_WIDTH]
_deltaY = stats[label][cv2.CC_STAT_HEIGHT]
_centerX = int(_x + _deltaX/2)
_centerY = int(_y + _deltaY/2)
_area = stats[label][cv2.CC_STAT_AREA]
if (_deltaX > _deltaY):
    maxSide = _deltaX
    minSide = _deltaY
else:
    maxSide = _deltaY
    minSide = _deltaX

# Se verifica forma
if((maxSide/minSide < _squarenessFacor) or (_area>1500)):
    if(_deltaX < common.MIN_SIZE_X):
        _deltaX = common.MIN_SIZE_X
        _deltaY = common.MIN_SIZE_Y
        _x = _centerX - int(_deltaX/2)
        _y = _centerY - int(_deltaY/2)

    cv2.rectangle(maskOut, (_x,_y), (_x+_deltaX, _y+_deltaY),255,3)
```

```
    imgOut = imgRGB.copy()
    imgOut[maskOut[:,:] == 255] = (255,0,0)

    _keys += ['Defectos']
    _imagesToDisplay += [{'Imagen de entrada':imgRGB,'Defectos':imgOut}]


###############################################################################
    # Prepara imagen para mostrarla en pantalla
###############################################################################
    if(common.processToDisplay == common.OUTTER_DEFECTS):
        common.imagesToDisplay = _imagesToDisplay
        common.keys = _keys

    # Tiempo de procesamiento
    print("Termina procesamiento Bordes. ", time.time()-startTime, " seg transcurridos.")

    return maskOut
```

ANEXO 5. ARCHIVO COMMON.PY

Este archivo contiene definiciones de parámetros comunes usadas por todas las funciones.

```
################################################################################
#### Librerías
################################################################################
import cv2


################################################################################
#### Definición de funciones
################################################################################
#
# Esta función inicializa las variables globales que serán utilizadas en todo
# el proyecto. Debe llamarse solo una vez en main.py
#
def init():
    global DEBUG, \
        USE_CAMERA, \
        PATH, \
        DEBUG, \
        USE_CAMERA, \
        PATH, \
        EXTENSION, \
        STAGE0, \
```

```
STAGE1, \
STAGE2, \
STAGE3, \
STAGE4, \
STAGE5, \
STAGE6, \
STAGE7, \
STAGE8, \
STAGE9, \
stageToDisplay, \
SEGMENTATION, \
OUTTER_DEFECTS, \
INNER_DEFECTS, \
FINAL_IMAGE, \
processToDisplay, \
SEGMENTATION, \
OPTION1, \
OPTION2, \
imagesToDisplay, \
stageToDisplay, \
threshold, \
keys, \
KEY_INPUTS, \
```

```
    MIN_SIZE_X, \
    MIN_SIZE_Y


STAGE0 =        0
STAGE1 =        1
STAGE2 =        2
STAGE3 =        3
STAGE4 =        4
STAGE5 =        5
STAGE6 =        6
STAGE7 =        7
STAGE8 =        8
STAGE9 =        9


SEGMENTATION =    0
OUTTER_DEFECTS =  1
INNER_DEFECTS =   2
FINAL_IMAGE =     3


OPTION1 =        0
OPTION2 =        1


imagesToDisplay =   []
```

```
    keys =          []
    threshold =     0

    MIN_SIZE_X =    120
    MIN_SIZE_Y =    80
################################################################################
## Inicia sección configurable por usuario
#
################################################################################
    # Habilita el modo de Desarrollo
    DEBUG =         True

    # Habilita el uso de cámara, si la cámara no se utiliza, las imágenes serán
    # tomadas del disco duro
    USE_CAMERA =    False

    # Directorio para cargar imágenes (cuando no se obtienen de la cámara)
    PATH =          './imagenes/'
    EXTENSION =     '*.jpg' #Formato de las imágenes

    # Configuración de teclas para manipular el programa, las teclas deben escribirse
    # siempre entre comillas simples '', se debe evitar utilizar la misma tecla para
    # más de una función.
```

```
KEY_INPUTS =                {}
KEY_INPUTS['exit'] =        'escape'    # Termina la ejecución del programa
KEY_INPUTS['next'] =        'n'         # Siguiente imagen (captura o archivo)
KEY_INPUTS['refresh'] =     'r'         # Procesa nuevamente ultima imagen
KEY_INPUTS['debug'] =       'd'         # Habilita / deshabilita el modo debug

