
**DEPARTAMENTO DE ELECTRÓNICA Y AUTOMÁTICA**

**FACULTAD DE INGENIERÍA – UNIVERSIDAD NACIONAL DE SAN JUAN**

**Proyecto Integrador Final**

# Detección de aberraciones en el Tomate

**Asignatura:** Complementos de Informática – Temas específicos de Control I

**Ingeniería Electrónica**

***Autores***

*Jalil, Simón – Registro 26514*

*Pelaez, Pablo – Registro 26506*

**1º Semestre**

**Año 2019**

# Contenido

# Tabla de Ilustraciones

# 1) INTRODUCCIÓN

Este documento presentará el desarrollo y la evaluación de un sistema de clasificación de tomates con el objetivo de buscar una selección adecuada para los estándares comerciales. Hoy en día, es de suma importancia en la industria agrícola poder detectar de forma automática los tomates defectuosos para una selección de calidad, es por eso que buscamos realizar el análisis de imágenes de distintos tomates haciendo uso de la biblioteca OpenCV, en el sistema operativo Linux.

Para minimizar el tiempo de procesamiento y la cantidad de información que debe ser tratada por un clasificador automático de tomates, se ha tomado en cuenta el número mínimo de variables que se consideran necesarias para realizar una evaluación objetiva de su calidad. Estas características son golpes y manchas.

## 1.1) Objetivos

El principal objetivo de este trabajo es detectar golpes y manchas en el tomate, haciendo uso del procesamiento de imágenes con las librerías de OpenCV. Para lograr dicho objetivo se integraron todos los contenidos vistos en las siguientes asignaturas:

1. Temas Específicos de Control I: Realizar la adquisición y procesamiento de imágenes para luego poder hacer uso de las distintas herramientas de OpenCV.
2. Complementos de Informática: Realizar el proyecto cumpliendo con los paradigmas de la programación orientada a objetos (POO).

# 2) DESARROLLO

Se realizó un trabajo fuera de línea, con imágenes propias que son leídas a través de un archivo .txt mediante la utilización del IDE Eclipse, junto con la librería OpenCV de C++, además se consideró la realización de un reporte considerando los defectos encontrados en cada uno de los tomates analizados.

## 2.1) Hardware.

### 2.1.2) Iluminador.

| | |
|---|---|
| El nombre del producto | LED infrarrojo |
| Número de modelo | FY-5336 |
| Brillo de la luz | Infrarrojos |
| Poder | 10mil |
| Si el impermeable | NO |
| Tensión de | DC12V |
| La corriente eléctrica | 240-250mA |
| Estructura de producto | Placa de PCB |
| El ángulo de la luz | 45 °-60 °, 90 ° |
| Arreglar lente | 8mm/6mm/4mm |
| Efectiva iluminación de | 5-20 metros |
| Rango de longitud de onda | 850nM |
| Cantidad de led | 36 |
| Vida Útil | 200000horas |

Fig. 1: Características generales del iluminador.

Fig. 2: Iluminador.

### 2.1.1) Cámara.

El trabajo fue realizado con la WEBCAM LOGITECH C170).

Fig. 3: Cámara utilizada.


Fig. 4: Características generales de la cámara.

**Ambos elementos fueron brindados por la cátedra.**

## 2.2) Software.

### 2.2.1) IDE Eclipse.

Se trabajó con la versión 3.8.1 montada sobre el SO Debian 9 de la distribución de Linux, y el compilador g++, que es un compilador de línea de órdenes que compila y enlaza programas en C++, generando el correspondiente archivo ejecutable.

### 2.2.3) Librerías OpenCV.

Se utilizó la versión 2.4.13.7. Con el uso de estas librerías se pudo aplicar todo tipo de métodos necesarios para el procesamiento, para detectar bordes, evaluar intensidad de píxeles, aplicar filtros, detectar contornos, áreas, etc.

## 2.3) Resolución e implementación del procesamiento.

La secuencia a seguir para el correcto procesamiento de las diferentes imágenes es la siguiente:

1. **Creación del archivo contenedor de paths:** Antes de ejecutar el programa, se debe crear en la carpeta del mismo un archivo .txt con el nombre de "UbicacionDeImagenes.txt", en el cual se deben almacenar los paths correspondientes a las imágenes a analizar.

