OOP Final Project Report

Name: Doğukan Avcı

Student Number: 151220202051

Date: 22.12.2024

For this semester's OOP course final project, we developed a project that uses many OOP concepts and OpenCV library to perform line and corner detection.

We used 4 classes to develop our project: **CommonProcesses**, **Detection**, **LineDetection** and **CornerDetection**. We also derived the Detection class from the CommonProcesses class using the **inheritance** operation. We **derived** the LineDetection and CornerDetection classes from the Detection class.

As a result, the **CommonProcesses** class is the base class, the Detection class is both a derived and a base class, and the **LineDetection** and **CornerDetection** classes are derived classes.

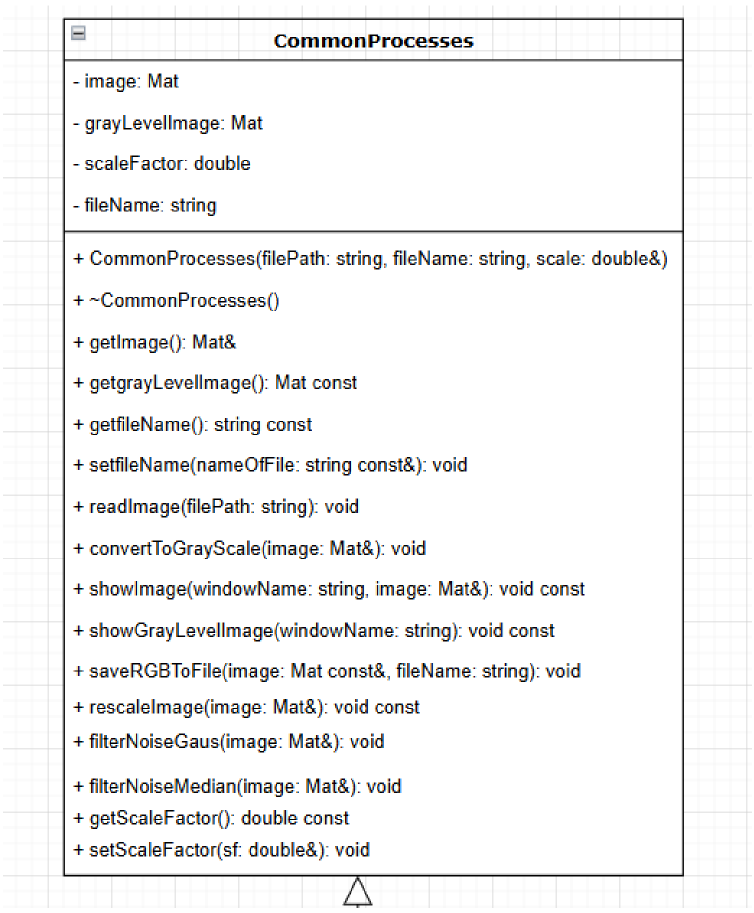

CommonProcesses
- image: Mat
- grayLevelImage: Mat
- scaleFactor: double
- fileName: string
+ CommonProcesses(filePath: string, fileName: string, scale: double&)
+ ~CommonProcesses()
+ getImage(): Mat&
+ getgrayLevelImage(): Mat const
+ getfileName(): string const
+ setfileName(nameOfFile: string const&): void
+ readImage(filePath: string): void
+ convertToGrayScale(image: Mat&): void
+ showImage(windowName: string, image: Mat&): void const
+ showGrayLevelImage(windowName: string): void const
+ saveRGBToFile(image: Mat const&, fileName: string): void
+ rescaleImage(image: Mat&): void const
+ filterNoiseGaus(image: Mat&): void
+ filterNoiseMedian(image: Mat&): void
+ getScaleFactor(): double const
+ setScaleFactor(sf: double&): void

Figure 1 CommonProcesses UML

Let's start introducing our project with the **CommonProcess** class. In the **CommonProcess** class, we implement functions such as image opening, reading, and filtering, which are frequently used in the field of image processing, and by inheriting these functions, we prevent these operations from being easily used in derived classes and from being rewritten.

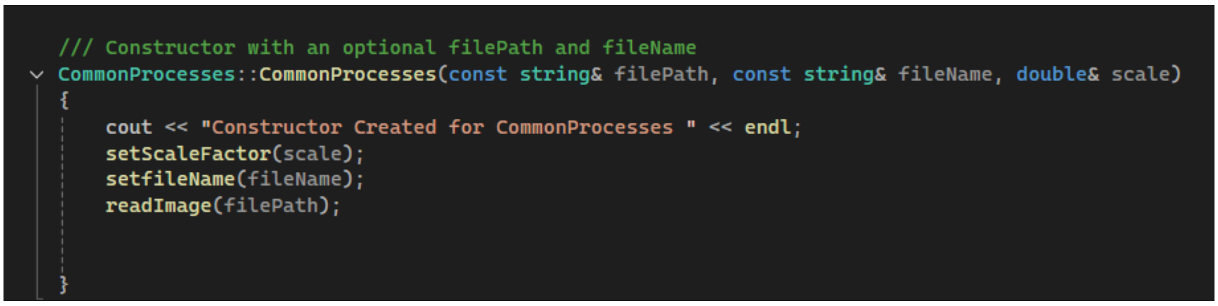
The CommonProcess class contains 4 private data members. These are: **image** in **Mat** type, **grayLevelImage** in **Mat** type, **scaleFactor** in **double** type, **fileName** in string **type**.

string and double types are known types that we have worked with before, but I worked with **Mat** type for the first time in the final project. Mat is one of the basic data structures of **OpenCV** library and is used to represent images. In **OpenCV**, an image contains a matrix of pixel values. For Grayscale images, that is, for black and white images (which consist only of gray tones), Mat contains the Gray brightness value of each pixel. We have also covered this topic in detail in the Image Processing lesson. For color images, Matte contains three channels, these channels are called Blue, Green and Red **BGR**. This matrix is used for operations such as reading, processing and saving images.

Data Members

- The **image(Mat)** variable holds the main image and represents the image read or processed from the File.
- The **grayLevelImage(Mat)** variable allows us to keep a color image in this variable after converting it to grayscale.
- The **scaleFactor(double)** variable is the scale factor used when resizing dimensions. It is taken from the user.
- The **filename**(string) variable holds the path to the image files. With this variable, you have options when we encode our image.

Member Functions



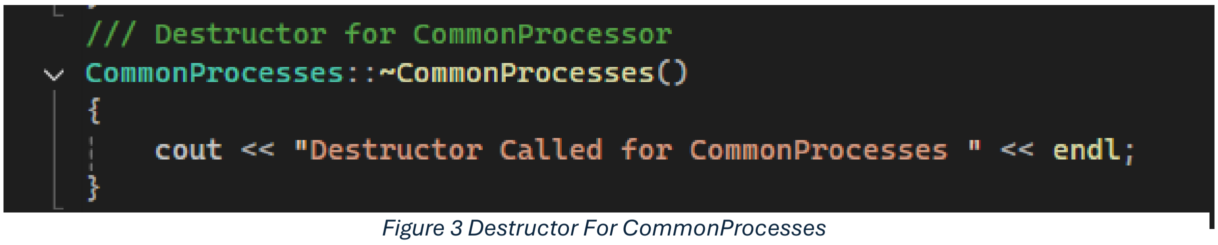
```
/// Constructor with an optional filePath and fileName
CommonProcesses::CommonProcesses(const string& filePath, const string& fileName, double& scale)
{
    cout << "Constructor Created for CommonProcesses " << endl;
    setScaleFactor(scale);
    setfileName(fileName);
    readImage(filePath);


}
```

Figure 2 Constructor For CommonProcesses

Constructor for CommonProcesses

- The **CommonProcesses constructor** takes **three** parameters as explained above: **filePath**, **fileName** and scale. Then there is a console message in the constructor that the constructor has been created. Then we set the **scaleFactor** and **fileName** variables using the set functions. **We did not use the initializer method** because they have their own **set** member functions. And finally, we read our image with the **readImage** member function.

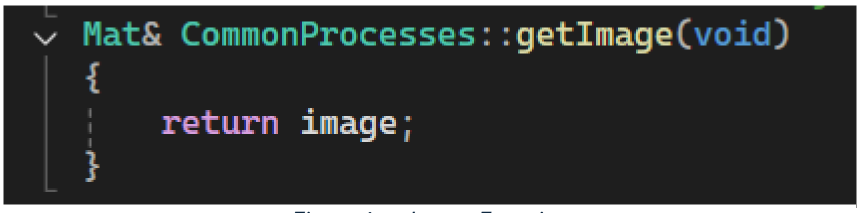
Destructor For CommonProcesses



```cpp
/// Destructor for CommonProcessor
CommonProcesses::~CommonProcesses()
{
    cout << "Destructor Called for CommonProcesses " << endl;
}
```

Figure 3 Destructor For CommonProcesses

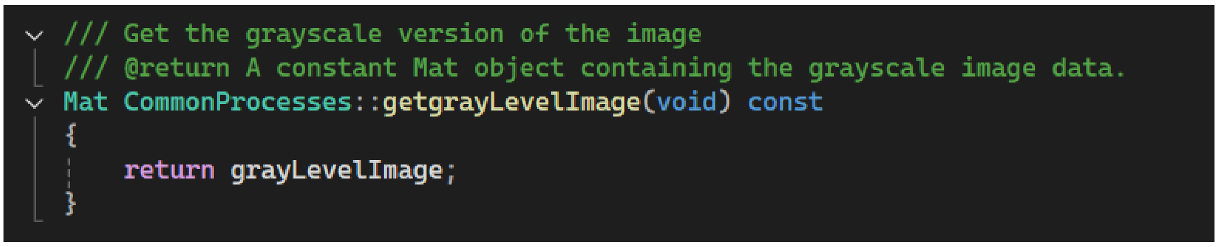
- ➢ We created the **Destructor** empty and it only contains a message to write to the console when called.



```cpp
Mat& CommonProcesses::getImage(void)
{
    return image;
}
```

Figure 4 getImage Function

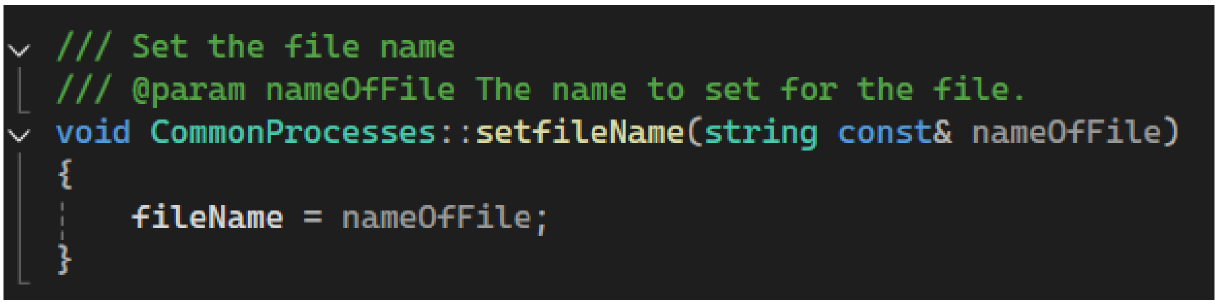
- ➢ Since **image** is a private data member, we added a get function to call it in other derived classes and strengthen the encapsulation structure. Since the image file is larger than normal variables, we returned it by reference and saved memory. Since the image variable is of **Mat** type, we returned it as **Mat** type.



```cpp
/// Get the grayscale version of the image
/// @return A constant Mat object containing the grayscale image data.
Mat CommonProcesses::getgrayLevelImage(void) const
{
    return grayLevelImage;
}
```

Figure 5 getgrayLevelImage

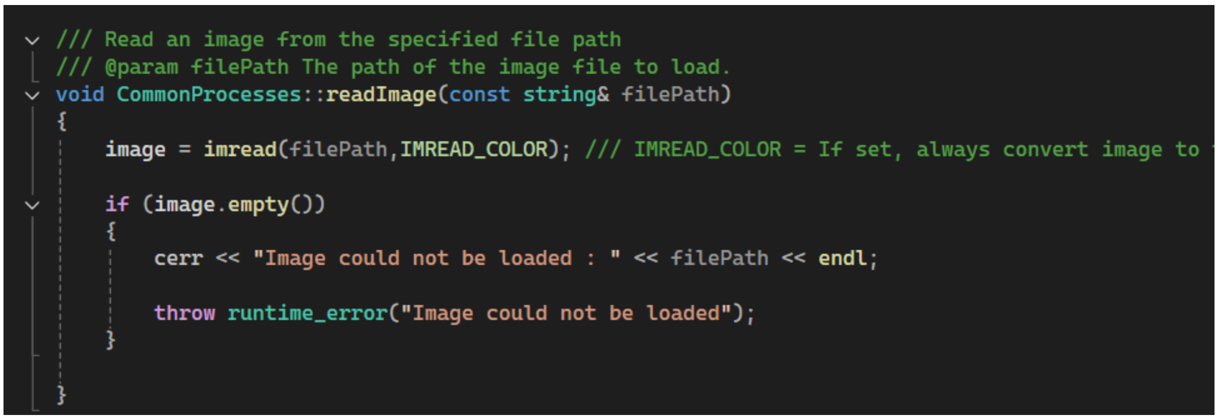
- ➢ Similarly, a get member function was created for **grayLevelImage**.



```cpp
/// Set the file name
/// @param nameOfFile The name to set for the file.
void CommonProcesses::setfileName(string const& nameOfFile)
{
    fileName = nameOfFile;
}
```

Figure 6 setFileName

➢ The set function was created to assign the **fileName** variable and encapsulation was strengthened.

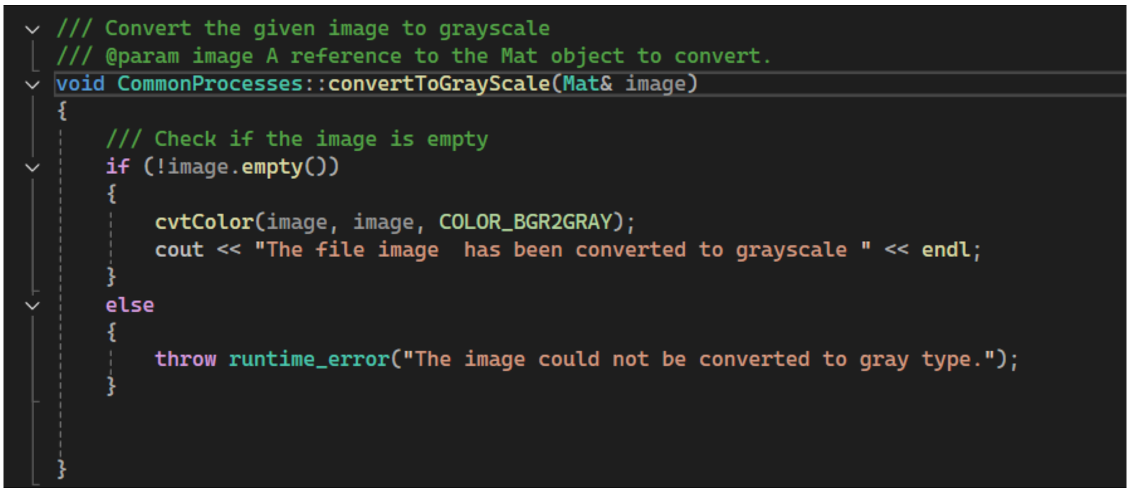


```cpp
/// Read an image from the specified file path
/// @param filePath The path of the image file to load.
void CommonProcesses::readImage(const string& filePath)
{
    image = imread(filePath,IMREAD_COLOR); /// IMREAD_COLOR = If set, always convert image to

    if (image.empty())
    {
        cerr << "Image could not be loaded : " << filePath << endl;

        throw runtime_error("Image could not be loaded");
    }

}
```

Figure 7 readImage

➢ With the help of imread function in OpenCV library, we read the image in our file path and save it to our **2D** array defined with Mat variable. During image file assignment, we check if the file is empty. If the file is empty, we send **runtime_error**.

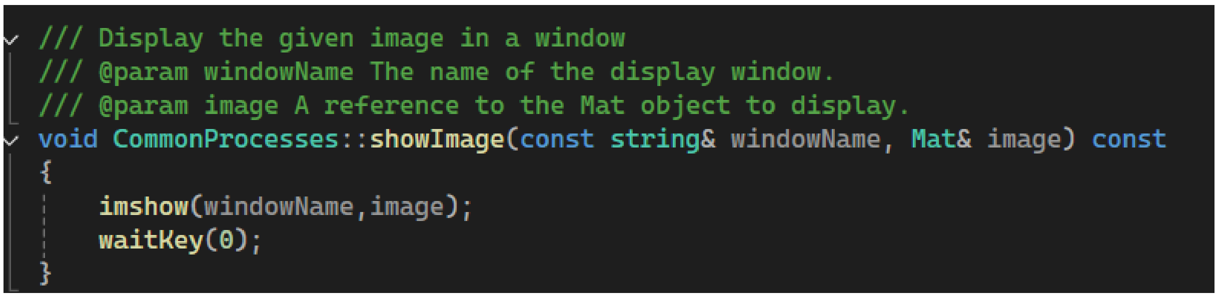
➢ Here **IMREAD_COLOR** indicates that the file's color, i.e. **RGB** values, should be read. We can make different readings with other options, for example, we can read the image directly with the grayscale value.



```cpp
/// Convert the given image to grayscale
/// @param image A reference to the Mat object to convert.
void CommonProcesses::convertToGrayScale(Mat& image)
{
    /// Check if the image is empty
    if (!image.empty())
    {
        cvtColor(image, image, COLOR_BGR2GRAY);
        cout << "The file image  has been converted to grayscale " << endl;
    }
    else
    {
        throw runtime_error("The image could not be converted to gray type.");
    }


}
```

Figure 8 convertToGrayScale

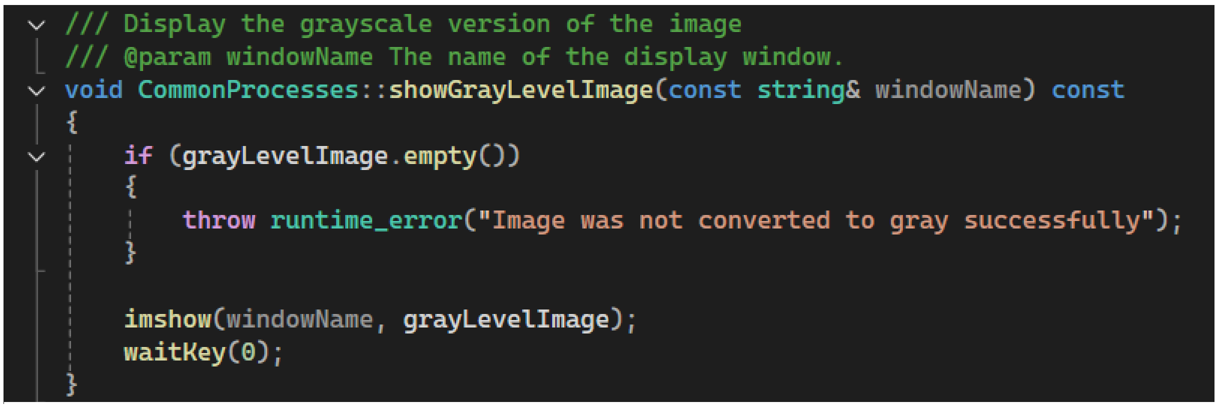
➢ We convert the **Mat& Image** file that we sent as a reference with the convertToGrayScale function to grayscale value with the **cvtColor** function in OpenCV and write it on the image again. Here, we check again whether the image variable is empty and send **runtime_error** according to the result. Converting the image to gray level is important for algorithms such as the Canny algorithm in the later stages.



```cpp
/// Display the given image in a window
/// @param windowName The name of the display window.
/// @param image A reference to the Mat object to display.
void CommonProcesses::showImage(const string& windowName, Mat& image) const
{
    imshow(windowName,image);
    waitKey(0);
}
```

