Polynomial Line Detection

Tyler Goyea

August 4, 2025

1 Image Histogram and Segmentation

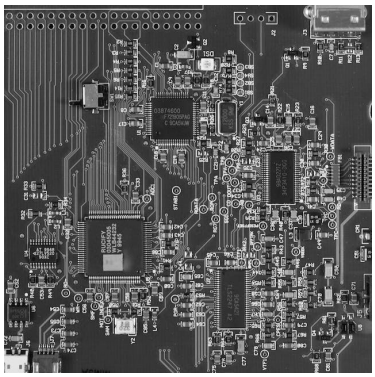
1.1 Histogram

The full code to *histogram.cpp*, (part 3.1 and 3.2) is available in the **appendix A**. However, function *createHistogram* is included just below:

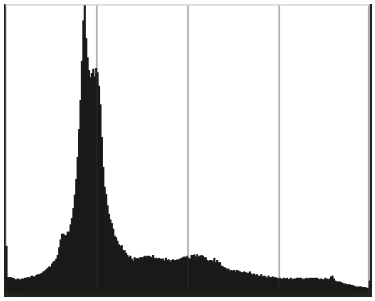
```
void createHistogram(Mat &img, Mat &hist)
{
    int histSize = 256; // Number of bins (256 possible
        intensity values for grayscale)
    vector<int> histData(histSize, 0);
    // Create a black canvas for the histogram image
    Mat histImage(HIST_IMG_HEIGHT, 512, CV_8UC3,
        Scalar(255, 255, 255)); // Black canvas to draw
        the histogram

    bool uniform = true, accumulate = false;
    int imgRows = img.rows;
    int imgCols = img.cols;
    int max_height = 0;

    // 1. Iterate through the image to calculate the
        histogram (frequency of pixel intensities)
    for (int i = 0; i < imgRows; i++)
    {
        for (int j = 0; j < imgCols; j++)
        {
            // Get the pixel intensity value (grayscale
                value)
            int pixelValue = img.at<uchar>(i, j);

            // Increment the corresponding histogram bin
                for this pixel intensity
            histData[pixelValue]++;
            max_height = max(histData[pixelValue],
                max_height);
        }
    }

    // 2. Normalize the histogram
    int hist_w = 512;
    int hist_h = HIST_IMG_HEIGHT;
    int bin_w = cvRound((double)hist_w / histSize); //
        Width of each bin
```

```
    for (int i = 0; i < histSize; i++)
    {
        histData[i] = cvRound((float)histData[i] *
            hist_h / max_height); // Normalize based on
            max count
    }

    for (int i = 0; i < histSize; i++)
    {
        int binVal = histData[i]; // already normalized
            value
        // Calculate the x position and width of each bin
        int x = i * bin_w;
        // Draw a filled rectangle for each bin
        rectangle(histImage, Point(x, hist_h - binVal),
            Point(x + bin_w, hist_h),
                  Scalar(25, 25, 25), cv::FILLED);
    }

    // Add vertical lines at intensity 127 and 255
    int lineHeight = hist_h; // The line spans the full
        height of the histogram
    line(histImage, Point(255 * bin_w, 0), Point(255 *
        bin_w, lineHeight), Scalar(50, 50, 50), 1, 4, 0);
    line(histImage, Point(192 * bin_w, 0), Point(192 *
        bin_w, lineHeight), Scalar(50, 50, 50), 1, 4, 0);
    line(histImage, Point(128 * bin_w, 0), Point(128 *
        bin_w, lineHeight), Scalar(50, 50, 50), 1, 4, 0);
    line(histImage, Point(64 * bin_w, 0), Point(64 *
        bin_w, lineHeight), Scalar(50, 50, 50), 1, 4, 0);

    // Set the calculated histogram image in the input
        parameter "hist"
    hist = histImage;
}
```