**Tomates analizados:**

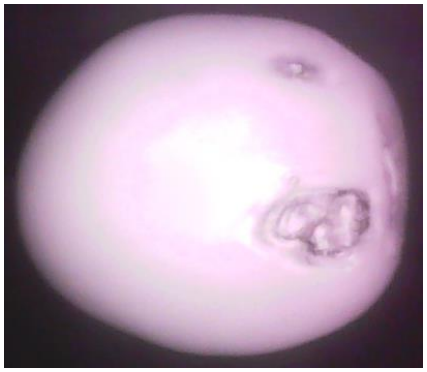

Fig. 5.a: Doble aberración

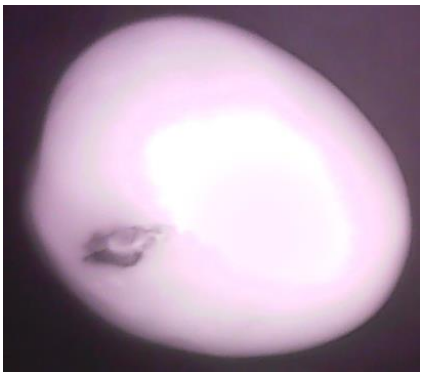

Fig. 6.a: Aberración lateral izquierdo.

Fig. 7.a: Aberración circular chica.

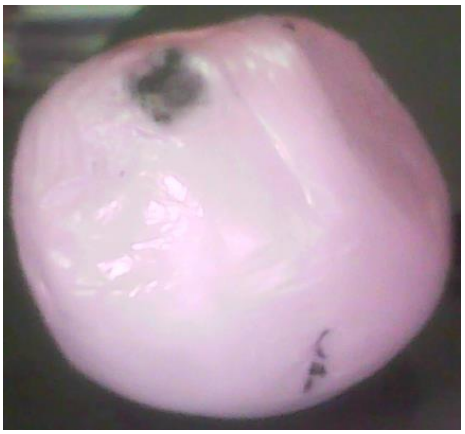

Fig. 8.a: Aberración más golpe.

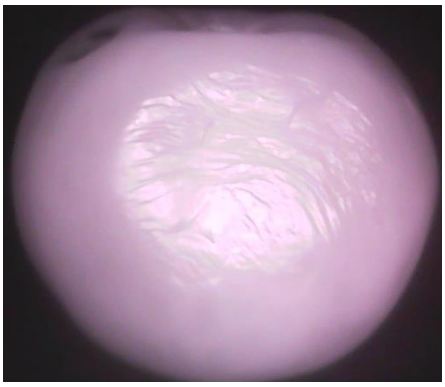

Fig. 9.a: Golpe visto de frente y mancha superior izquierda.

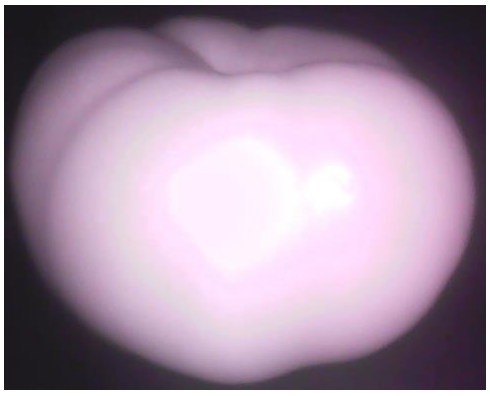

Fig. 10.a: Sin manchas ni golpes.

2. **Generación del archivo de parámetros YML:** Si es la primera vez que se va a usar el programa, se debe descomentar la función generaYML() (en el código aparece como comentario), para crear el mismo con los distintos parámetros definidos por defecto.

3. **Transformación de espacio de colores a imagen binaria:** Una vez almacenado los distintos paths de las imágenes en un contenedor, se procede a analizar una por una comenzando a obtener el área total del tomate en cuestión, por lo tanto, para obtener una mayor precisión del contorno del perímetro del mismo, se consideró umbralizar mediante la función de threshold para así poder obtener como resultado una imagen binaria con sus contornos bien marcados. Como elemento opcional para el usuario, se considera un acceso de calibración en cuanto a la zona de trabajo se refiere, lo cual es ajustado según el criterio del operario a utilizarlo. Por otro lado, desde el punto de

vista del hardware se consideró una superficie plana oscura, por lo que aumenta considerablemente las condiciones de la imagen binaria.