Figure 9 showImage

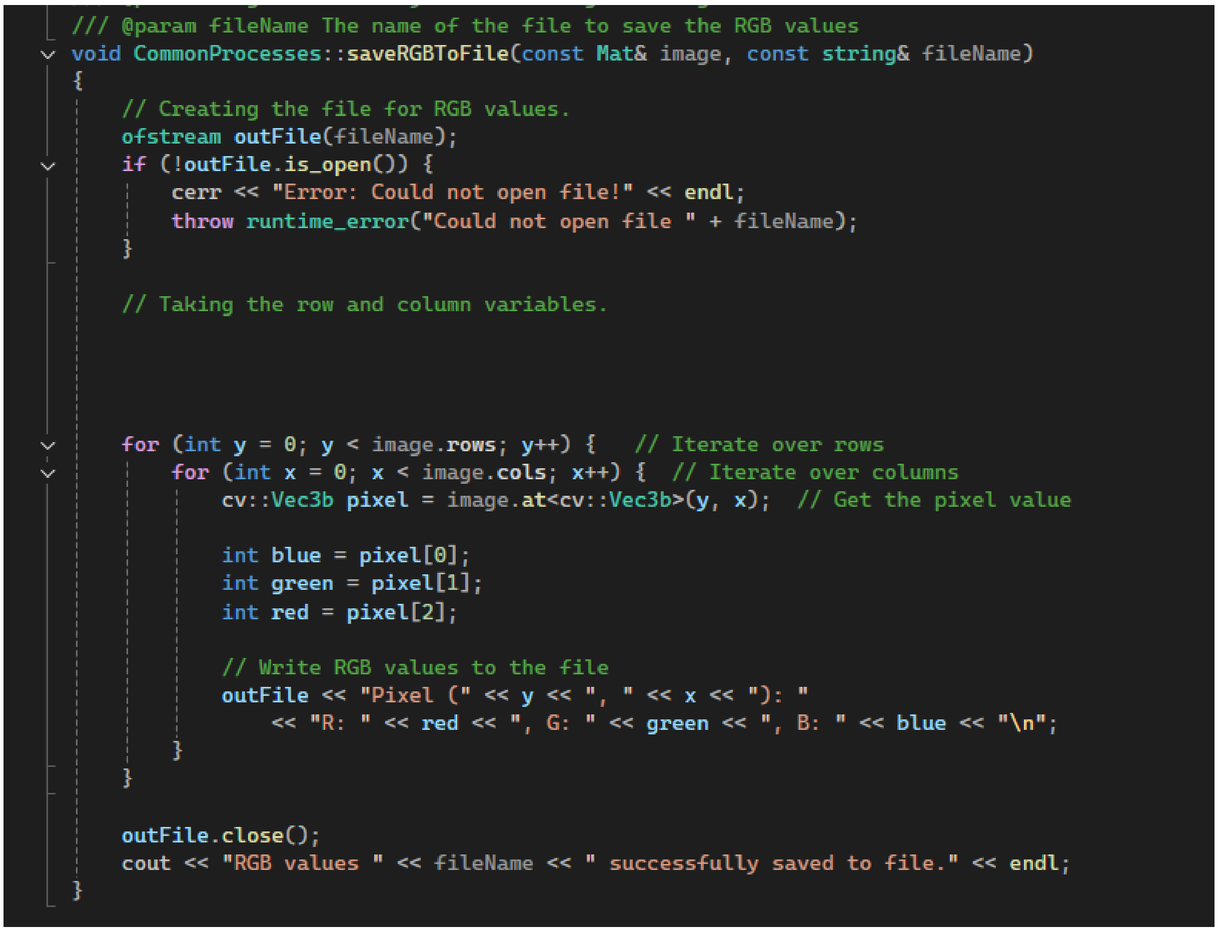
- ➢ The **showImage** member function takes the Image variable as a reference and the window name variable named **windowName** as a reference. With the help of the **imshow** function in the **OpenCV** library, it serves to print the image entered in the specified window name to the screen as output. Later, in order for the images to remain on the screen for a long time, the user is waited to press a key with the **waitKey(0**) in **OpenCV**.



```cpp
/// Display the grayscale version of the image
/// @param windowName The name of the display window.
void CommonProcesses::showGrayLevelImage(const string& windowName) const
{
    if (grayLevelImage.empty())
    {
        throw runtime_error("Image was not converted to gray successfully");
    }

    imshow(windowName, grayLevelImage);
    waitKey(0);
}
```

Figure 10 showGrayLevelImage

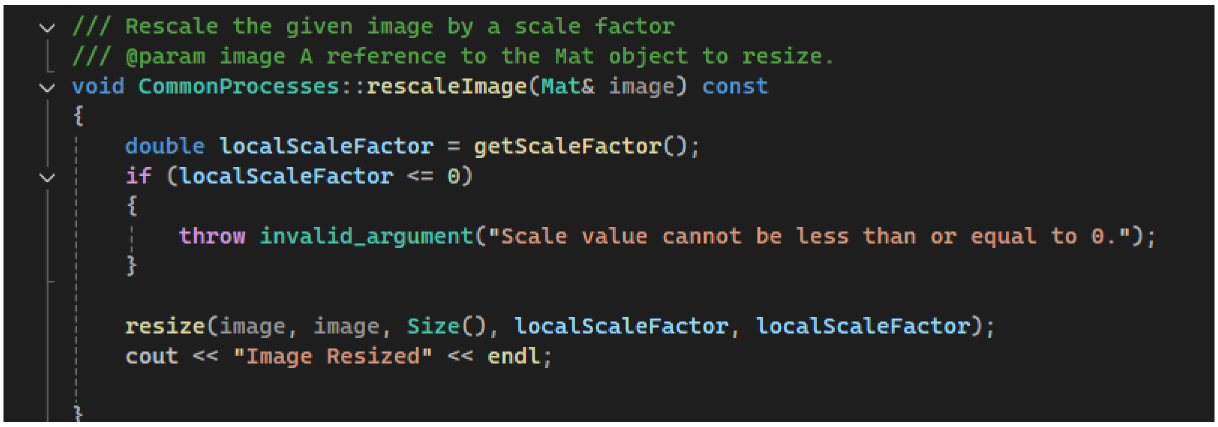
- ➢ To print gray level image screen, it prints gray level image with only window name without any input image value. Error checking is done before printing.



```cpp
/// @param fileName The name of the file to save the RGB values
void CommonProcesses::saveRGBToFile(const Mat& image, const string& fileName)
{
    // Creating the file for RGB values.
    ofstream outFile(fileName);
    if (!outFile.is_open()) {
        cerr << "Error: Could not open file!" << endl;
        throw runtime_error("Could not open file " + fileName);
    }

    // Taking the row and column variables.



    for (int y = 0; y < image.rows; y++) {   // Iterate over rows
        for (int x = 0; x < image.cols; x++) {  // Iterate over columns
            cv::Vec3b pixel = image.at<cv::Vec3b>(y, x);  // Get the pixel value

            int blue = pixel[0];
            int green = pixel[1];
            int red = pixel[2];

            // Write RGB values to the file
            outFile << "Pixel (" << y << ", " << x << "): "
                << "R: " << red << ", G: " << green << ", B: " << blue << "\n";
        }
    }

    outFile.close();
    cout << "RGB values " << fileName << " successfully saved to file." << endl;
}
```

Figure 11 saveRGBToFile

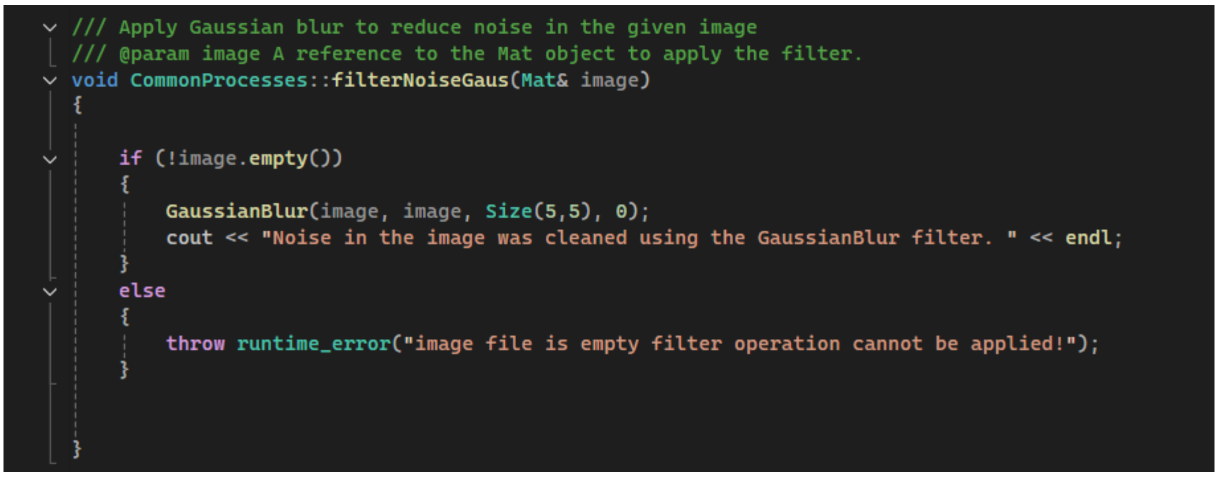
- With the help of the **fstream** library, we write the **RGB** values of the pixels to the specified file. This function takes the image whose **RGB** values will be written as input. We create an output file with **ofstream outFile**. Then we check whether it has been created or not. Then, in nested for loops, we assign these values to the blue, green and red variables. With the help of **outfile<<**, we write the values to the file and finally close our file with **.close()**. Not closing the file may cause certain errors.



```cpp
/// Rescale the given image by a scale factor
/// @param image A reference to the Mat object to resize.
void CommonProcesses::rescaleImage(Mat& image) const
{
    double localScaleFactor = getScaleFactor();
    if (localScaleFactor <= 0)
    {
        throw invalid_argument("Scale value cannot be less than or equal to 0.");
    }

    resize(image, image, Size(), localScaleFactor, localScaleFactor);
    cout << "Image Resized" << endl;

}
```

Figure 12 rescaleImage

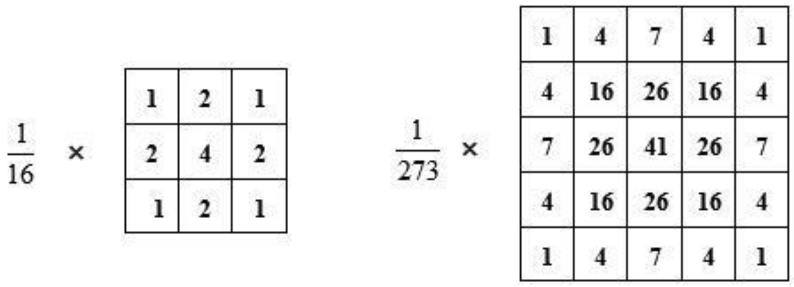
- The **rescaleImage** function takes the image variable as an input value by reference. The reason for this constant change is that the image variability is very large and we do not need to save it constantly. Then, we change the state of the image with the **resize()** function in **OpenCV** and write it back to the image. Before this, we check whether the scale factor is an invalid argument, i.e. a positive value. We throw an error for non-positive indicators.



```
/// Apply Gaussian blur to reduce noise in the given image
/// @param image A reference to the Mat object to apply the filter.
void CommonProcesses::filterNoiseGaus(Mat& image)
{

    if (!image.empty())
    {
        GaussianBlur(image, image, Size(5,5), 0);
        cout << "Noise in the image was cleaned using the GaussianBlur filter. " << endl;
    }
    else
    {
        throw runtime_error("image file is empty filter operation cannot be applied!");
    }


}
```

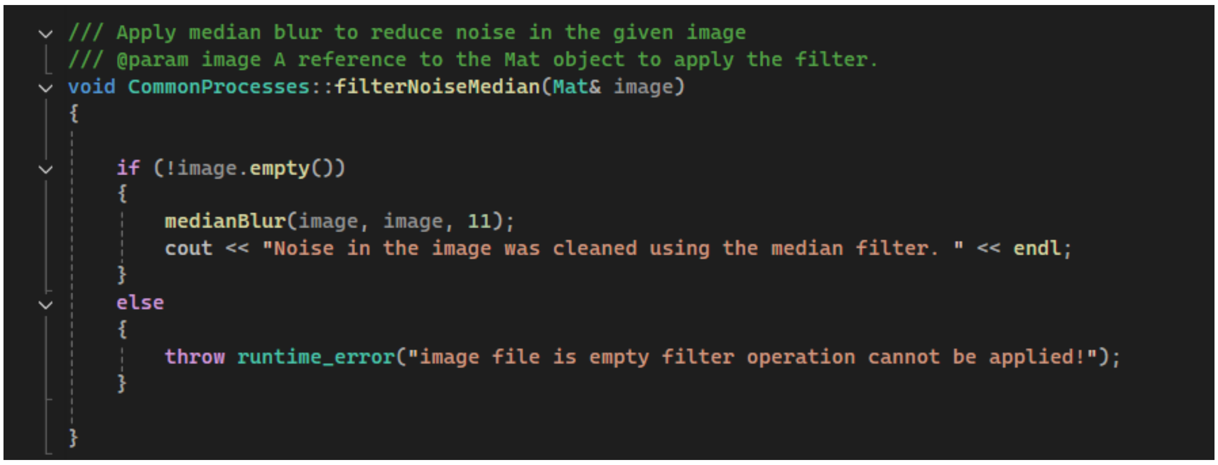
Figure 13 filterNoiseGaus

- In our project, two filter options have been added to add extra functionality. The first filter is the **Gaussian filter**. We can perform this operation thanks to the **GaussianBlur** function in **OpenCV**. First, we check whether our image file is empty or not, and then if the image file is full, we apply the Gaussian operation. The first input in the function is the input, the second input is the output value, and the third value is the size of the Gaussian filter. The Gaussian filter can only have a single-digit number of edges, for example, 3 to 3, 5 to 5, 7 to 7, and the last parameter is the standard deviation, according to which the smoothing rate of the image increases or decreases. The higher the standard deviation value, the higher the smoothing rate of the image.



$$\frac{1}{16} \times \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \qquad \frac{1}{273} \times \begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 7 & 4 & 1 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 7 & 26 & 41 & 26 & 7 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 1 & 4 & 7 & 4 & 1 \\ \hline \end{array}$$

Figure 14 Example of Gausian Filtering



```
/// Apply median blur to reduce noise in the given image
/// @param image A reference to the Mat object to apply the filter.
void CommonProcesses::filterNoiseMedian(Mat& image)
{

    if (!image.empty())
    {
        medianBlur(image, image, 11);
        cout << "Noise in the image was cleaned using the median filter. " << endl;
    }
    else
    {
        throw runtime_error("image file is empty filter operation cannot be applied!");
    }

}
```

Figure 15 Median Filter

- Another filter used in the project is the **median filter**. This filter, like the Gauss filter, allows the removal of high-frequency pixels on the image. Similarly, the image variable is checked to see if it is empty or not, and we can apply this function thanks to the **medianBlur** function in **OpenCV**. The **medianBlur** function takes 3 different parameters: input, output and filter size. The larger the filter size, the softer the image.

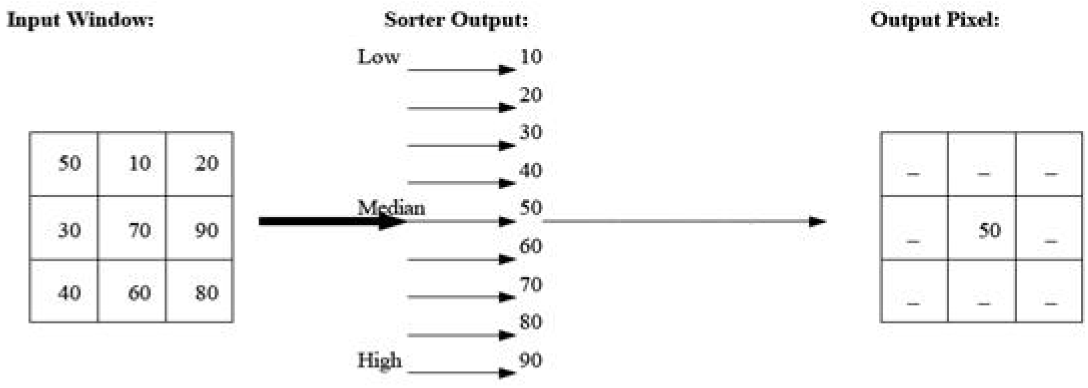


Figure 16 Example of Median Filtering

The images below were produced using the codes in the project:

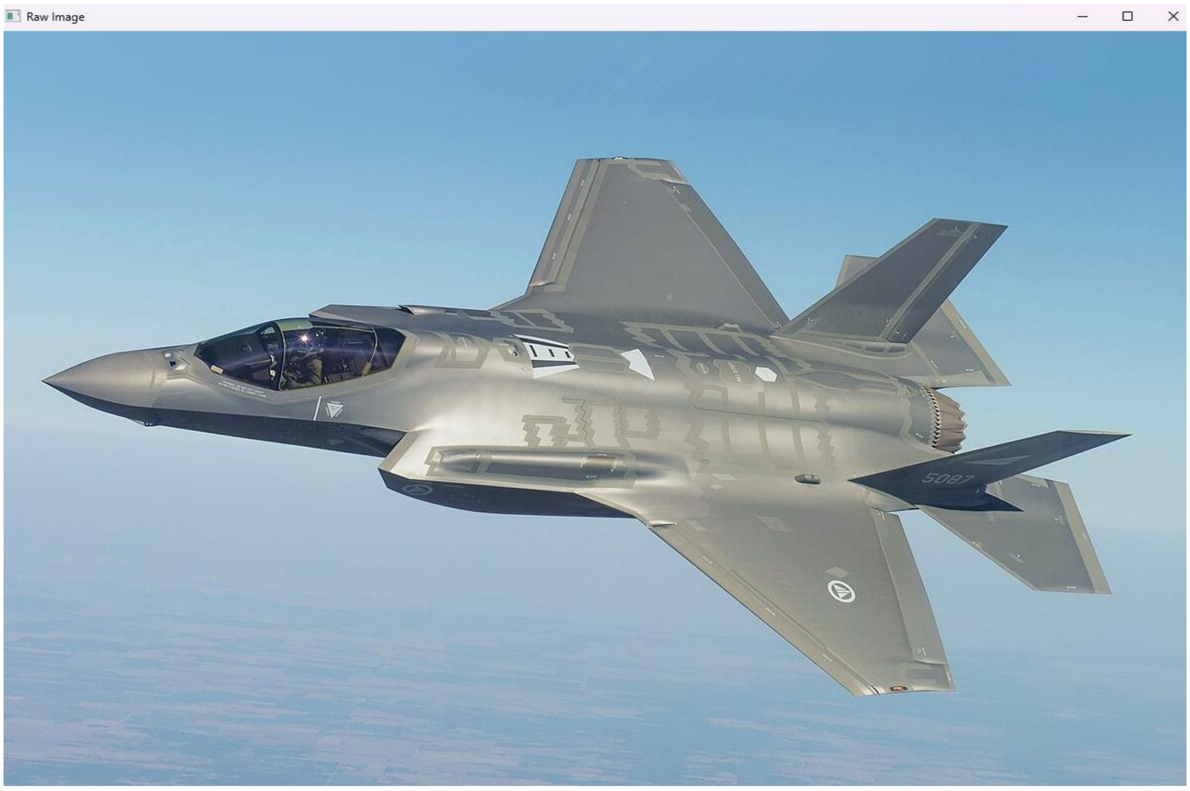


Figure 17 Raw Image

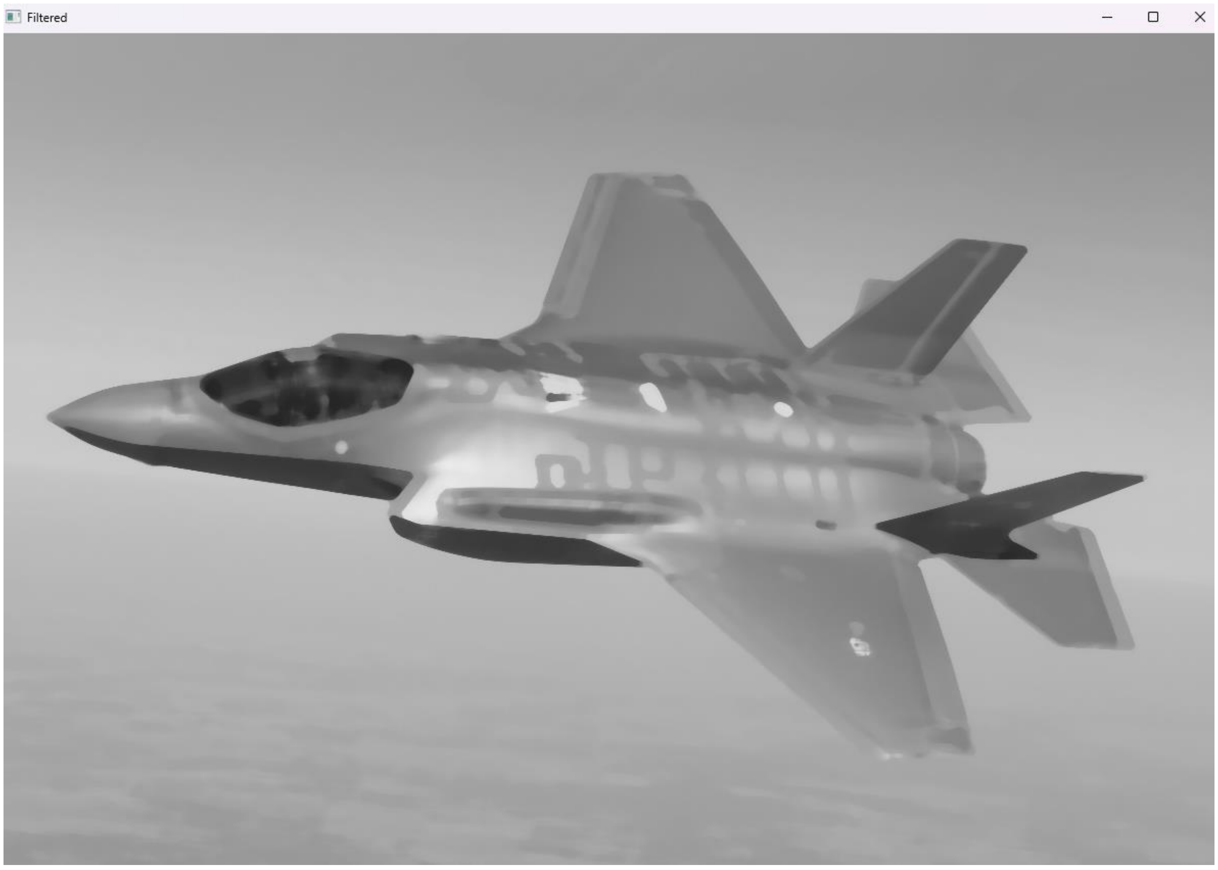


Figure 18 Gaussian 3x3

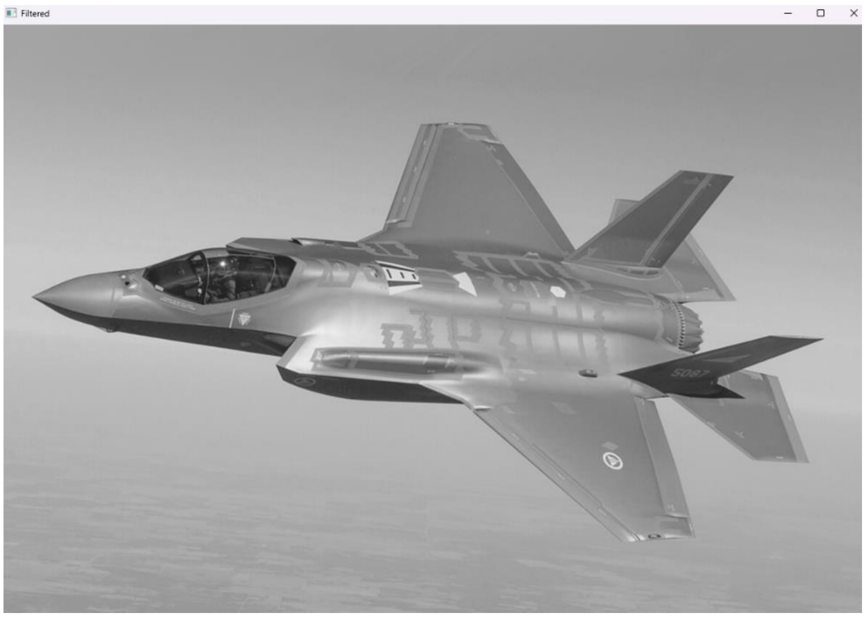
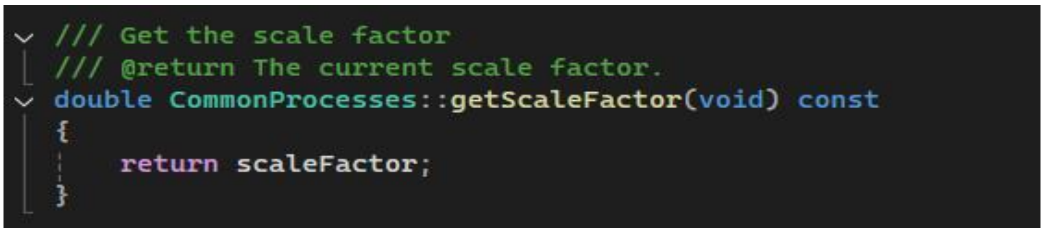


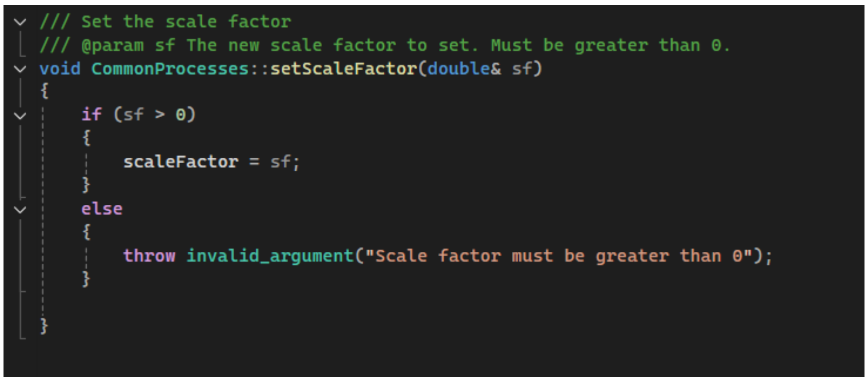
Figure 19 Median 3x3



```
/// Get the scale factor
/// @return The current scale factor.
double CommonProcesses::getScaleFactor(void) const
{
    return scaleFactor;
}
```

Figure 20 getScaleFactor

➢ We used this to access the **scalefactor** variable from derived classes and increase **encapsulation**.

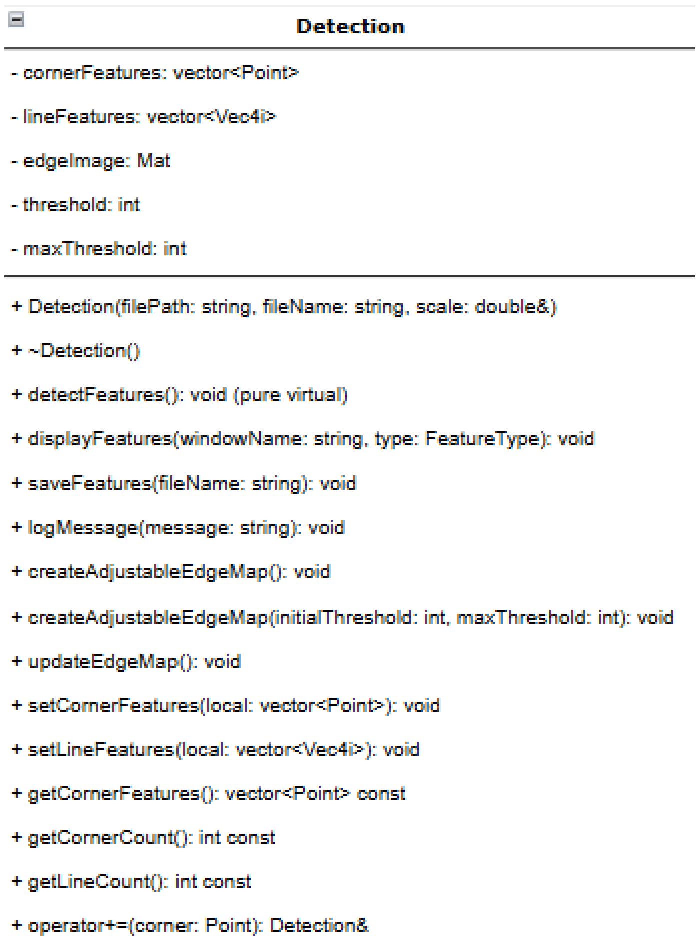


```
/// Set the scale factor
/// @param sf The new scale factor to set. Must be greater than 0.
void CommonProcesses::setScaleFactor(double& sf)
{
    if (sf > 0)
    {
        scaleFactor = sf;
    }
    else
    {
        throw invalid_argument("Scale factor must be greater than 0");
    }

}
```

Figure 21 setScaleFactor

➢ We also added the set member function to assign the **scaleFactor** structure and to check for invalid arguments.

Detection Class



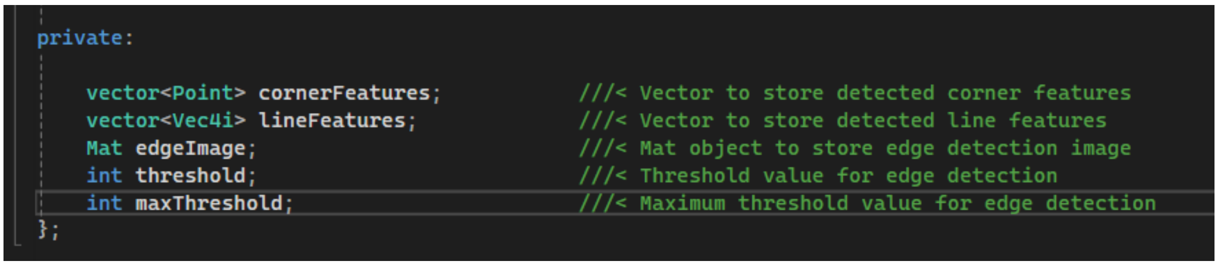
Detection

- cornerFeatures: vector<Point>
- lineFeatures: vector<Vec4i>
- edgeImage: Mat
- threshold: int
- maxThreshold: int

+ Detection(filePath: string, fileName: string, scale: double&)
+ ~Detection()
+ detectFeatures(): void (pure virtual)
+ displayFeatures(windowName: string, type: FeatureType): void
+ saveFeatures(fileName: string): void
+ logMessage(message: string): void
+ createAdjustableEdgeMap(): void
+ createAdjustableEdgeMap(initialThreshold: int, maxThreshold: int): void
+ updateEdgeMap(): void
+ setCornerFeatures(local: vector<Point>): void
+ setLineFeatures(local: vector<Vec4i>): void
+ getCornerFeatures(): vector<Point> const
+ getCornerCount(): int const
+ getLineCount(): int const
+ operator+=(corner: Point): Detection&

Figure 22 DetectionClassUML

Data Members



```
private:

    vector<Point> cornerFeatures;          ///< Vector to store detected corner features
    vector<Vec4i> lineFeatures;            ///< Vector to store detected line features
    Mat edgeImage;                         ///< Mat object to store edge detection image
    int threshold;                         ///< Threshold value for edge detection
    int maxThreshold;                      ///< Maximum threshold value for edge detection
};
```

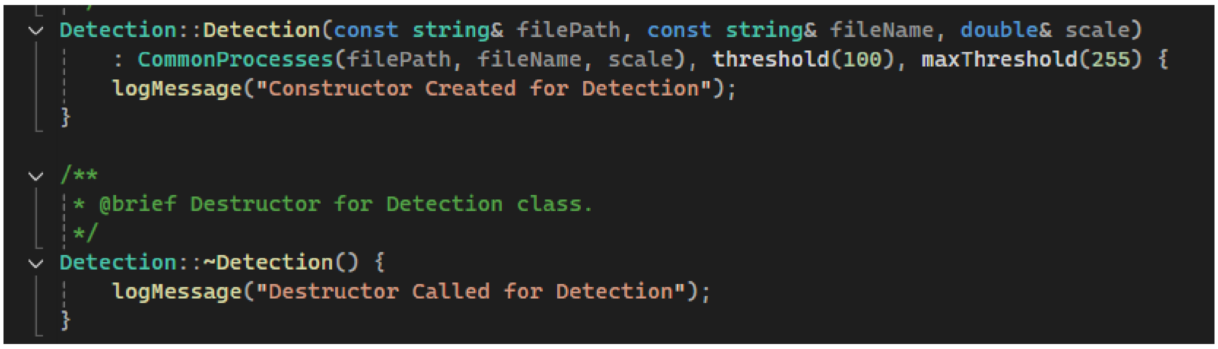
Figure 23 Data members of Detection Class

➢ **CornerFeatures** (vector<Point>) is a vector that stores the corner points detected in the image. Each point is stored as a Point object. This variable is updated with **setCornerFeatures()** after the corner settings.

- The **lineFeatures**(vector<Vec4i>) variable stores the lines detected in the image. Each line is stored as a Vec4i representing the start and end points. After line detection, it is provided with the setLineFeatures() method.
- **edgeImage** (Mat) Stores the image created as a result of edge detection. It is updated with the updateEdgeMap() function and created with the Canny edge detection algorithm.
- **threshold** (int) minimum threshold value used for edge detection. The user can change this value via an interactive trackbar.
- **maxThreshold** (int) The maximum threshold that can be used for edge detection. Creates a boundary for the trackbar.

Vec4i is also a data type that we encountered for the first time, just like Mat. Vec4i is one of the basic data types provided by **OpenCV**. This data type means a vector of four integers. Its full name can be explained as "**Vector of 4 Integers**". It is used to specify lines in algorithms such as Hough transform.

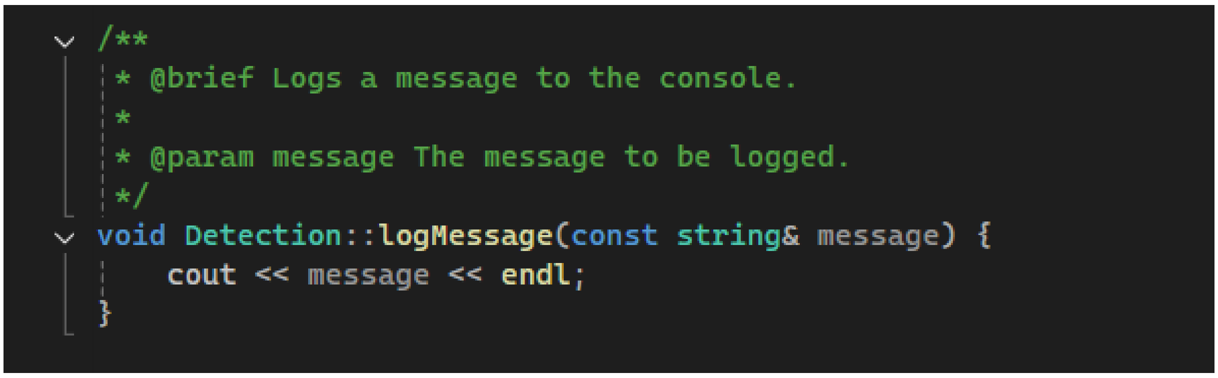
Member Functions



```
Detection::Detection(const string& filePath, const string& fileName, double& scale)
    : CommonProcesses(filePath, fileName, scale), threshold(100), maxThreshold(255) {
    logMessage("Constructor Created for Detection");
}

/**
 * @brief Destructor for Detection class.
 */
Detection::~Detection() {
    logMessage("Destructor Called for Detection");
}
```

Figure 24 Constructor and Destructor for Detection

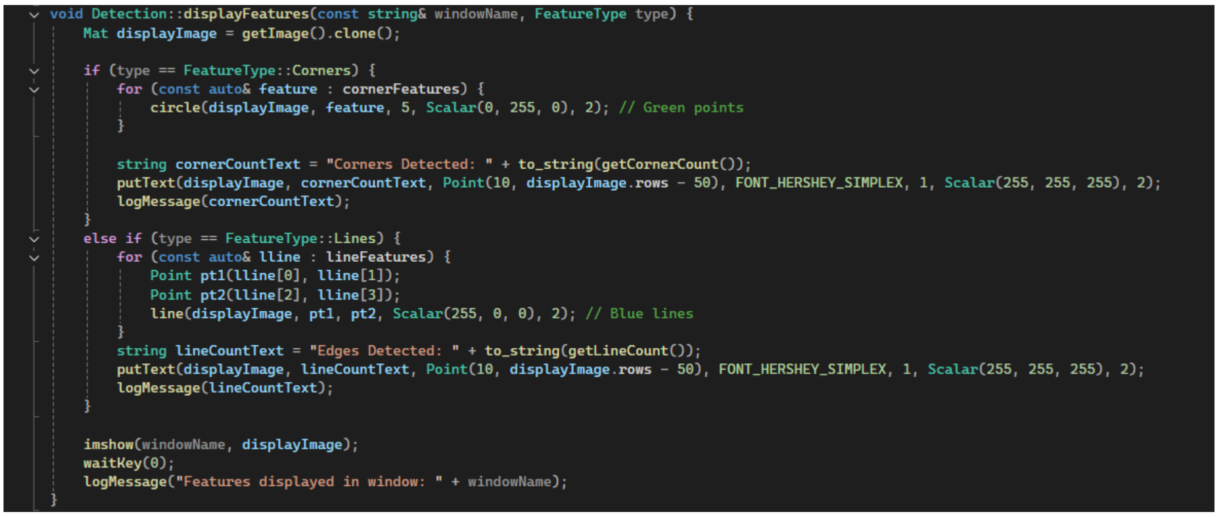
- In Detection Constructor, constructor was created with the help of **logmessage** function written in the project and threshold and **maxThreshold** values were assigned with initializer method. Similarly, since Detection is a derived class, **CommonProcesses** Class also needs to be defined. We define **CommonProcesses** class with **FilePath**, **fileName** and scale values. Then we add **Destructo**r and we can check whether it is called with **logMessage** function or not from console screen.



```
/**
 * @brief Logs a message to the console.
 *
 * @param message The message to be logged.
 */
void Detection::logMessage(const string& message) {
    cout << message << endl;
}
```