The results of the **histogram.cpp** are



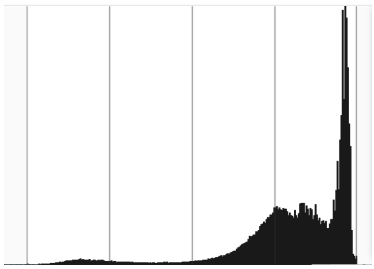
(a) Original Image



(b) Histogram

Figure 1: histogram.cpp applied to circuit_board.jpg

(a) Original Image

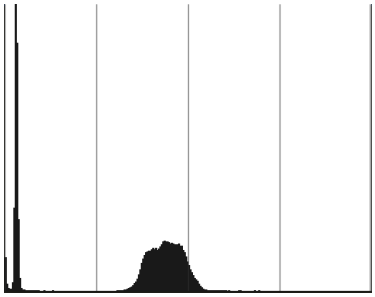


(b) Histogram

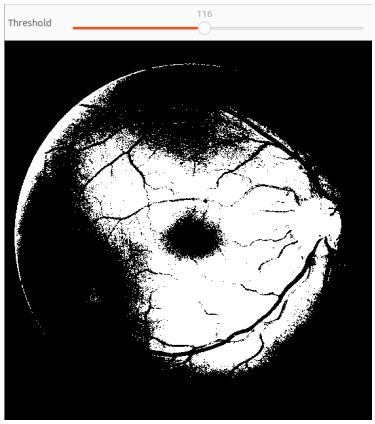
Figure 2: histogram.cpp applied to science_person.jpg

1.2 Thresholding

(a) Original Image



(b) Histogram



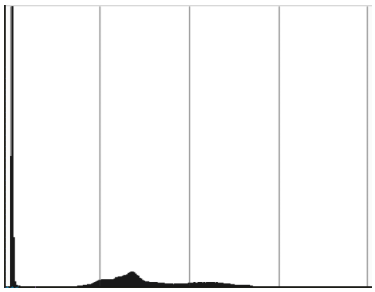
(c) Threshold 1

(d) Threshold 2

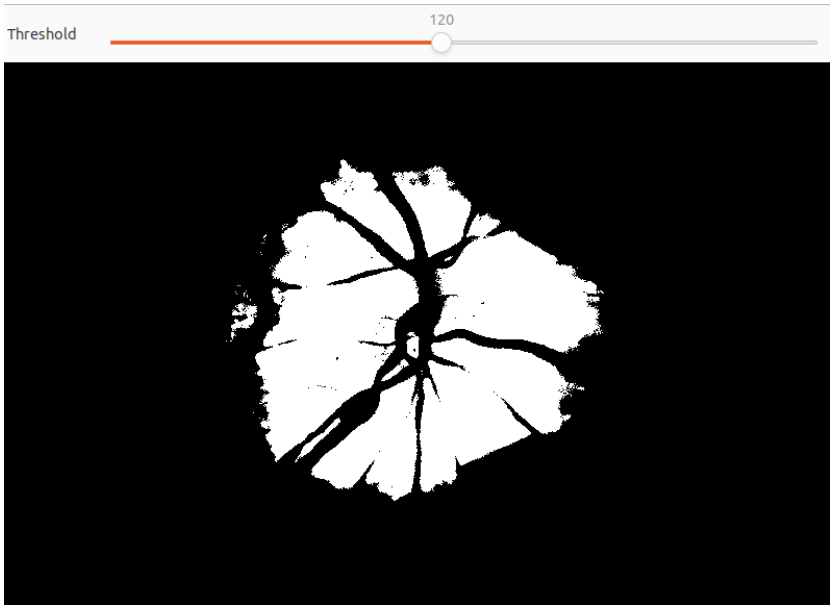
Figure 3: histogram.cpp applied to fundus.tif

Threshold 1 has a lower bound threshold of 116 and Threshold 2 has one of 98. A key observation is, as seen with Threshold 1, lighter blood vessels are more visible and dark blood vessels are becoming engulfed by the blackness from the threshold. However, for Threshold 2 darker vessels are only visible with lighter vessels being invisible.

(a) Original Image



(b) Histogram

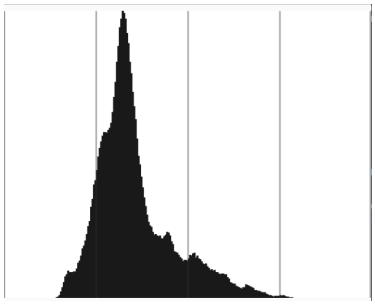


(c) Threshold 1

Figure 4: histogram.cpp applied to glaucoma.jpg

Threshold 1 has a lower bound threshold of 120. The thresholding seems to capture the shape of the border of the lighter region but lost all the detail outside of the border.