### Imágenes Binarias

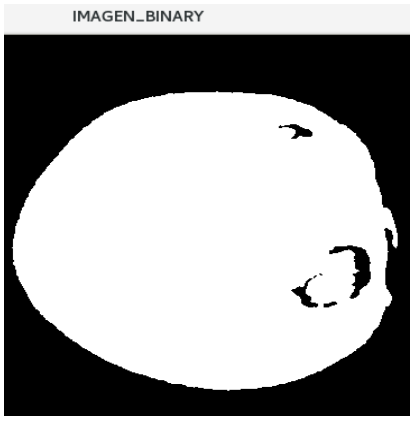


Fig 5.b: Imagen Binaria

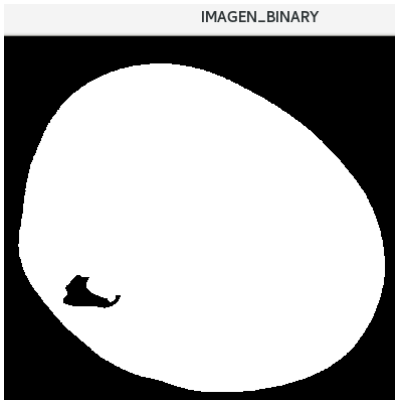


Fig 6.b: Imagen Binaria

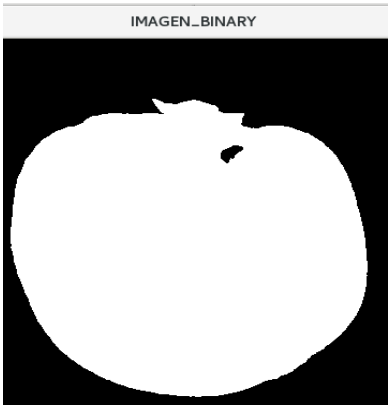


Fig. 7.b: Imagen Binaria

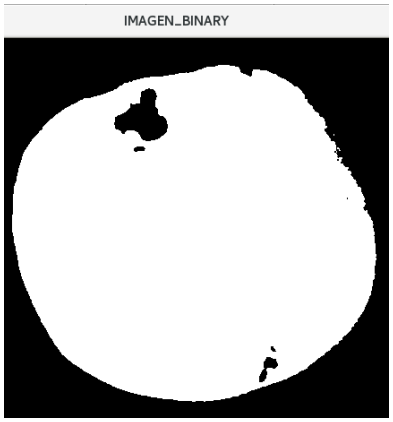


Fig 8.b: Imagen Binaria

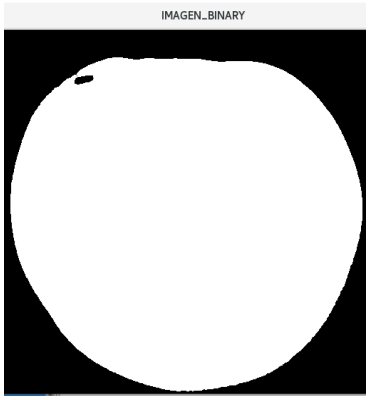


Fig 9.b: Imagen Binaria

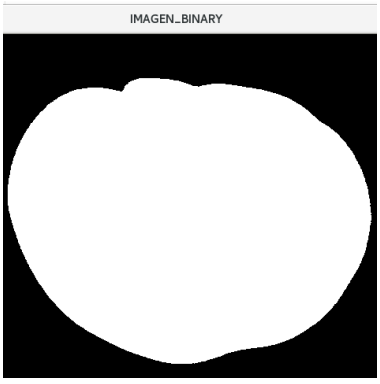


Fig 10.b: Imagen Binaria

```
//CALIBRACION (Opcional).
cout<<"\n\nDesea ajustar el thresh de la imagen? SI/NO: "<<endl;
cin>>calib;

if(calib == 0 ){
    threshcalib=85;//VALOR ELEGIDO POR DEFECTO. ESTARA INCLUIDO EN EL MANUAL DE USO.
}
if(calib == 1) {
    cv::Mat imgCalib;
    imgCalib = cv::Mat::zeros(img.rows, img.cols, img.type());
    cv::namedWindow("IMAGEN_DE_CALIBRACION", cv::WINDOW_AUTOSIZE);
    cv::createTrackbar("threshCalibracion", "IMAGEN_DE_CALIBRACION", &thresh, maxThresh, Procesamiento::onTrackbarThreshCalib, (void*)this);
    char c = (char)cv::waitKey(0);
    if (c == 'g' || c == 'G') {
        cout<<"\nEl nuevo valor del thresh elegido es: "<<auxThresh<<endl;
        cv::destroyWindow("IMAGEN_DE_CALIBRACION");
        }
    threshcalib= auxThresh;
    }

//REALIZO LA TRANSFORMACION A IMAGEN BINARIA.
cv::cvtColor(img, imgGray, cv::COLOR_BGR2GRAY);
cv::GaussianBlur(imgGray, imgGray, cv::Size(3, 3), 1, 1);
cv::threshold(imgGray,imgBinary,threshcalib,255,cv::THRESH_BINARY);
```

Fig. 11: Código para obtener imagen binaria.

4. **Detección de áreas de interés:** Teniendo la imagen binaria, utilizamos la función *findContours()* para encontrar los contornos de dicha imagen y de esa manera estar mas cerca de encontrar las áreas de interés. Este método opera de forma tal que si le brindamos una imagen binaria, nos devuelve como salida un vector de vector de puntos, donde engloba todos los puntos correspondientes a cada contorno de la matriz analizada. Realizado esto, se procede a encontrar las áreas de dichos contornos mediante el uso de la función *countourArea(),* la cual al introducirle un vector de vector de puntos, tenemos como resultado un vector con las correspondiente área en píxeles de cada contorno en cuestión. Por último, mediante un algoritmo de ordenación de burbuja se obtienen las áreas ordenadas, donde la primera corresponde al área total del tomate y las subsiguientes a las posibles aberraciones encontradas.