Figure 25 logMessage member function

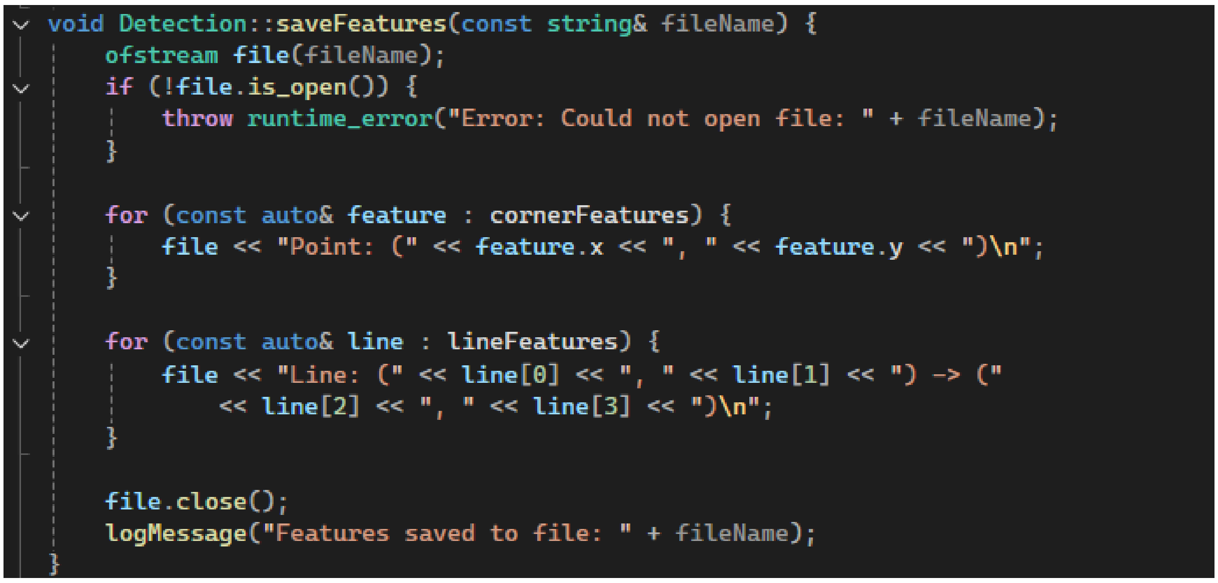
- The **logMessage** function takes the message variable with its address in string type and prints it with cout. It was added to the code for ease of reading and easier application.



```cpp
void Detection::displayFeatures(const string& windowName, FeatureType type) {
    Mat displayImage = getImage().clone();

    if (type == FeatureType::Corners) {
        for (const auto& feature : cornerFeatures) {
            circle(displayImage, feature, 5, Scalar(0, 255, 0), 2); // Green points
        }

        string cornerCountText = "Corners Detected: " + to_string(getCornerCount());
        putText(displayImage, cornerCountText, Point(10, displayImage.rows - 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255), 2);
        logMessage(cornerCountText);
    }
    else if (type == FeatureType::Lines) {
        for (const auto& lline : lineFeatures) {
            Point pt1(lline[0], lline[1]);
            Point pt2(lline[2], lline[3]);
            line(displayImage, pt1, pt2, Scalar(255, 0, 0), 2); // Blue lines
        }
        string lineCountText = "Edges Detected: " + to_string(getLineCount());
        putText(displayImage, lineCountText, Point(10, displayImage.rows - 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255), 2);
        logMessage(lineCountText);
    }

    imshow(windowName, displayImage);
    waitKey(0);
    logMessage("Features displayed in window: " + windowName);
}
```

Figure 26 Display Features Function

- The **disPlayFeatures** function allows any features (corner, edge) to be displayed with certain marks when encountered. For example, this function takes the **FeatureType** property as input. It takes into account the conditions within the feature type property and if it is a corner, it marks all the feature values within **cornerFeatures** with the circle function in **OpenCV**. The first input shows which image to use, the other input shows the features, the others express the circle diameter, color and thickness respectively. The same operations are done for the line. Each line feature is drawn with the line function in **OpenCV**.

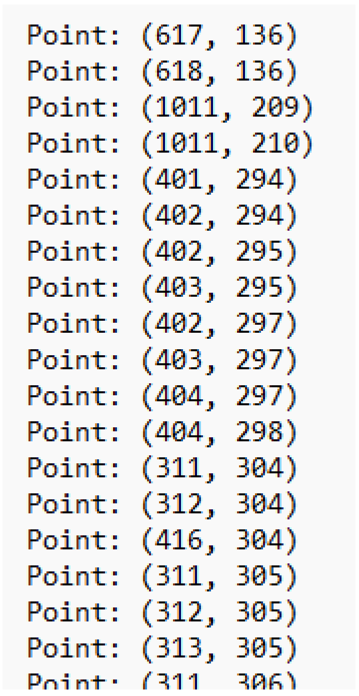


```cpp
void Detection::saveFeatures(const string& fileName) {
    ofstream file(fileName);
    if (!file.is_open()) {
        throw runtime_error("Error: Could not open file: " + fileName);
    }

    for (const auto& feature : cornerFeatures) {
        file << "Point: (" << feature.x << ", " << feature.y << ")\n";
    }

    for (const auto& line : lineFeatures) {
        file << "Line: (" << line[0] << ", " << line[1] << ") -> ("
            << line[2] << ", " << line[3] << ")\n";
    }

    file.close();
    logMessage("Features saved to file: " + fileName);
}
```

Figure 27 saveFeatures

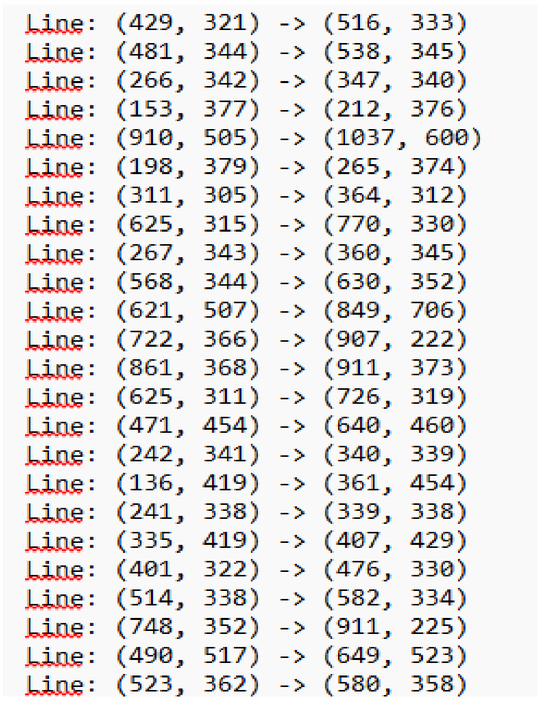
- Thanks to the **saveFeatures** function, we use the **fstream** library that we have explained before and save the found corner and line feature values to our file. Then, we close the file again and print the log message. This function also takes the **fileName** with its address as the input value.

FILE OUTPUT :



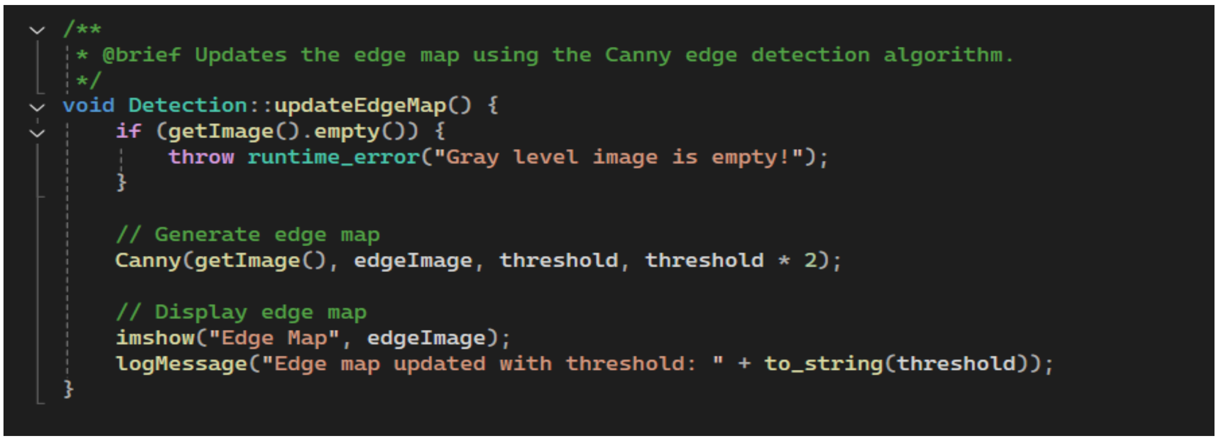
```
Point: (617, 136)
Point: (618, 136)
Point: (1011, 209)
Point: (1011, 210)
Point: (401, 294)
Point: (402, 294)
Point: (402, 295)
Point: (403, 295)
Point: (402, 297)
Point: (403, 297)
Point: (404, 297)
Point: (404, 298)
Point: (311, 304)
Point: (312, 304)
Point: (416, 304)
Point: (311, 305)
Point: (312, 305)
Point: (313, 305)
Point: (311, 306)
```

Figure 28 Saved Point Values



```
Line: (429, 321) -> (516, 333)
Line: (481, 344) -> (538, 345)
Line: (266, 342) -> (347, 340)
Line: (153, 377) -> (212, 376)
Line: (910, 505) -> (1037, 600)
Line: (198, 379) -> (265, 374)
Line: (311, 305) -> (364, 312)
Line: (625, 315) -> (770, 330)
Line: (267, 343) -> (360, 345)
Line: (568, 344) -> (630, 352)
Line: (621, 507) -> (849, 706)
Line: (722, 366) -> (907, 222)
Line: (861, 368) -> (911, 373)
Line: (625, 311) -> (726, 319)
Line: (471, 454) -> (640, 460)
Line: (242, 341) -> (340, 339)
Line: (136, 419) -> (361, 454)
Line: (241, 338) -> (339, 338)
Line: (335, 419) -> (407, 429)
Line: (401, 322) -> (476, 330)
Line: (514, 338) -> (582, 334)
Line: (748, 352) -> (911, 225)
Line: (490, 517) -> (649, 523)
Line: (523, 362) -> (580, 358)
```

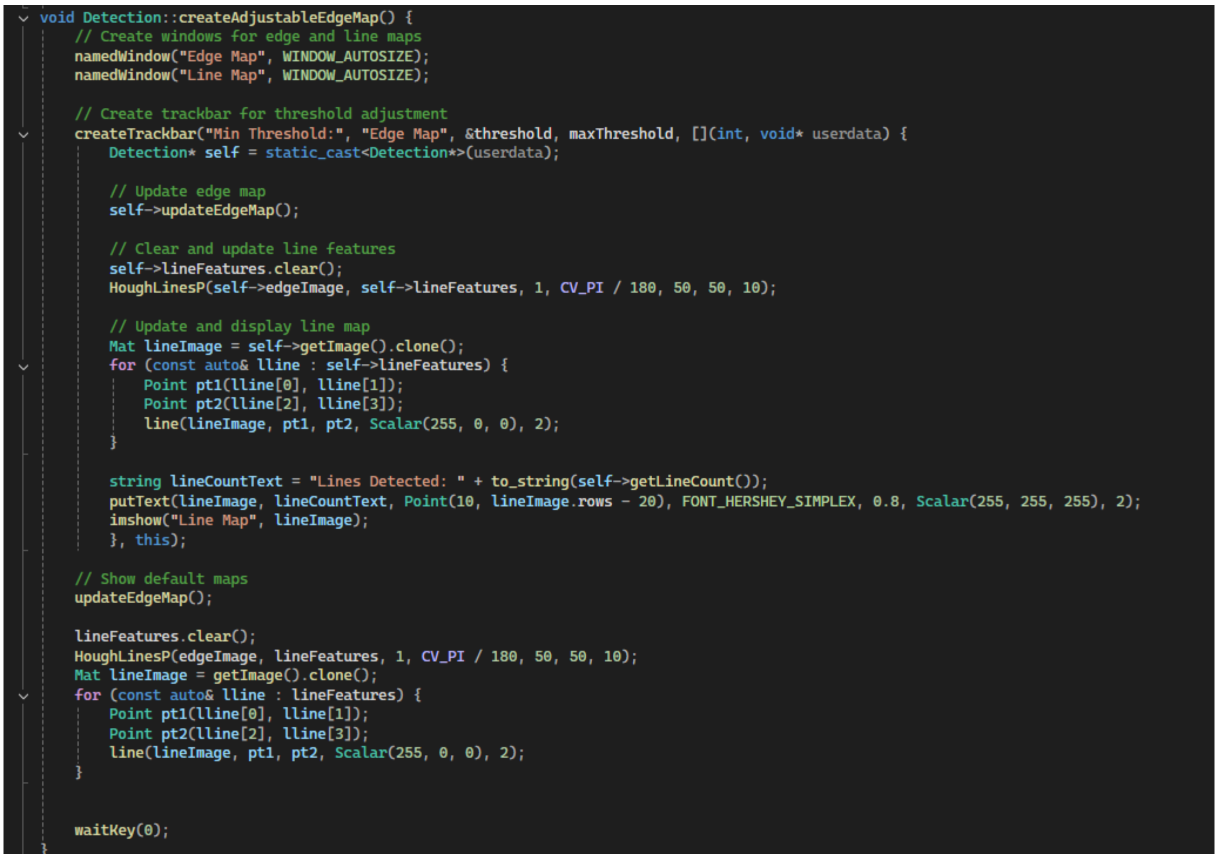
Figure 29 Saved Line Values



```cpp
/**
 * @brief Updates the edge map using the Canny edge detection algorithm.
 */
void Detection::updateEdgeMap() {
    if (getImage().empty()) {
        throw runtime_error("Gray level image is empty!");
    }

    // Generate edge map
    Canny(getImage(), edgeImage, threshold, threshold * 2);

    // Display edge map
    imshow("Edge Map", edgeImage);
    logMessage("Edge map updated with threshold: " + to_string(threshold));
}
```

Figure 28 Update Edge Map

The **updateEdgeMap** function checks if it is empty using the **getImage** member function. If it is empty, it sends an error. Then, the **Canny** function in **OpenCV** continuously finds the edges according to the specified threshold values and records this to the **edgeImage** variable. The updated thresh hold values are sent as a log message and the new image is printed on the screen with the **imshow** function.



```cpp
void Detection::createAdjustableEdgeMap() {
    // Create windows for edge and line maps
    namedWindow("Edge Map", WINDOW_AUTOSIZE);
    namedWindow("Line Map", WINDOW_AUTOSIZE);

    // Create trackbar for threshold adjustment
    createTrackbar("Min Threshold:", "Edge Map", &threshold, maxThreshold, [](int, void* userdata) {
        Detection* self = static_cast<Detection*>(userdata);

        // Update edge map
        self->updateEdgeMap();

        // Clear and update line features
        self->lineFeatures.clear();
        HoughLinesP(self->edgeImage, self->lineFeatures, 1, CV_PI / 180, 50, 50, 10);

        // Update and display line map
        Mat lineImage = self->getImage().clone();
        for (const auto& lline : self->lineFeatures) {
            Point pt1(lline[0], lline[1]);
            Point pt2(lline[2], lline[3]);
            line(lineImage, pt1, pt2, Scalar(255, 0, 0), 2);
        }

        string lineCountText = "Lines Detected: " + to_string(self->getLineCount());
        putText(lineImage, lineCountText, Point(10, lineImage.rows - 20), FONT_HERSHEY_SIMPLEX, 0.8, Scalar(255, 255, 255), 2);
        imshow("Line Map", lineImage);
        }, this);

    // Show default maps
    updateEdgeMap();

    lineFeatures.clear();
    HoughLinesP(edgeImage, lineFeatures, 1, CV_PI / 180, 50, 50, 10);
    Mat lineImage = getImage().clone();
    for (const auto& lline : lineFeatures) {
        Point pt1(lline[0], lline[1]);
        Point pt2(lline[2], lline[3]);
        line(lineImage, pt1, pt2, Scalar(255, 0, 0), 2);
    }


    waitKey(0);
}
```

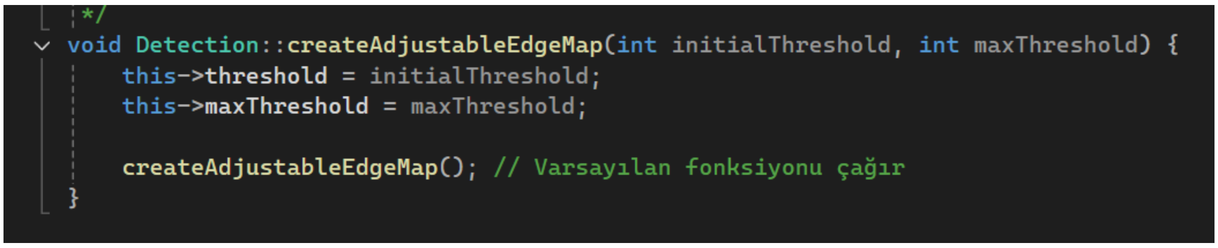
Figure 29 createAdjutableEdgeMap

This function allows the user to **interactively adjust** the edge detection map and line map of an image. Two windows are created inside the function, a trackbar is added to change a threshold value, and the detected lines are displayed dynamically. It allows us to create an **EdgeMap** that can be adjusted according to the threshold value. We create two windows called **EdgeMap** and **LineMap** with the **namedWindow** function in **OpenCV**. Again, we can create an adjustable bar with the **createTrackbar** function in the **OpenCV** library. The first input value of this bar is the text file that should be written to the left of the bar, namely **"Min Threshold"**, the second parameter is the window name **"Edge Map"**, the threshold value set by the user is kept in the **&threshold** variable. The **maxThreshold** value holds the maximum threshold limit. The **Lambda function** is the callback function to be called when the trackbar moves.

Detection* self = static_cast<Detection*>(userdata);

In this line, we assign a pointer to the Detection object. The goal is to reach the this pointer and access other members of the class.