(a) Original Image



(b) Histogram

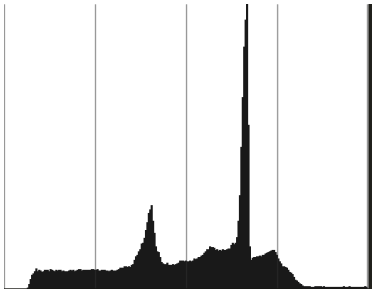
(c) Threshold 1

Figure 5: histogram.cpp applied to optic_nerve_head.jpg

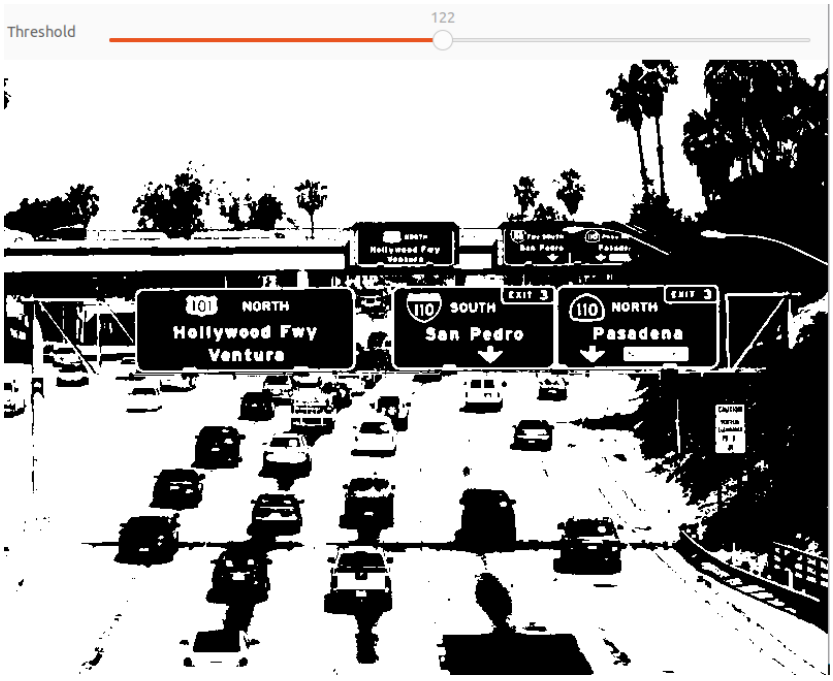
Threshold 1 has a lower bound threshold of 95. Smaller faint vessels were not

captured during the thresholding very well.

(a) Original Image



(b) Histogram



(c) Threshold 1

Figure 6: histogram.cpp applied to motorway.png

Threshold 1 has a lower bound threshold of 122. Further away text is less clear with the thresholding and this thresholding value also captures the borders of the signs well, which makes it easier for visual computing methods to identify what is a sign.

2 Horizon Detection

2.1 Processing Pipeline

See **Appendix B** for the full code for *horizon.cpp*. In short, pre-processing was limited to simply blurring using *cv::blur*. A collection of tracker bars were used to vary the following parameters for *cv::HoughLinesP* & *cv::Canny*:

- **Min Threshold:** `lowThreshold` – range: 30 to 100
- **Lowerthreshold Difference to Upperthreshold (* 0.1):** `thresholdWindow` – range: 1 to 100
- **Blur Size:** `xBlur` – range: 1 to 100
- **Max Line Length:** `Max_Line_Len` – range: 1 to 200
- **Polynomial Degree:** `DEGREE` – range: 1 to 6
- **Minimum Change in X of Line:** `MIN_DX` – range: 0 to 100
- **Rho:** `rho` – range: 1 to 10
- **Theta = π/value:** `value` – range: 180 to 360
- **Hough Threshold:** `houghThreshold` – range: 1 to 100
- **Minimum Length of Hough:** `minLen` – range: 10 to 100
- **Max Gap of Hough:** `maxGap` – range: 5 to 100