```
//ENCUENTRO LOS CONTORNOS
cv::findContours(imgBinary,contornos,cv::noArray(),cv::RETR_LIST, cv::CHAIN_APPROX_NONE);
vector<float> areas(contornos.size());

//DIBUJO LOS CONTORNOS EN LA IMAGEN ORIGINAL.
cv::drawContours(frame, contornos, -1, cv::Scalar(0, 255, 255), 3);

//ENCUENTRO EL AREA PARA CADA CONTORNO
for (unsigned int i = 0; i < contornos.size(); i++) {
    areas[i] = contourArea(contornos[i], false);
}


//ORDENO EL VECTOR DE AREAS.
cout << "\n AREAS ORDENADAS:\n";
ordenaAreas(areas);

for (unsigned int i = 0; i < areas.size(); i++) {
        cout << "\nEl Area N: " << i << " es: " << areas[i];
    }
```

Fig. 12: Código para encontrar áreas.

5. **Chequeo de umbral:** Ahora que tenemos el área, debemos saber si la misma cumple con los requisitos de un tomate y además para excluir otros posibles tomates dentro de una linea de producción en caso de extrapolar la aplicación. Para esto , se hace uso del archivo YML, el cual posee dichos parámetros (estos se pueden modificar directamente del mismo archivo según el criterio del operario). Si el área del tomate cumple con estos parámetros, la imagen sigue con el curso normal del procesamiento, caso contrario se exigirá al usuario volver a cargar la imagen calibrando de manera más eficaz la zona de trabajo de la misma.

6. **Chequeo de umbral de falla:** Una vez cumplido el umbral anterior, se analiza si el resto de las áreas del contenedor, se consideran o no fallas. Para esto, se usa un segundo umbral para seleccionar que áreas se añadirán al vector de fallas del objeto FallaDeTomate, lo cual este ultimo, a parte de tener este contenedor, posee la fecha de procesamiento y el área de tomate en cuestión; todo esto se utiliza posteriormente para ordenación y cálculo de aberraciones totales.