KEY_INPUTS['stages'] =      {}
KEY_INPUTS['stages']['0'] = '0'         # Despliega el paso 0 del procesamiento
KEY_INPUTS['stages']['1'] = '1'         # Despliega el paso 1 del procesamiento
KEY_INPUTS['stages']['2'] = '2'         # Despliega el paso 2 del procesamiento
KEY_INPUTS['stages']['3'] = '3'         # Despliega el paso 3 del procesamiento
KEY_INPUTS['stages']['4'] = '4'         # Despliega el paso 4 del procesamiento
KEY_INPUTS['stages']['5'] = '5'         # Despliega el paso 5 del procesamiento
KEY_INPUTS['stages']['6'] = '6'         # Despliega el paso 6 del procesamiento
KEY_INPUTS['stages']['7'] = '7'         # Despliega el paso 7 del procesamiento
KEY_INPUTS['stages']['8'] = '8'         # Despliega el paso 8 del procesamiento
KEY_INPUTS['stages']['9'] = '9'         # Despliega el paso 9 del procesamiento

KEY_INPUTS['processes'] =       {}
KEY_INPUTS['processes']['0'] = 'z'      # Despliega etapa de segmentación
KEY_INPUTS['processes']['1'] = 'x'      # Despliega etapa de defectos externos
KEY_INPUTS['processes']['2'] = 'c'      # Despliega etapa de defectos internos
KEY_INPUTS['processes']['3'] = 'v'      # Despliega la imagen final
```

```
    # Permite seleccionar el paso que se desplegara por defecto en la imagen de
    # salida, las opciones son STAGE0 - STAGE9
    stageToDisplay =            STAGE0

    # Permite seleccionar la etapa de procesamiento que se desplegara en la
    # imagen de salida, las opciones son SEGMENTATION, OUTTER_DEFECTS,
INNER_DEFECTS, FINAL_IMAGE
    processToDisplay =          FINAL_IMAGE

################################################################################
## Termina sección configurable
################################################################################

class CENTER:
    X =                     int(2592 / 2)
    Y =                     int(1944 / 2)
    DISTANCE_TRESHOLD = 350

class FORMAT:
    WIDTH =                 2591
    HEIGTH =                1943
```