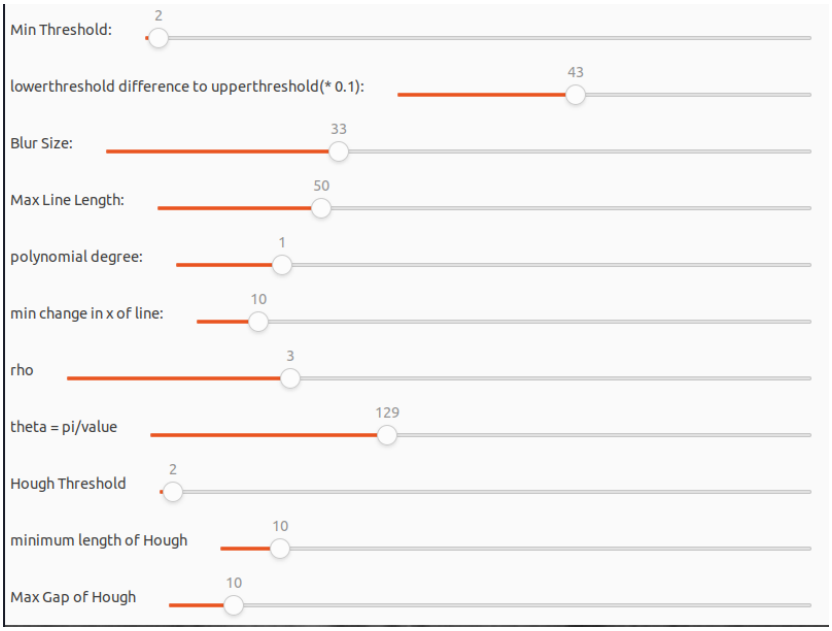


Figure 7: Tracker bars example

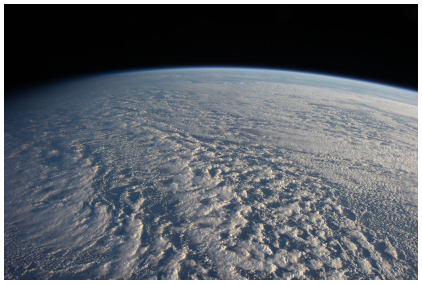
Pipeline steps are:

1. image converted to greyscale
2. image blurred
3. Apply a Canny filter on the image, leaving us with an image of the edges
4. Apply a probabilistic Hough transformation that will return a list of pairs of Points defining the start and end coordinates for line segments
5. Filter out the short lines, using Pythagoras
6. Filter out the vertical lines by checking whether the x co-ordinates of the segment's endpoints are similar.
7. find a curve that best fits all those points

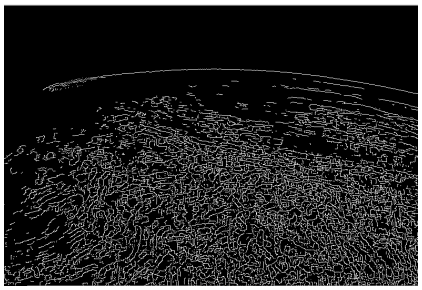
2.2 Processing

Here is the final usage of **horizon.cpp** on the different horizon images.