7. **Segmentación de la zona de trabajo:** Hecho todo esto, se procede a encontrar si el tomate posee o no un golpe, considerando esa zona como una aberración mas, subiendo su correspondiente área al vector de aberraciones. Para esto se estudia la imagen de manera mas minuciosa considerando una zona de trabajo, la cual es producida mediante el enmascaramiento de entre la imagen original y una mascara ajustada por un threshold mediante el método *bitwise_and().* Con esto logramos obtener una imagen del tomate, apartado de todo lo que lo rodea.

```
//SI EL CONTORNO DEL TOMATE CUMPLE CON LOS REQUISITOS DEL UMBRAL, SE ANALIZA LA MISMA, CASO CONTRARIO SE EXIGE OTRA MUESTRA.
if(MaxU > areas[0] && MinU < areas[0]){

    falla.setAreaTomate(areas[0]);

    //OBSERVO SI LA IMAGEN TIENE PRESCENCIAS DE ABERRACIONES CON OTRO UMBRAL PARA PODER HACER PUSHBACK DE ELLAS.
    for(unsigned int i = 1; i<areas.size(); i++){
        if(areas[i]<9000 && areas[i]>250)
            falla.nuevaAberracion(areas[i]);
    }
    cout << falla;

    //REALIZO EL ENMASCARAMIENTO PARA OBTENER LA ZONA DE TRABAJO.
    mask = cv::Mat::zeros(img.rows, img.cols, img.type());
    zonaTrabajo = cv::Mat::zeros(img.rows, img.cols, img.type());

    cv::threshold(frame, mask, 100, maxThresh, cv::THRESH_BINARY);
    cv::bitwise_and(frame, mask, zonaTrabajo);
    cv::drawContours(mask, contornos, -1, cv::Scalar(0, 255, 255), 2);
    cv::namedWindow("Zona de trabajo", cv::WINDOW_AUTOSIZE);
    cv::imshow("Zona de trabajo", zonaTrabajo);
    cv::waitKey(0);
    cv::destroyWindow("Zona de trabajo");

    area = areas[0];

    cv::waitKey(0);

}else{

    cout << "\nEl area del tomate es: " << areas[0];
    cout<<"\n\nEl tomate ingresado tiene un area que no corresponde al umbral calibrado. Vuelva a calibrar."<<endl;
    area = 1;
    falla.setAreaTomate(area);
    cv::waitKey(0);
```

Fig. 13: Código para encontrar zona de trabajo.

**Zonas de trabajo**

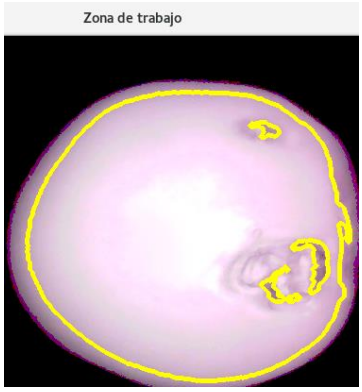


Fig. 5.c: Zona de Trabajo.


Fig. 6.c: Zona de Trabajo.

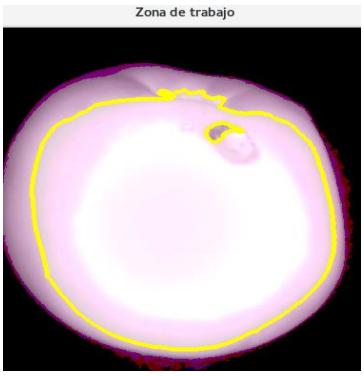


Fig. 7.c: Zona de Trabajo.

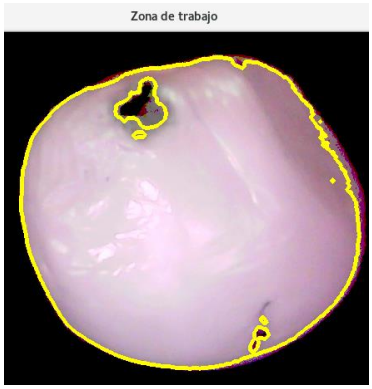


Fig. 8.c: Zona de Trabajo.