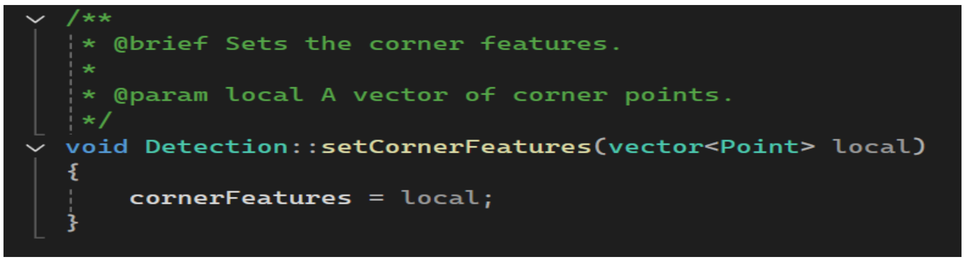
A new edge map is drawn in the **updateEdgeMap** function according to the threshold value. Then the line features are cleaned and new line values are created according to the new threshold value. Lines are detected with the HoughLinesP function in OpenCV and the detected features are stored in lineFeatures. CV_PI/180 angle resolution, 50 minimum vote requirement 10 minimum spacing of lines.



```
 */
void Detection::createAdjustableEdgeMap(int initialThreshold, int maxThreshold) {
    this->threshold = initialThreshold;
    this->maxThreshold = maxThreshold;

    createAdjustableEdgeMap(); // Varsayılan fonksiyonu çağır
}
```

Figure 30 createAdjustableEdgeMap

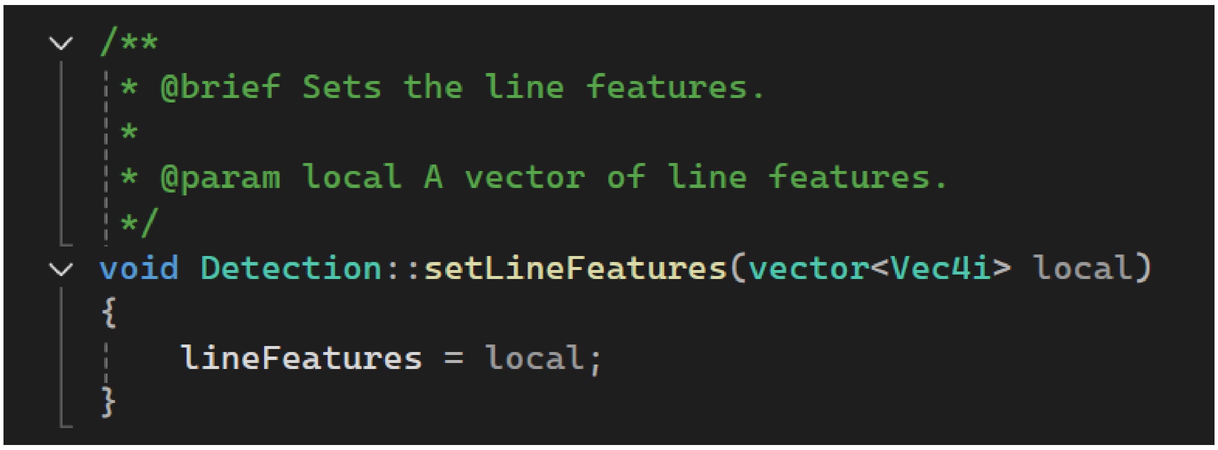
- Here, we rewrite the **createAdjustableEdgeMap** function, which is one of the best features of C++, by overloading the function. Here, the user can perform more specific operations and operations in smaller ranges by specifying special threshold values. If he calls the same function by giving parameters.



```
/**
 * @brief Sets the corner features.
 *
 * @param local A vector of corner points.
 */
void Detection::setCornerFeatures(vector<Point> local)
{
    cornerFeatures = local;
}
```

Figure 31 setCornerFeatures

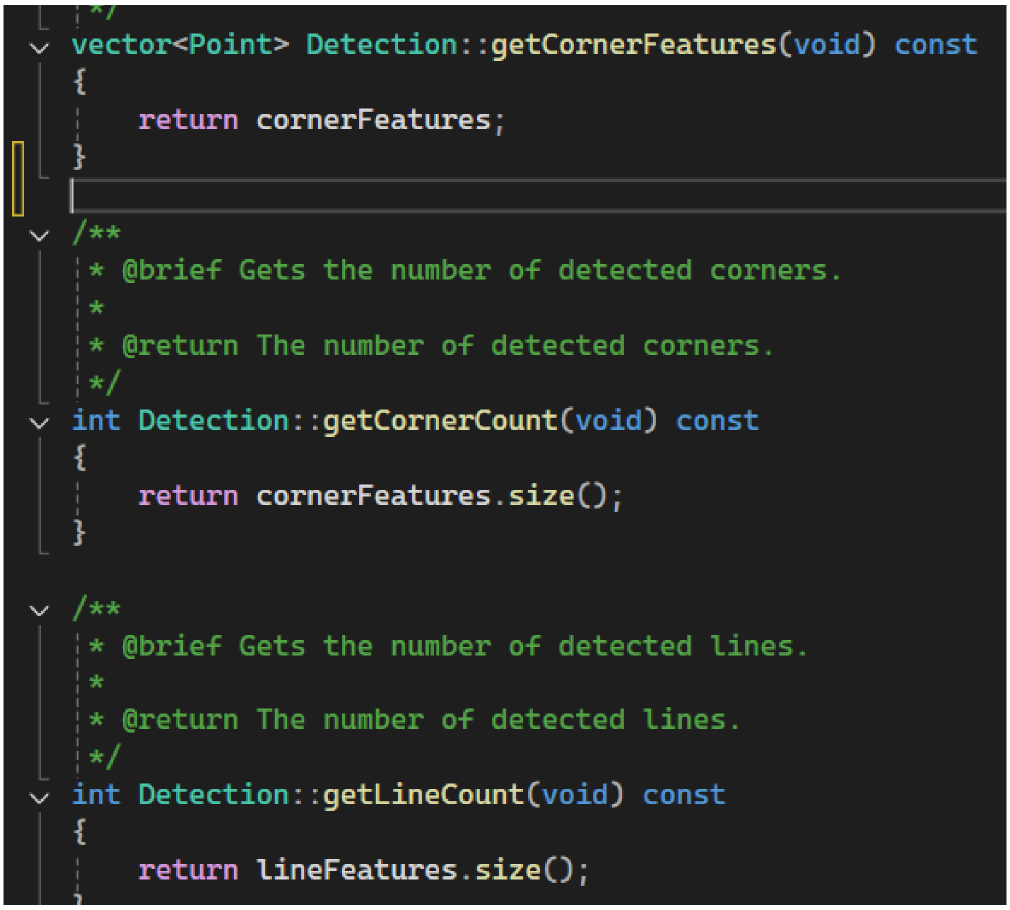
We can set the **CornerFeatures** values with the help of this function and make it possible to perform this assignment in derived classes as well. Because the **CornerFeatures** variable is **private**.



```
/**
 * @brief Sets the line features.
 *
 * @param local A vector of line features.
 */
void Detection::setLineFeatures(vector<Vec4i> local)
{
    lineFeatures = local;
}
```

Figure 32 setLineFeatures

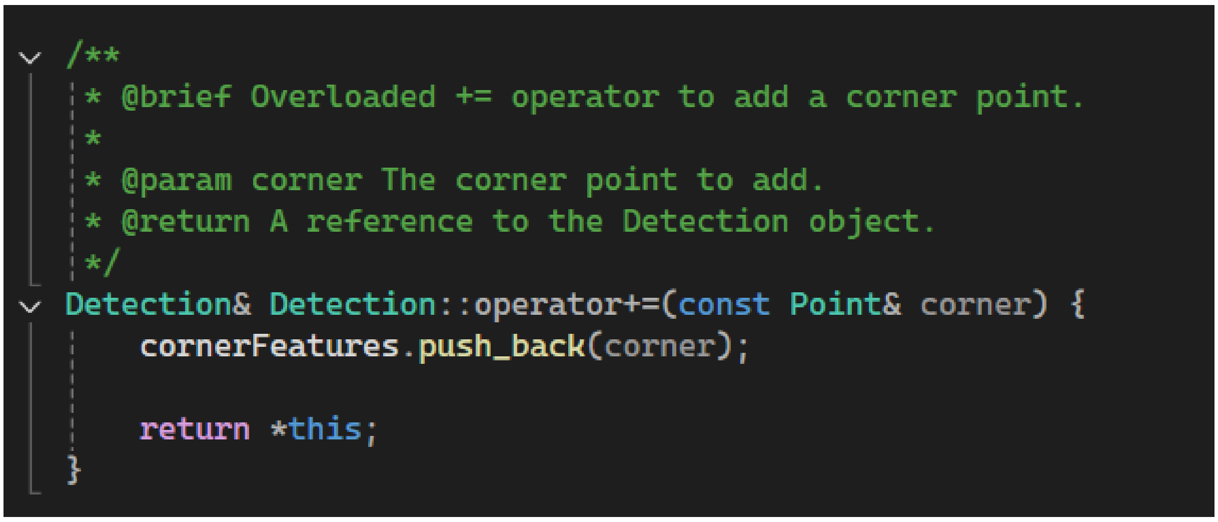
- Likewise, we wrote the **set** member function to set the **LineFeatures** variable from derived classes.



```
 */
vector<Point> Detection::getCornerFeatures(void) const
{
    return cornerFeatures;
}

/**
 * @brief Gets the number of detected corners.
 *
 * @return The number of detected corners.
 */
int Detection::getCornerCount(void) const
{
    return cornerFeatures.size();
}

/**
 * @brief Gets the number of detected lines.
 *
 * @return The number of detected lines.
 */
int Detection::getLineCount(void) const
{
    return lineFeatures.size();
```

Figure 33 Other getFunctions in Detection Class

➢ In order to return the detected vertex and line numbers to the user in an easy way and to further strengthen the **OOP structure**, we returned the detected vertex and line numbers with the help of the **.size()** function.

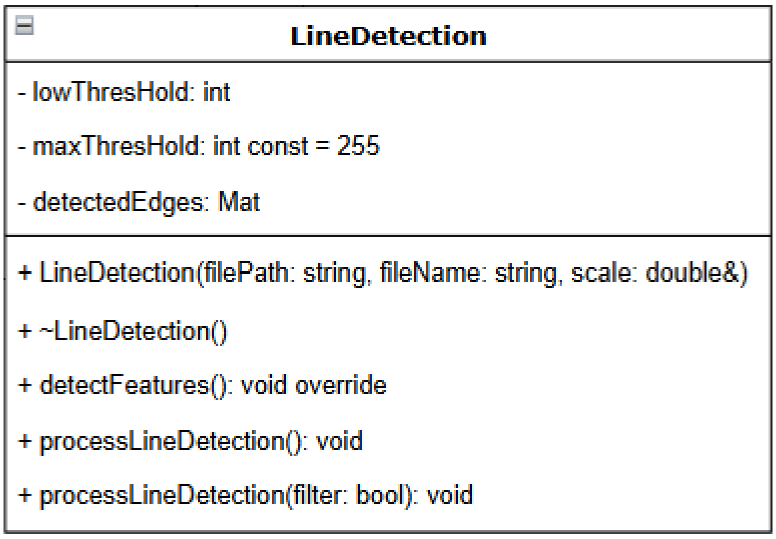


```cpp
/**
 * @brief Overloaded += operator to add a corner point.
 *
 * @param corner The corner point to add.
 * @return A reference to the Detection object.
 */
Detection& Detection::operator+=(const Point& corner) {
    cornerFeatures.push_back(corner);

    return *this;
}
```

Figure 34 Operator Overload

➢ In order to add new elements to the **cornerFeatures** vector more easily and conveniently, we overloaded the **+=** operator and added the value behind the vector with **.push_back(corner).** In this way, the **+** operator can also be used between **Point** objects.

LineDetection Class

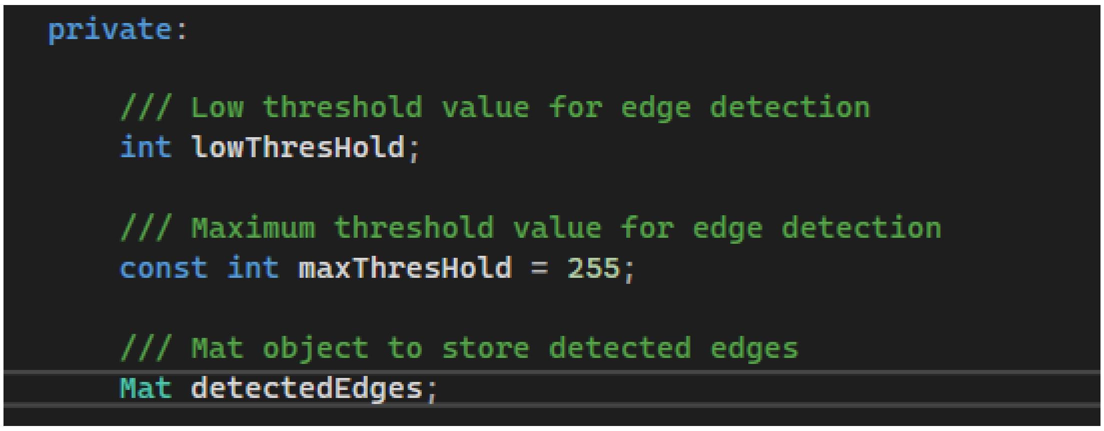


LineDetection

- lowThresHold: int
- maxThresHold: int const = 255
- detectedEdges: Mat

+ LineDetection(filePath: string, fileName: string, scale: double&)
+ ~LineDetection()
+ detectFeatures(): void override
+ processLineDetection(): void
+ processLineDetection(filter: bool): void

Figure 35 LineDetection Class

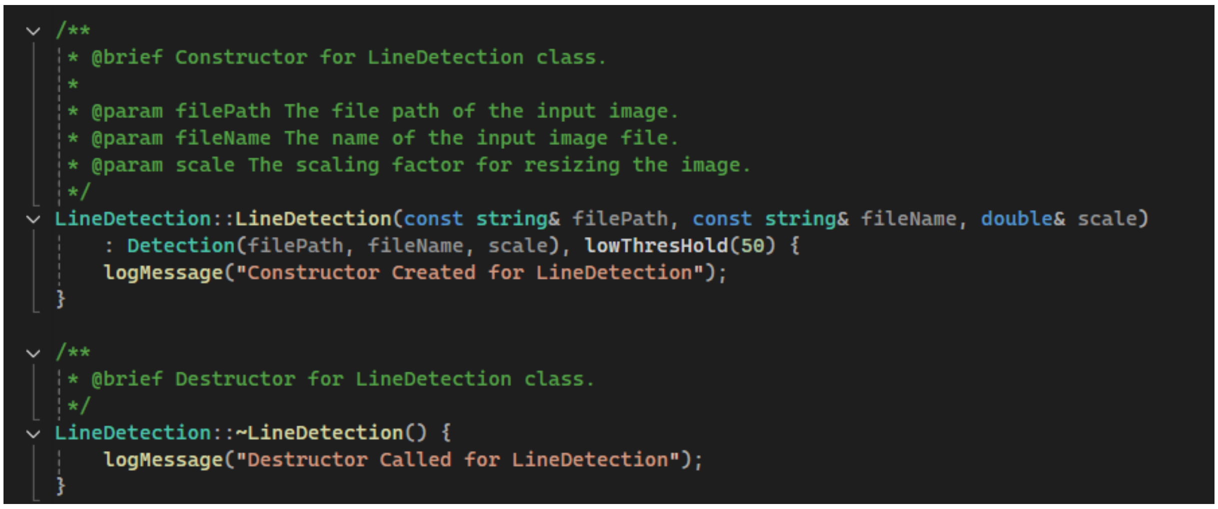


```
private:

    /// Low threshold value for edge detection
    int lowThresHold;

    /// Maximum threshold value for edge detection
    const int maxThresHold = 255;

    /// Mat object to store detected edges
    Mat detectedEdges;
```

Figure 36 Variables of LineDetection Class

- The **lowThreshHold** value is used to store the low threshold value.
- The **maxThreshold** value holds the maximum threshold value. As can be seen from the picture, we fixed it to **255**. Since it is const, we initialized it in the header file.
- The **detectedEdges** variable is a **Mat** variable and stores the detected edges.

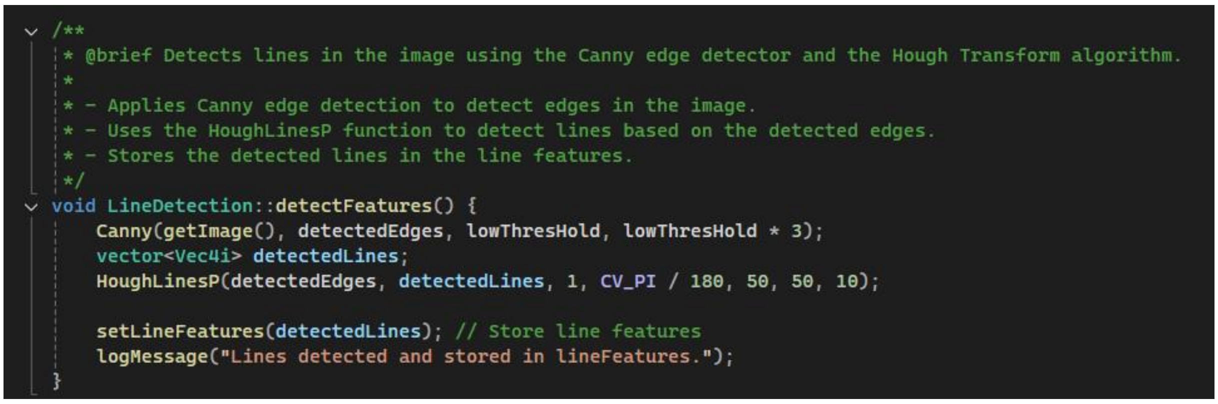
Member Functions



```
/**
 * @brief Constructor for LineDetection class.
 *
 * @param filePath The file path of the input image.
 * @param fileName The name of the input image file.
 * @param scale The scaling factor for resizing the image.
 */
LineDetection::LineDetection(const string& filePath, const string& fileName, double& scale)
    : Detection(filePath, fileName, scale), lowThresHold(50) {
    logMessage("Constructor Created for LineDetection");
}

/**
 * @brief Destructor for LineDetection class.
 */
LineDetection::~LineDetection() {
    logMessage("Destructor Called for LineDetection");
}
```

Figure 37 Constructor and Destructor for LineDetections

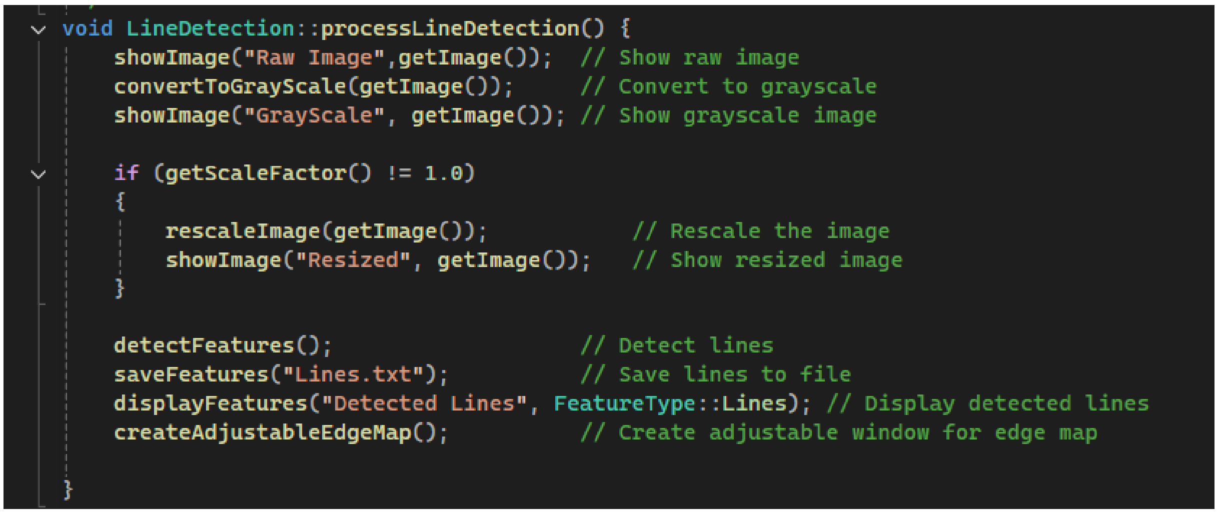
- When called in **Constructor** and **Destructor**, it prints a log message to the screen. Since the **LineDetection** Class is generated from the **Detection class,** we define the Detection class as the **initializer** method. And finally, we define the **lowThreshold** value as **50**.



```
/**
 * @brief Detects lines in the image using the Canny edge detector and the Hough Transform algorithm.
 *
 * - Applies Canny edge detection to detect edges in the image.
 * - Uses the HoughLinesP function to detect lines based on the detected edges.
 * - Stores the detected lines in the line features.
 */
void LineDetection::detectFeatures() {
    Canny(getImage(), detectedEdges, lowThresHold, lowThresHold * 3);
    vector<Vec4i> detectedLines;
    HoughLinesP(detectedEdges, detectedLines, 1, CV_PI / 180, 50, 50, 10);

    setLineFeatures(detectedLines); // Store line features
    logMessage("Lines detected and stored in lineFeatures.");
}
```