```
#
# Función para diferenciar la versión de OpenCV (por problemas de
# compatibilidad) entre OpenCV4 y OpenCV3
#
def Is_cv4():
    (major, minor, _) = cv2.__version__.split(".")
    return (major == '4') and (minor != 0)
```

ANEXO 6. DESCRIPCIÓN DE MUESTRAS

La tabla A6.1 contiene la información de todas las muestras utilizadas durante el presente trabajo, además de los resultados obtenidos al procesar cada una de ellas.

Tabla A6.1 – Listado de muestras, descripción de sus defectos, y resultado de su procesamiento.

Nombre de imagen	**Defecto**	**Resultado de Segmentación**	**Defecto presente en bordes**	**Resultado de detección de bordes**	**Defecto presente en superficie**	**Resultado de detección de superficie**	**Defecto**	**Resultado general**
1.jpg	De-laminación	correcto	si	correcto	no	correcto	si	correcto
2.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
3.jpg	Ojo de pescado	correcto	no	correcto	si	correcto	si	correcto
4.jpg	Burbuja / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
5.jpg	Aluminio expuesto	correcto	si	correcto	no	correcto	si	correcto
6.jpg	Deformación	correcto	no	incorrecto	si	correcto	si	correcto
7.jpg	Rasgado / De-laminación	correcto	si	correcto	si	correcto	si	correcto
8.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
9.jpg	Contaminación / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
10.jpg	De-laminación	correcto	si	correcto	no	incorrecto	si	correcto
11.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
12.jpg	Contaminación / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
13.jpg	Deformación / Rasgado	correcto	si	correcto	si	correcto	si	correcto
14.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
15.jpg	Ojo de pescado	correcto	no	incorrecto	no	correcto	no	incorrecto
16.jpg	De-laminación / Burbuja	correcto	si	correcto	si	correcto	si	correcto
17.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
18.jpg	Contaminación	correcto	no	correcto	si	correcto	si	correcto
19.jpg	Rasgado	correcto	si	correcto	no	correcto	si	correcto
20.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto

21.jpg	Burbuja	correcto	no	correcto	si	correcto	si	correcto
22.jpg	Rasgado	correcto	si	correcto	no	incorrecto	si	correcto
23.jpg	Aluminio expuesto	correcto	si	correcto	no	correcto	si	correcto
24.jpg	Contaminación	correcto	no	correcto	si	correcto	si	correcto
25.jpg	De-laminación	correcto	si	correcto	no	incorrecto	si	correcto
26.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
27.jpg	Contaminación / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
28.jpg	De-laminación	correcto	si	correcto	no	incorrecto	si	correcto
29.jpg	Aluminio expuesto	correcto	si	correcto	no	incorrecto	si	correcto
30.jpg	Rasgado	correcto	no	correcto	si	correcto	si	correcto
31.jpg	Aluminio expuesto	correcto	si	correcto	no	incorrecto	si	correcto
32.jpg	Ojo de pescado	correcto	no	incorrecto	si	incorrecto	si	correcto
33.jpg	Burbuja / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
34.jpg	Contaminación / Aluminio expuesto	correcto	si	correcto	si	correcto	si	correcto
35.jpg		correcto	no	correcto	no	correcto	no	correcto
36.jpg	Rasgado	correcto	no	correcto	si	correcto	si	correcto
37.jpg	Aluminio expuesto	correcto	si	correcto	no	incorrecto	si	correcto
38.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
39.jpg	Rasgado	correcto	no	incorrecto	si	correcto	si	correcto
40.jpg	Aluminio expuesto	incorrecto						
41.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
42.jpg	Aluminio expuesto / Burbuja	incorrecto						
43.jpg	Aluminio expuesto / Burbuja	incorrecto						
44.jpg		incorrecto						
45.jpg	Burbuja	incorrecto						
46.jpg		correcto	no	correcto	no	incorrecto	no	incorrecto
47.jpg	Rasgado	correcto	no	incorrecto	si	correcto	si	correcto
48.jpg	Aluminio expuesto	correcto	si	correcto	no	correcto	si	correcto

49.jpg		correcto	no	correcto	no	correcto	no	correcto
50.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
51.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
52.jpg	Burbuja	correcto	no	correcto	si	correcto	si	correcto
53.jpg	De-laminación	correcto	si	correcto	no	correcto	si	correcto
54.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
55.jpg		correcto	no	correcto	no	correcto	no	correcto
56.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
57.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
58.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
59.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
60.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
61.jpg	Aluminio expuesto	correcto	si	correcto	no	correcto	si	correcto
62.jpg	Burbuja	correcto	si	correcto	no	correcto	si	correcto
63.jpg	Contaminación	correcto	no	correcto	si	incorrecto	si	incorrecto
64.jpg	Contaminación	correcto	no	correcto	si	correcto	si	correcto
65.jpg		correcto	no	correcto	no	correcto	no	correcto
66.jpg	Rasgado / De-laminación	correcto	si	correcto	si	correcto	si	correcto
67.jpg	Rasgado / Burbujas	correcto	si	correcto	si	correcto	si	correcto
68.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
69.jpg	Rasgado / Burbuja	correcto	no	correcto	si	correcto	si	correcto
70.jpg	Aluminio expuesto / Rasgado	correcto	si	correcto	si	correcto	si	correcto
71.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
72.jpg	Ojo de pescado	correcto	no	correcto	si	incorrecto	si	incorrecto
73.jpg	Rasgado	correcto	si	incorrecto	no	correcto	si	incorrecto
74.jpg		correcto	no	incorrecto	no	incorrecto	no	incorrecto
75.jpg	Contaminación	correcto	no	incorrecto	si	correcto	si	correcto
76.jpg	Deformación / Rasgado	correcto	si	correcto	si	correcto	si	correcto
77.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
78.jpg	Contaminación	correcto	no	incorrecto	si	correcto	si	correcto
79.jpg	Rasgado	correcto	si	correcto	no	correcto	si	correcto
80.jpg		correcto	no	incorrecto	no	correcto	no	incorrecto
81.jpg	Contaminación	correcto	no	correcto	si	correcto	si	correcto
82.jpg	Rasgado	incorrecto						