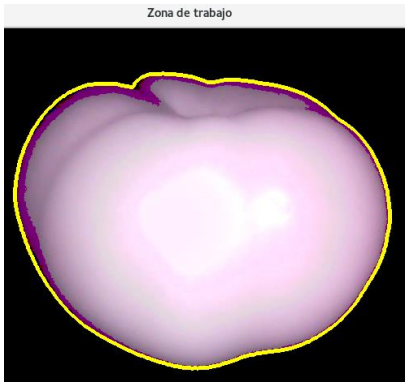


Fig. 10.c: Zona de Trabajo.

8. **Detección de golpe:** En este punto del proceso se probaron distintos métodos para alcanzar lo planteado. Una de las opciones que se analizó fue el hecho de observar la diferencia de intensidad que se veía reflejado en el golpe, pero, debido a las limitaciones existentes en el hardware no logramos un resultado óptimo que nos pueda ayudar a detectar dichos golpes. Descartado este planteo, nos proponemos en detectar la textura rugosa presente en la zona del golpe, que nos permita aproximar su área.

# 3) PARADIGMAS DE LA POO.

## 3.1) Paradigmas utilizados.

- **Clases:** Con la idea de hacer el código lo más reutilizable posible se crearon 3 distintas clases genéricas, las cuales son:

Clase Procesamiento: En ella se encuentran todos los métodos que ejecutan el procesamiento de las imágenes de los tomates; desde la carga de archivos, tanto YML para los parámetros, como de texto para los paths de imágenes; hasta la visualización de todo el procesamiento.

Clase FallaDeTomate: En esta se crean objetos que almacenan las aberraciones existentes en el tomate analizado. Además, se implementan los métodos para almacenar la fecha de procesamiento de las imágenes para poder luego establecer un ordenamiento.

Clase ContenedorFallaDeTomate: En esta clase se almacena en un contenedor todas las fallas de los tomates que fueron analizados. En el mismo tenemos un listado con todas las aberraciones encontradas, teniendo además métodos de ordenación en base al criterio del usuario mediante el uso de enums.

La justificación de la creación de tres clases distintas reside en la necesidad de ir teniendo un encapsulamiento por separado del procesamiento y de crear clases que se correspondan con los objetos creados, ya que no es propio de un objeto FallaDeTomate tener un método que se dedique al procesamiento de la imagen. Además, se debe cumplir el objetivo de realizar un código que sea lo más genérico posible, ayudando a su reutilización.

En las siguientes figuras se muestra la implementación de las clases mencionadas:

```
class Procesamiento
{
private:
    int thresh;
    int maxThresh;
    float area,MaxU, MinU,thresh1;
    cv::Mat frame, imgGray, imgBinary, zonaTrabajo,mask;

public:
    //CONSTRUCTOR Y DESTRUCTOR.
    Procesamiento();
    ~Procesamiento() {};

    //METODO PARA CARGAR LOTE DE IMAGENES.
    int cargarDatosTomates();

    //METODO DONDE SE EJECUTAN LOS TRACKBAR.
    void TreshCalib ();

    //METODO DONDE SE OBTIENE LA IMAGEN ANALIZADA FINAL.
    void getResultado() const;

    //METODOS DE USO DE TRACKBARS
    static void onTrackbarThreshCalib(int, void*);

    //METODOS DE OBTENCION DE AREA POR PASO DE CAMARA, O PASO DE IMAGEN FIJA.
    void calculoArea(FallaDeTomate &);

    //METODO DE ORDENACION
    void ordenaAreas(vector<float>&);

    //METODO PARA GUARDAR IMAGEN EN ARCHIVO JPG A PARTIR DE CAMARA.
    void guardaImg(string&);

    //METODO PARA OBTENER EL AREA DEL TOMATE.
    float getArea() const;

    //METODO PARA LIMPIAR OBJETO
    void limpiarProcesamiento();

};
```