Figure 38 detectFeatures

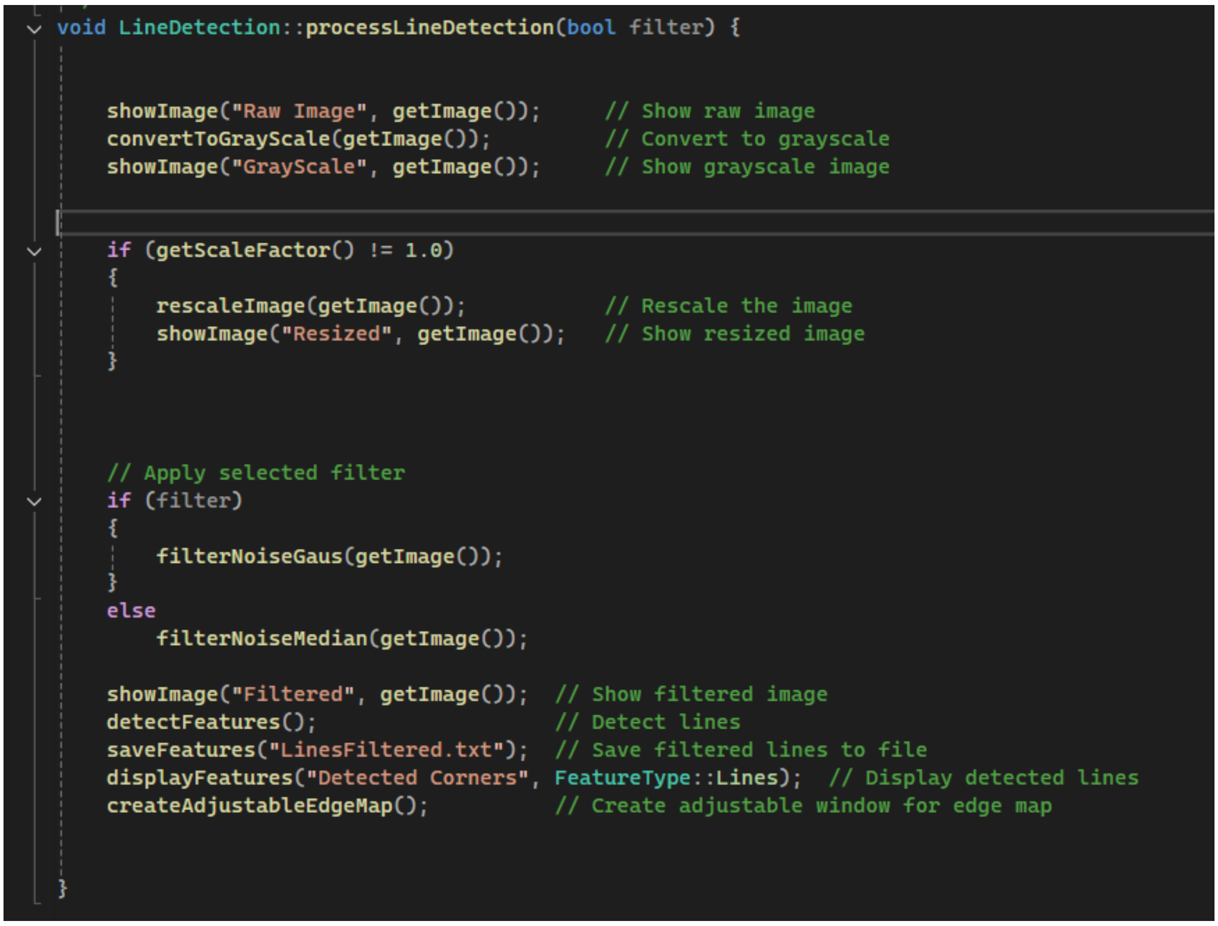
➢ We perform line detection with the **detectFeatures** function, first we perform edge detection with the **Canny** function in **OpenCV**. The **Canny** function was mentioned earlier in the report. Then the detected edges are transferred to the **detectedLines**, which is a 4-parameter vector, with **HoughLinesP** in **OpenCV**, and set with the set function. Finally, the log message is printed on the screen that the process is completed.



```
void LineDetection::processLineDetection() {
    showImage("Raw Image",getImage());  // Show raw image
    convertToGrayScale(getImage());     // Convert to grayscale
    showImage("GrayScale", getImage()); // Show grayscale image

    if (getScaleFactor() != 1.0)
    {
        rescaleImage(getImage());           // Rescale the image
        showImage("Resized", getImage());   // Show resized image
    }

    detectFeatures();                   // Detect lines
    saveFeatures("Lines.txt");          // Save lines to file
    displayFeatures("Detected Lines", FeatureType::Lines); // Display detected lines
    createAdjustableEdgeMap();          // Create adjustable window for edge map

}
```

Figure 39 processLineDetection

It is a function that carries out the line detection event completely from beginning to end by containing all the written functions. It suppresses the function with **showImage**. Then, it converts the image to gray level with the **convertToGrayScale** function, that is, it manipulates the image. Then, **GrayScale** suppresses the image. Then, it calls the scale factor with the get function and checks if it has changed. If it has changed, it changes the size of the image with the **rescaleImage** function. Then, it detects the lines with the **detectFeatures()** function and saves them to Lines.txt with the help of **saveFeatures("Lines.txt").** It also shows the lines on the image with the **displayFeatures** function. Finally, it prints an image whose threshold value can be controlled with the **createAdjustableEdgeMap()** function.

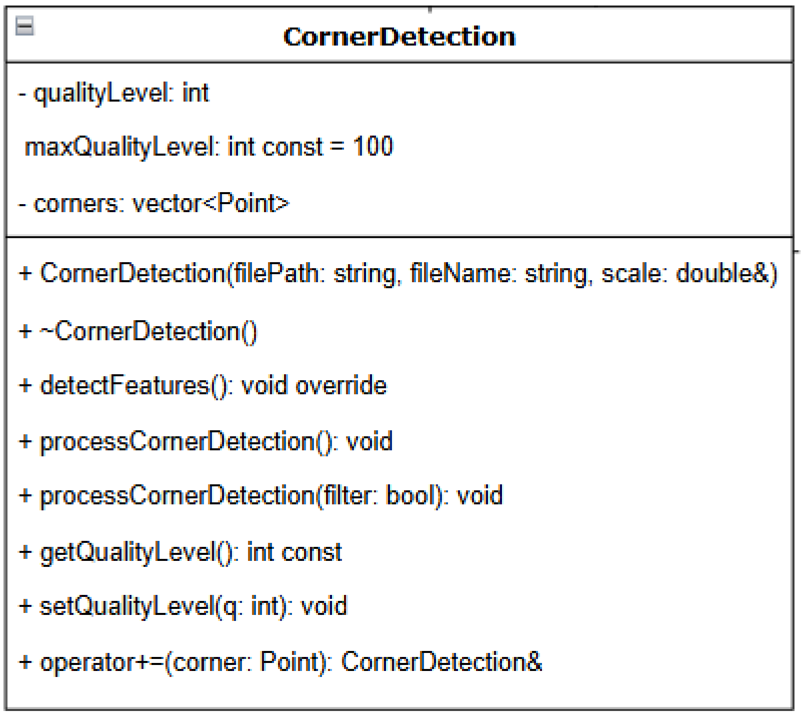


```
void LineDetection::processLineDetection(bool filter) {


    showImage("Raw Image", getImage());     // Show raw image
    convertToGrayScale(getImage());         // Convert to grayscale
    showImage("GrayScale", getImage());     // Show grayscale image


    if (getScaleFactor() != 1.0)
    {
        rescaleImage(getImage());           // Rescale the image
        showImage("Resized", getImage());   // Show resized image
    }



    // Apply selected filter
    if (filter)
    {
        filterNoiseGaus(getImage());
    }
    else
        filterNoiseMedian(getImage());

    showImage("Filtered", getImage());  // Show filtered image
    detectFeatures();                   // Detect lines
    saveFeatures("LinesFiltered.txt");  // Save filtered lines to file
    displayFeatures("Detected Corners", FeatureType::Lines);  // Display detected lines
    createAdjustableEdgeMap();          // Create adjustable window for edge map


}
```

Figure 40 processLineDetection overloaded

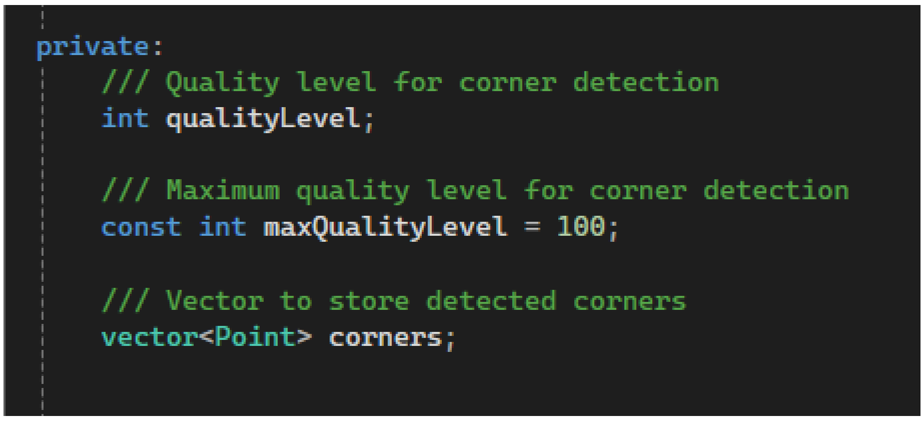
Here, unlike the first **processLineDetection** function, we take a **bool** value as input. And according to this value, in addition to the previous steps, we apply the filter operations in **CommonProcesses** before the detection process. If **TRUE** or **1** is entered, the Gaussian filter is applied with the **filterNoiseGaus()** function. If **FALSE** or **0** is entered, the median filter is applied with the **filterNoiseMedyan()** function, reducing high-frequency noise and smoothing the image.

CornerDetection



```
CornerDetection
- qualityLevel: int
 maxQualityLevel: int const = 100
- corners: vector<Point>
+ CornerDetection(filePath: string, fileName: string, scale: double&)
+ ~CornerDetection()
+ detectFeatures(): void override
+ processCornerDetection(): void
+ processCornerDetection(filter: bool): void
+ getQualityLevel(): int const
+ setQualityLevel(q: int): void
+ operator+=(corner: Point): CornerDetection&
```

Figure 41 Corner Detection UML



```
private:
    /// Quality level for corner detection
    int qualityLevel;

    /// Maximum quality level for corner detection
    const int maxQualityLevel = 100;

    /// Vector to store detected corners
    vector<Point> corners;
```

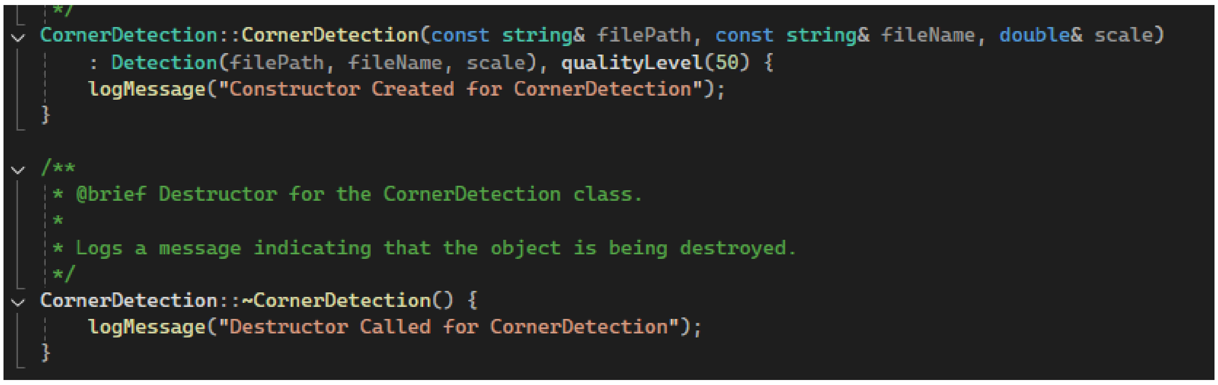
Figure 42 CornerDetection Class variables

Variables

- **qualityLevel** Determines the quality level used in corner detection. This value represents the minimum quality level of detected corners.
- const int **maxQualityLevel** = **100:** The maximum quality level setting that can be set for corner details and is a fixed value.

- **vector<Point> corners**: A vector that stores the coordinates of the detected corners. Each corner is stored as a Point object representing its **(x, y)** coordinates.

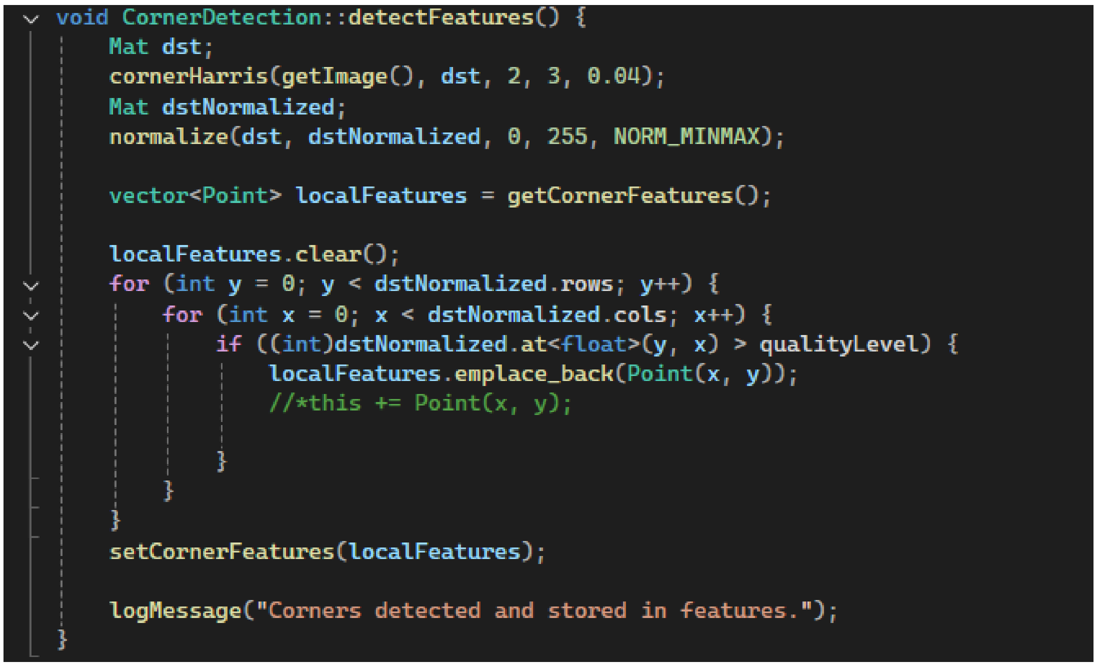
Member Functions



```cpp
*/
CornerDetection::CornerDetection(const string& filePath, const string& fileName, double& scale)
    : Detection(filePath, fileName, scale), qualityLevel(50) {
    logMessage("Constructor Created for CornerDetection");
}

/**
 * @brief Destructor for the CornerDetection class.
 *
 * Logs a message indicating that the object is being destroyed.
 */
CornerDetection::~CornerDetection() {
    logMessage("Destructor Called for CornerDetection");
}
```

Figure 43 Constructor and Destructor For CornerDetection

Since the **Cornerdetection** class is generated from the **Detection** class, we defined the Detection class with the initializer list method just like **Linedetection** and then we defined the **qualityLevel** variable with 50. We checked from the console whether the Constructor and Destructor were called with the **LogMessage** function we wrote earlier.

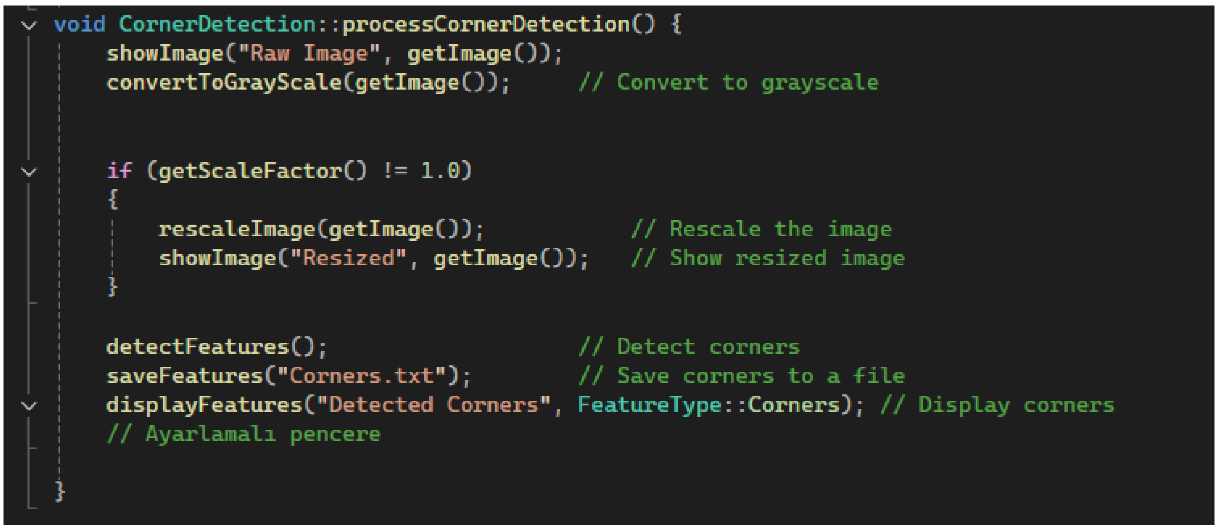


```cpp
void CornerDetection::detectFeatures() {
    Mat dst;
    cornerHarris(getImage(), dst, 2, 3, 0.04);
    Mat dstNormalized;
    normalize(dst, dstNormalized, 0, 255, NORM_MINMAX);

    vector<Point> localFeatures = getCornerFeatures();

    localFeatures.clear();
    for (int y = 0; y < dstNormalized.rows; y++) {
        for (int x = 0; x < dstNormalized.cols; x++) {
            if ((int)dstNormalized.at<float>(y, x) > qualityLevel) {
                localFeatures.emplace_back(Point(x, y));
                //*this += Point(x, y);

            }
        }
    }
    setCornerFeatures(localFeatures);

    logMessage("Corners detected and stored in features.");
}
```

Figure 44 detectFeatures

The **detectFeatures** function helps us find corner points here as in **LineDetection**. We detect corner points with the **cornerHarris** function in **OpenCV**. We get the image with **getImage().** dst is the output matrix, 2 is the block size around pixels, and 3 is the kernel size of the **sobel** filter. **dstNormalized** keeps the values normalized between **0** and **255**.

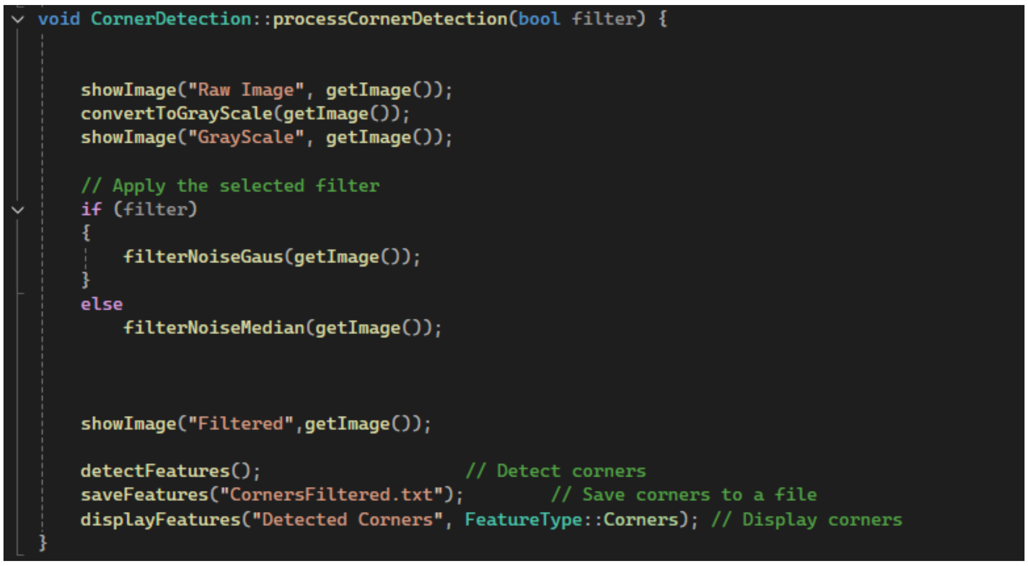
Again, the normalize function in **OpenCV** performs this function. **NORM_MINMAX** means that it will be normalized using the minimum and maximum values. The aim is to pull the values between 0 and 255. The detected points are converted to a Point vector called **locakFeatures**. If the normalized value is greater than the **qualityLevel** Variable, the values are transferred to the **localFeatures** vector. Then, the **localFeatures** values are assigned from the local to the main variable with the set function. And finally, a log message is printed on the screen where the transaction is made.



```cpp
void CornerDetection::processCornerDetection() {
    showImage("Raw Image", getImage());
    convertToGrayScale(getImage());     // Convert to grayscale


    if (getScaleFactor() != 1.0)
    {
        rescaleImage(getImage());           // Rescale the image
        showImage("Resized", getImage());   // Show resized image
    }

    detectFeatures();                   // Detect corners
    saveFeatures("Corners.txt");        // Save corners to a file
    displayFeatures("Detected Corners", FeatureType::Corners); // Display corners
    // Ayarlamalı pencere

}
```

Figure 45 processCornerDetection

Just like in **LineDetection**, each step is done in order and all steps are performed at this stage. First, the Raw Image is printed on the screen. Then, the Raw Image is converted to gray level with the help of the **convertToGrayScale()** function. Then, the **scaleFactor** variable is called with **getScaleFactor()** and the scale factor is checked. If the value is different from 1, the image is resized and the resized image is printed on the screen. Then, the corners are found with **detectFeatures()** and the corner value coordinates are written to Corners.txt with **saveFeatures("Corners.txt")**. Finally, the output is printed on the screen with the **disPlayFeatures()** function.

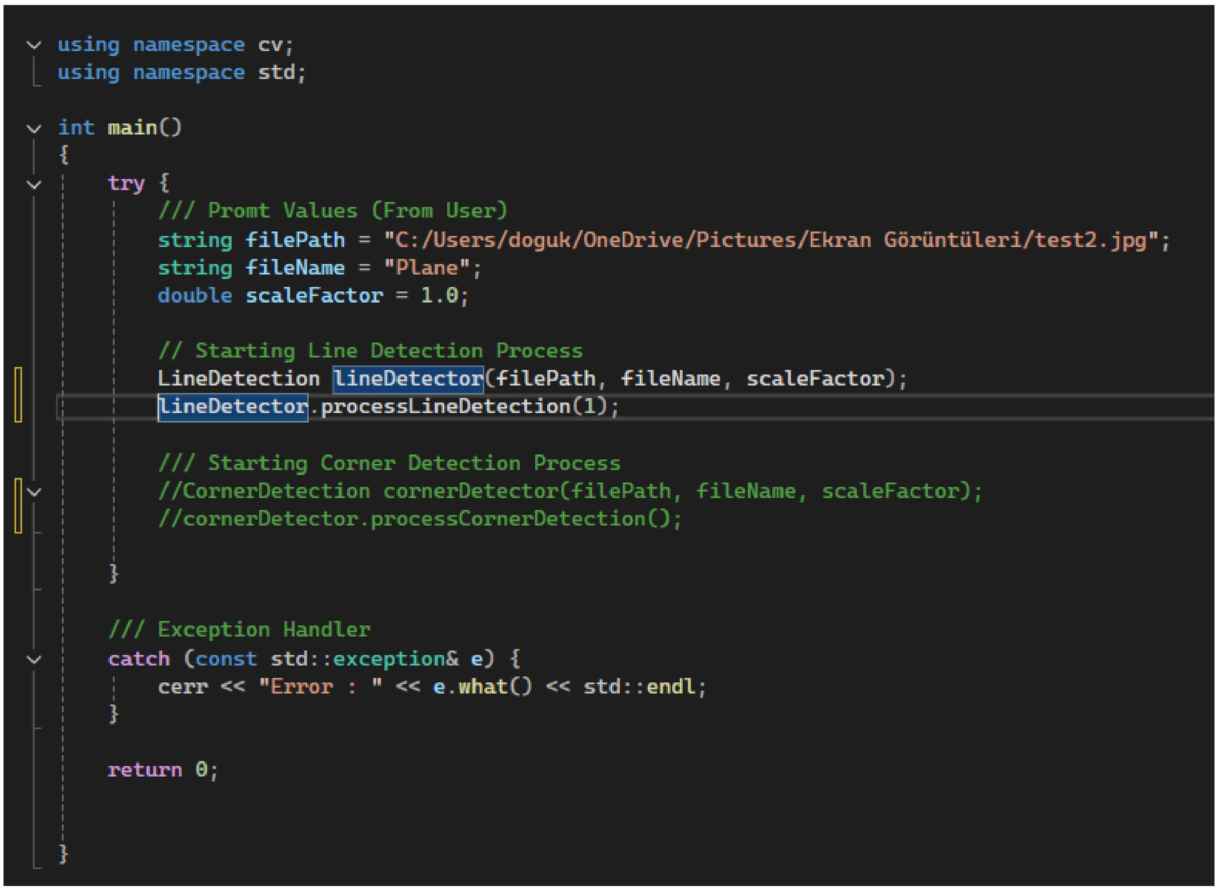


```
void CornerDetection::processCornerDetection(bool filter) {


    showImage("Raw Image", getImage());
    convertToGrayScale(getImage());
    showImage("GrayScale", getImage());

    // Apply the selected filter
    if (filter)
    {
        filterNoiseGaus(getImage());
    }
    else
        filterNoiseMedian(getImage());



    showImage("Filtered",getImage());

    detectFeatures();                   // Detect corners
    saveFeatures("CornersFiltered.txt");        // Save corners to a file
    displayFeatures("Detected Corners", FeatureType::Corners); // Display corners
}
```

Figure 46 Overloaded processCornerDetection

It takes a **bool** value as input and according to the selected **bool** value, if the value is **TRUE**, that is **1**, the **Gaussian filter** is applied with **filterNoiseGaus()**, if it is **FALSE**, that is **0**, the **median filter** is applied with **filterNoiseMedian()** and other operations continue the same.

Main Code (Driver Code)



```
using namespace cv;
using namespace std;

int main()
{
    try {
        /// Promt Values (From User)
        string filePath = "C:/Users/doguk/OneDrive/Pictures/Ekran Görüntüleri/test2.jpg";
        string fileName = "Plane";
        double scaleFactor = 1.0;

        // Starting Line Detection Process
        LineDetection lineDetector(filePath, fileName, scaleFactor);
        lineDetector.processLineDetection(1);

        /// Starting Corner Detection Process
        //CornerDetection cornerDetector(filePath, fileName, scaleFactor);
        //cornerDetector.processCornerDetection();

    }

    /// Exception Handler
    catch (const std::exception& e) {
        cerr << "Error : " << e.what() << std::endl;
    }

    return 0;


}
```

Figure 47 Main Code

RESULTS FOR LINE DETECTION

Input Raw Image

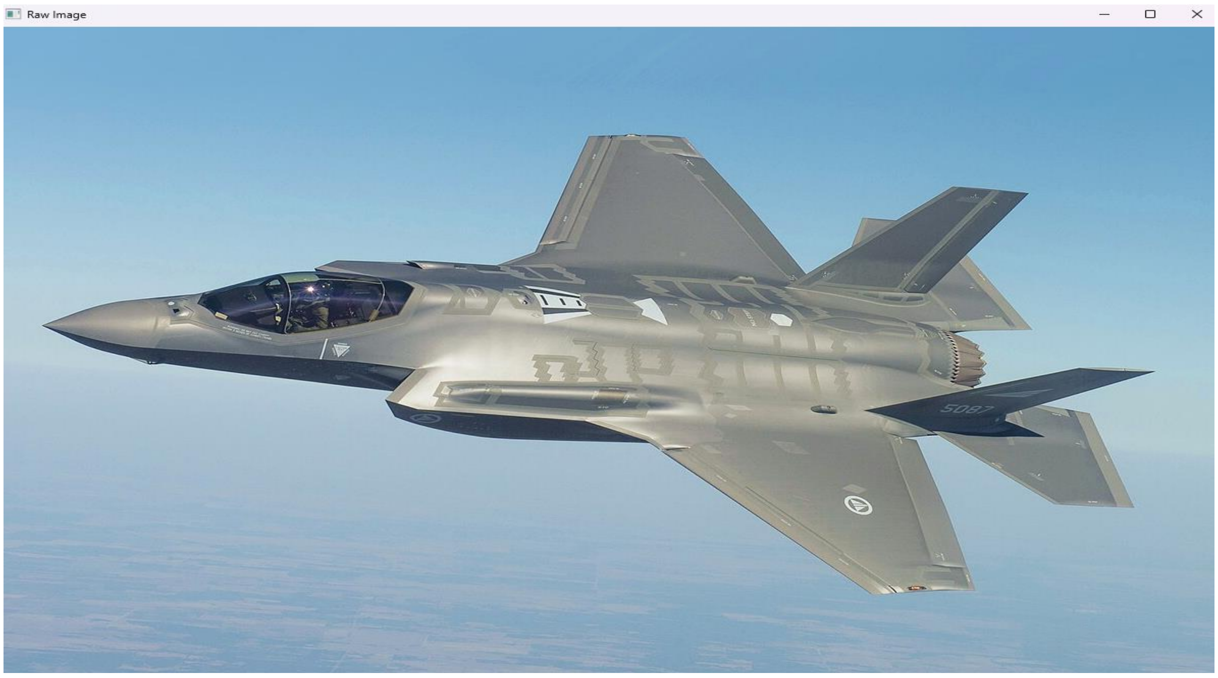


Figure 48 Raw Image

Gray Level Image:

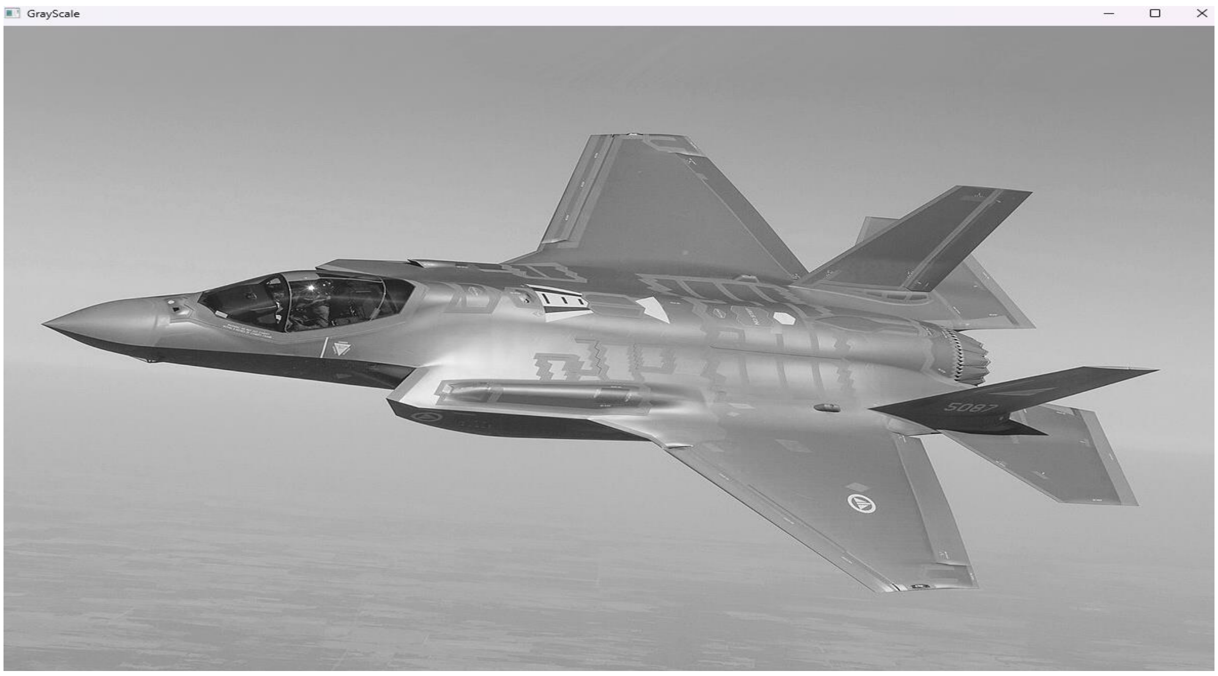


Figure 51 Gray Scaled Image

Detected Lines Threshold = 100:

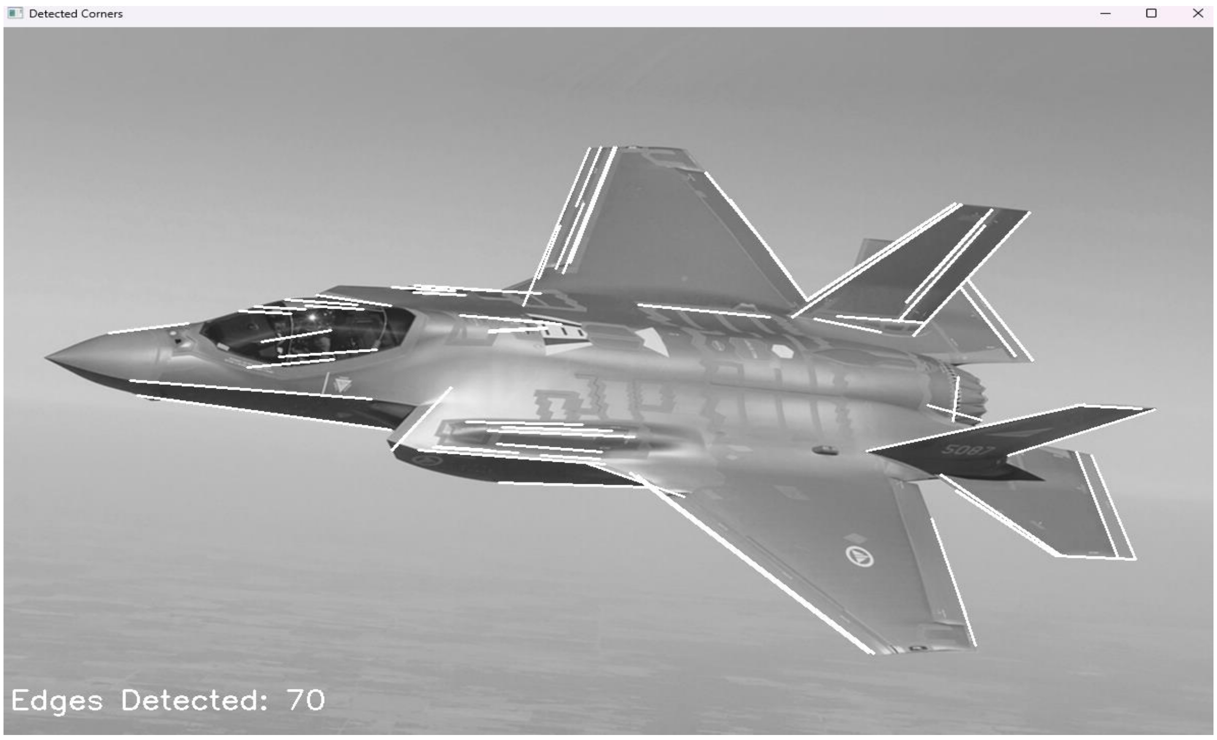


Figure 52 Detected Lines

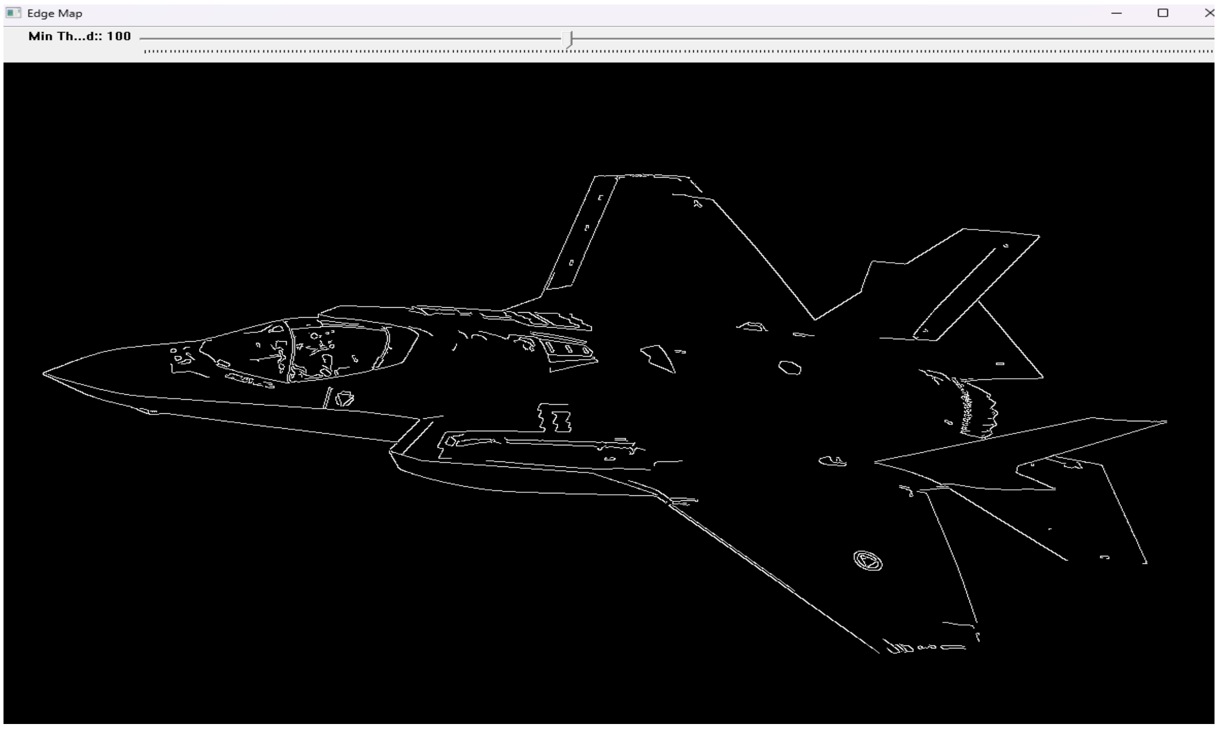


Figure 53 Edge Map

Detected Lines Threshold = 245:

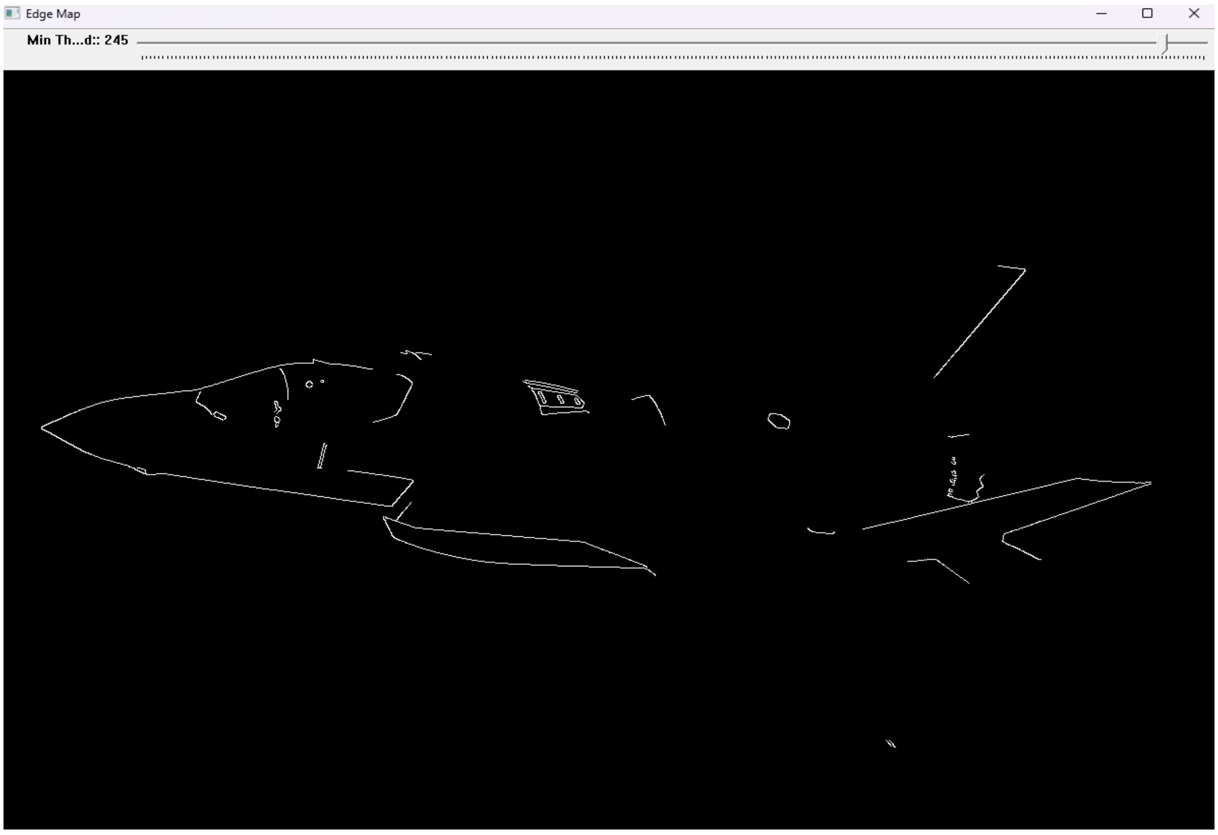
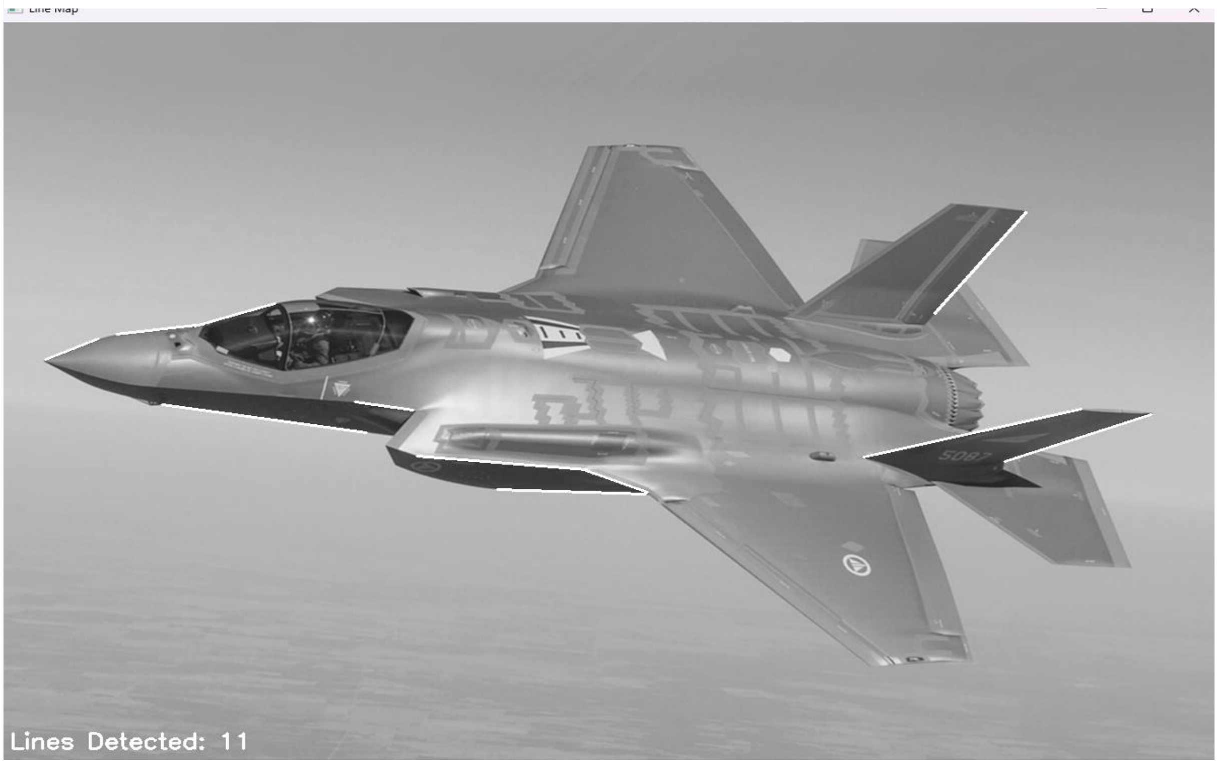


Figure 54 Edge Map 2



Lines Detected: 11

Figure 55 Detected Lines 2

Detected Lines Threshold = 17 :

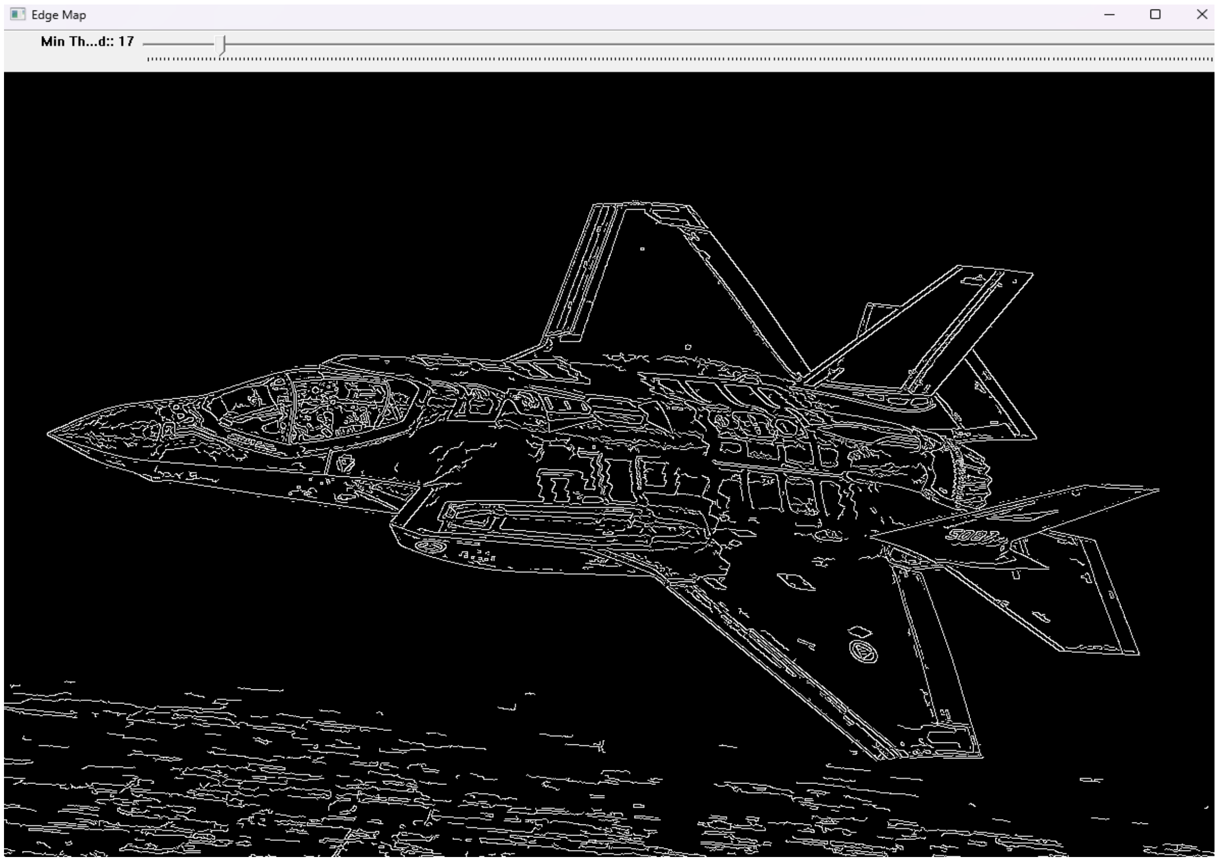


Figure 56 Edge Map 3

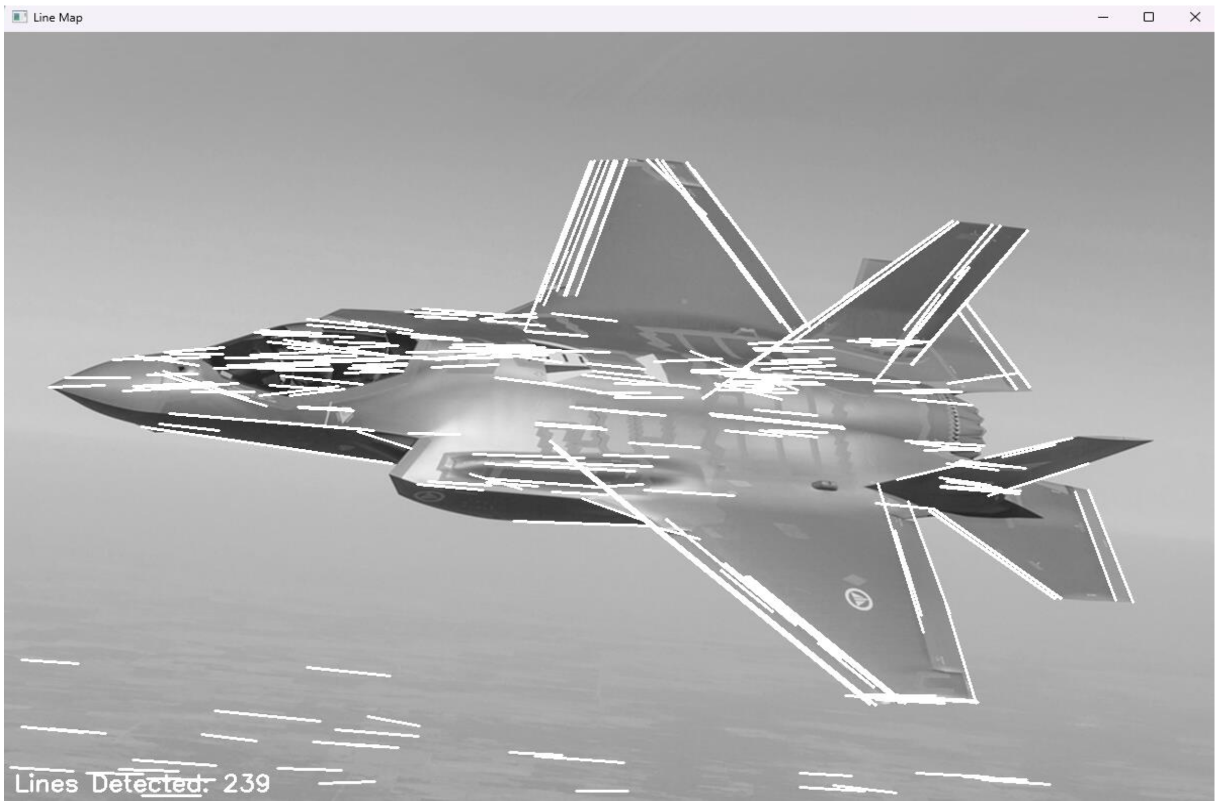


Figure 57 Lıne Detection 3

RESULTS FOR CORNER DETECTION

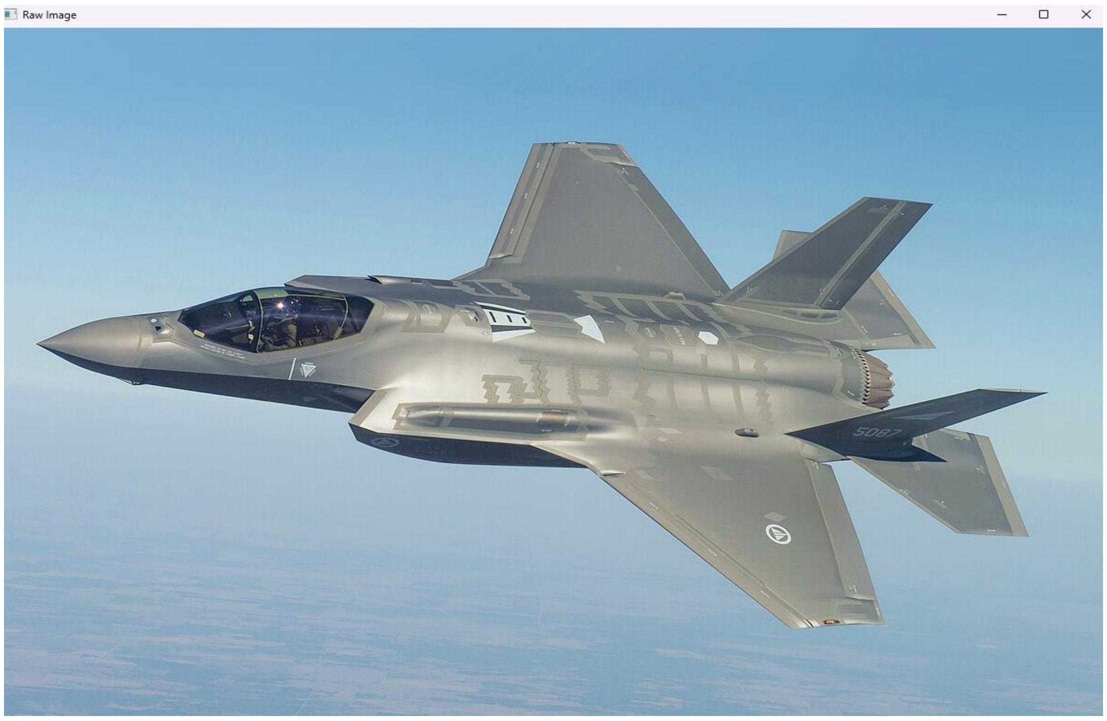
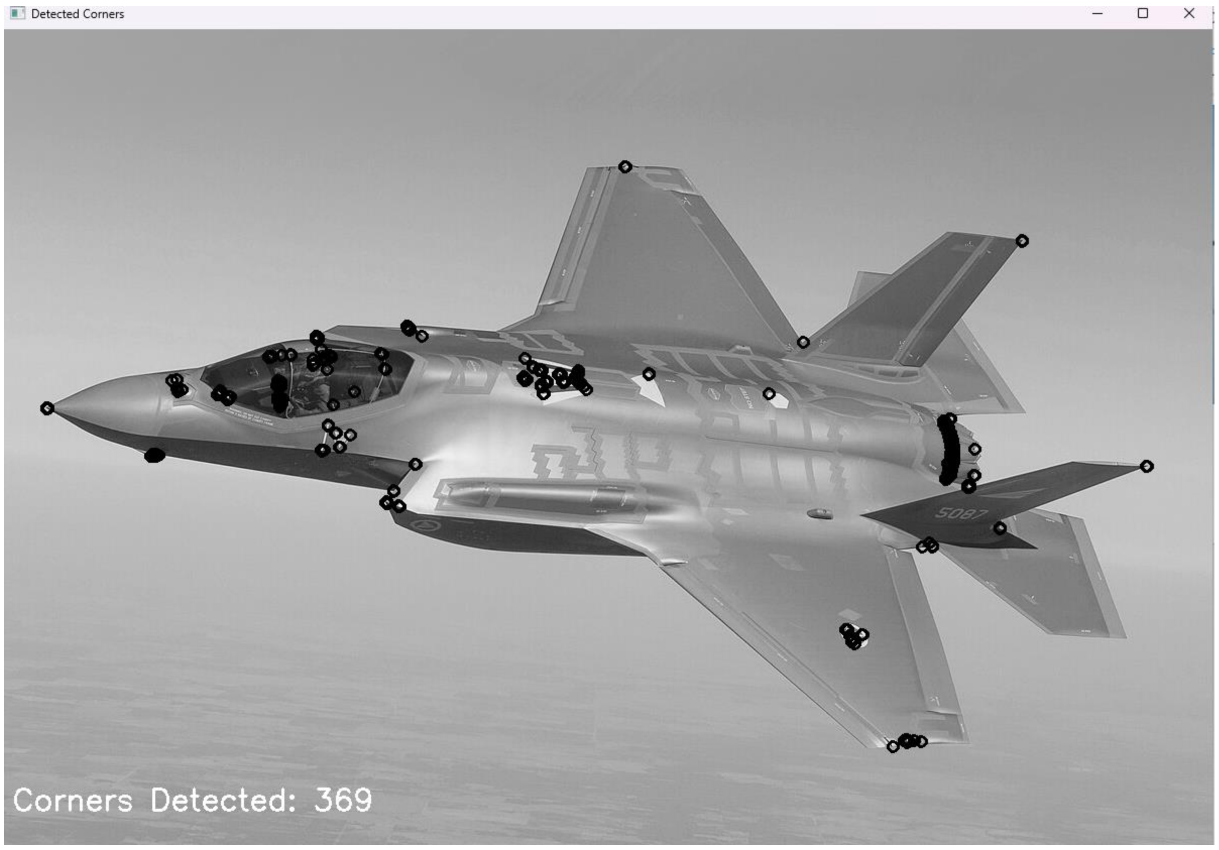


Figure 58 Input Raw Image

After Corner detection :



Corners Detected: 369

Figure 59 Corner Detection