Fig. 14: Clase Procesamiento.

```
class FallaDeTomate
{
private:
    time_t fecha;
    float areaTomate;
    vector<float> aberracion;

public:
    void setFecha(time_t&);
    void setAreaTomate(float &);
    void nuevaAberracion(float &); //agrega una nueva aberracion al tomate y retorna la cantidad del template aberraciones.
    unsigned int numFallas() const;
    float fallaGeneralTomate()const;
    void limparFalla();

    //SOBRECARGA DEL OPERADOR SORT QUE ORDENA DE MENOR A MAYOR POR FECHA, DESDE LA FECHA MAS RECIENTE HASTA LA MENOS RECIENTE.
    friend struct SortMenorMayorFecha;
    //SOBRECARGA DEL OPERADOR SORT QUE ORDENA DE MAYOR A MENOR POR FECHA, DESDE LA MENOS RECIENTE HASTA LA MAS RECIENTE.
    friend struct SortMayorMenorFecha;

};

//SOBRECARGA DE OPERADORES DE ENTRADA SALIDA
istream& operator>>(istream& is, FallaDeTomate &);
ostream& operator<< (ostream& os, FallaDeTomate& falla);
```

Fig. 15: Clase FallaDeTomate.

```
enum CriterioOrd{POR_ABERRACION_MAYOR_A_MENOR = 1,POR_ABERRACION_MENOR_A_MAYOR,POR_FECHA_MAYOR_A_MENOR,POR_FECHA_MENOR_A_MAYOR};

class ContenedorFallaDeTomate{
    vector<FallaDeTomate> contenedor;
public:
    void addFalla(FallaDeTomate &);
    void ordenar(CriterioOrd&);

    //SOBRECARGA DEL OPERADOR SORT QUE ORDENA DE MAYOR A MENOR POR TAMAÑO DE FALLA.
    friend struct SortMayorMenorFalla;
    //SOBRECARGA DEL OPERADOR SORT QUE ORDENA DE MENOR A MAYOR POR TAMAÑO DE FALLA.
    friend struct SortMenorMayorFalla;
    //SOBRECARGA DE OPERADOR DE SALIDA HACIA PANTALLA.
    friend void operator<<(ostream& os,ContenedorFallaDeTomate &);
    //SOBRECARGA DE OPERADOR DE SALIDA HACIA ARCHIVOS.
    friend ofstream& operator<<(ofstream &,ContenedorFallaDeTomate &);

};
```

Fig. 16: Clase ContenedorFallaTomate.

• **Polimorfismo:** En POO el concepto de polimorfismo permite escribir código con mayor naturalidad, llamando con el mismo nombre a diferentes métodos, resolviendo en tiempo de ejecución o compilación cuál es el método que realmente debe ser llamado en cada ocasión. Cuando la resolución del polimorfismo se realiza debido al tipo de argumentos del método, el compilador puede resolver en tiempo de compilación, como es en este caso.
Una de las formas que utiliza C++ para obtener el polimorfismo es mediante la sobrecarga de operadores. Esto nos permite definir operadores cuyos comportamientos varían de acuerdo a los parámetros que se les aplican. Así es posible, por ejemplo, agregar el operador ostream y ofstream, como los operadores istream e ifstream y hacer que se comporte de manera distinta cuando está haciendo referencia a un objeto.

• **Uso de Friend:** Se uso la declaración “friend” en algunas funciones que requerían el acceso a los elementos privados de la clase como por ejemplo, los métodos de ordenación por fecha en la clase FallaDeTomate, o los operadores de inserción en el flujo de salida para la clase ContenedorFallaTomate. Si bien esto no es conveniente, ya que viola el paradigma de encapsulamiento de c++, fue necesario para la implementación.

• **Uso de const:** En aquellos métodos que no modifican propiedades privadas de la clase, se los declaro como const. Esto es una buena practica de programación ya que los intentos de modificar el objeto son captados en tiempo de compilación en lugar de provocar errores en tiempo de ejecución.

• **Uso de Static:** Al definir una propiedad o método de clase como Static, significa que se declara esa propiedad o método no perteneciendo a la clase, pero si que para acceder a alguno de ellos es necesario resolver el ámbito con el nombre de la clase y el operador de resolución de ámbito ::. En el proyecto se uso Static para el uso de las variaciones de los trackbar al momento de calibrar algún parámetro, dentro la clase Procesamiento.

• **Herencia:** Si bien en un principio del proyecto se pensó en la posibilidad de utilizar herencia para detectar aberraciones en tomates de otra categoría (por ej perita, sanjuanino, etc), observamos que no era necesario por el hecho de que, al ser un software genérico, cambiando los parámetros de umbralización perfectamente se acoplaba a nuestras necesidades, por lo que se descartó su implementación.

• **Template o plantillas:** La idea de las plantillas o templates es la de generar código que pueda ser instanciado para distintos tipos de datos sin necesitar reescribirse. En este proyecto se vio la necesidad de utilizar contenedores de diferentes tipos de datos para el procesamiento de imágenes, por lo que fue útil la generación de los mismos para diferenciar contenedores de áreas, como de contornos, etc.

• **Contenedor STL: Clase vector:** La clase vector es un contenedor de la biblioteca estándar stl que a su vez es un template, lo que permite convertirlo en un contenedor de cualquier tipo de datos, en particular, de objetos del tipo de la clase o estructura que nosotros quisiéramos desarrollar. Este contenedor es de tipo dinámico, re-dimensiona la cantidad de memoria que utiliza de acuerdo al tamaño que tome, permitiendo extender su capacidad y agregarle nuevos elementos. También es posible pre dimensionarlo utilizando su constructor. La memoria que reserva es liberada al desaparecer el objeto asociado, que sigue las reglas que rigen a cualquier propiedad. En este proyecto nos fue de gran utilidad el uso de contenedores para almacenar información que luego iba a ser procesado y/o ordenada utilizando para ellos distintas funciones como pushback o sort.

## 4) MEJORAS AL PROYECTO.

• **Excepciones:** Las excepciones brindan un mecanismo que permite, frente a una situación que puede generar inestabilidad del ejecutable, evitarla disparando una excepción y contenerla según el criterio del programador, dado que quien programa un algoritmo puede saber cuándo una situación inestable se puede producir y la mejor manera de contenerla.

• **Memoria Compartida:** No se vio necesario implementar esto ya que todo se realizó fuera de línea y mediante la utilización de un solo ordenador. En el caso de realizarse en línea se podría implementar esto mediante el uso de memoria compartida entre una cámara y el microprocesador que recibe los datos.

• **Hilos o "Threads":** Se pensó como forma de aplicación, la división de la imagen en cuadrantes angulares, procesando cada uno en uno hilo de forma independiente mejorando el tiempo de procesamiento y la resolución dentro de cada uno.

• **Procesamiento en Línea:** Con la idea de progresar en el proyecto y profundizar temas de visión artificial para lograr un procesamiento más avanzado, se tiene pensado generar una base de datos de tomates, para de esa manera entrenar una cámara mediante inteligencia artificial por medio de feactures para que de esa manera extrapolemos este proyecto a una aplicación más sofisticada.

## 5) CONCLUSION.

Concluimos que la POO tiene una gran potencialidad a la hora de realizar la implementación de un algoritmo para una aplicación. Haciendo uso del encapsulamiento y de sus paradigmas, logramos obtener un código ordenado, lo cual es muy beneficioso si pretende ser reutilizado por cualquier otro usuario que quiera seguir investigando sobre el tema.

Con respecto a OpenCV es una librería que contiene muchas estructuras de datos complejos y funciones de alto nivel que pueden ser utilizadas para diversos procesos. Además, contiene una interfaz gráfica de usuario llamada Highgui que nos permite un fácil y rápido forma de interactuar y visualizar las imágenes. Esto nos ahorró gran cantidad de tiempo al momento de procesar las imágenes, detectando las aberraciones con cierta facilidad.

Por último, con el proyecto integrador se logró alcanzar satisfactoriamente los objetivos propuesto por ambas asignaturas a lo largo del semestre. Obtuvimos grandes herramientas de programación que serán de gran uso en nuestra carrera profesional.

# 6) REFERENCIAS.

[1] Apuntes de Cátedra, Complementos de Informática**.** -https://drive.google.com/drive/folders/0ByhW0CC2TKHyN2M4X1pqZzlhTUE.

[2] Apuntes de Cátedra, Sistemas para Control I https://drive.google.com/drive/folders/0ByhW0CC2TKHyOTByV2x4NlZLbDQ.

[3] https://docs.opencv.org/3.4.1/

[4] Learning OpenCV 3 COMPUTER VISION IN C++ WITH THE OPENCV LIBRARY.