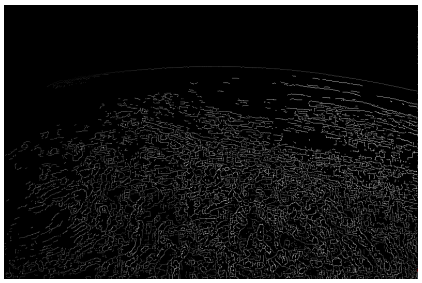
2.2.1 Horizon 1



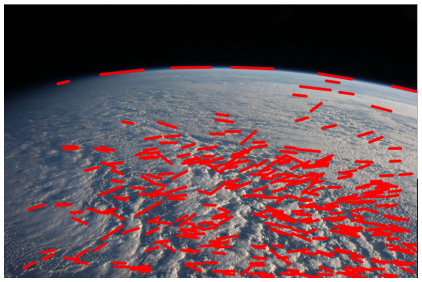
(a) Original Image



(b) Canny filter applied

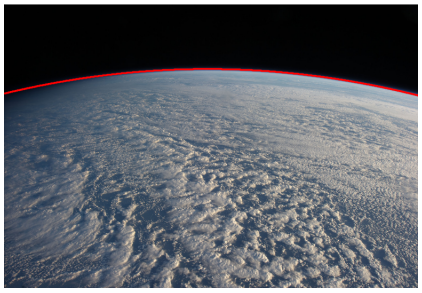


(c) hough applied

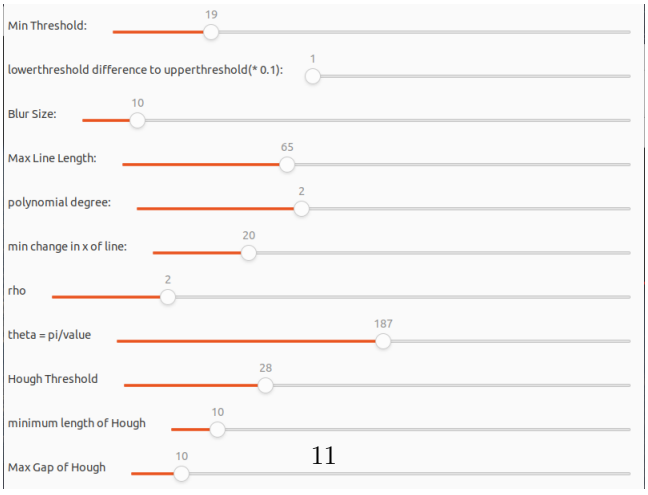


(d) edge lines (including short and "vertical" lines)

(e) edge lines (including "vertical" and excluding short lines)



(f) horizon final line

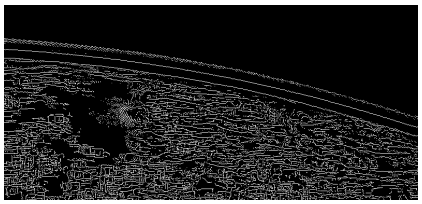


(g) bar values

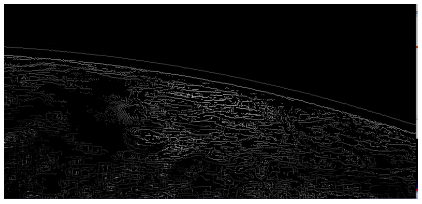
Figure 8: horizon 1

2.2.2 Horizon 2

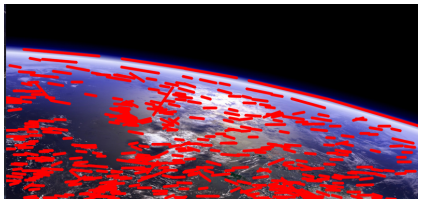
(a) Original Image



(b) Canny filter applied



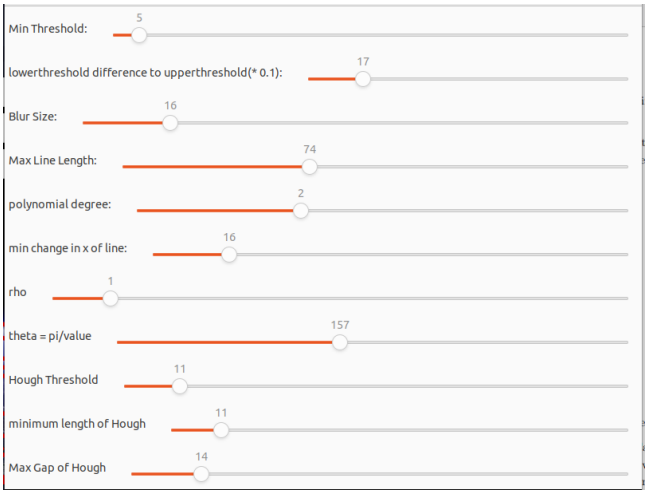
(c) hough applied



(d) edge lines (including short and "vertical" lines)

(e) edge lines (including "vertical" and excluding short lines)

(f) horizon final line

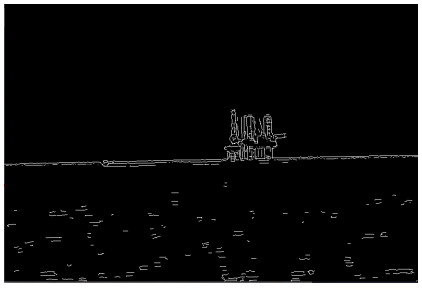


(g) bar values

Figure 9: horizon 2

2.2.3 Horizon 3

(a) Original Image



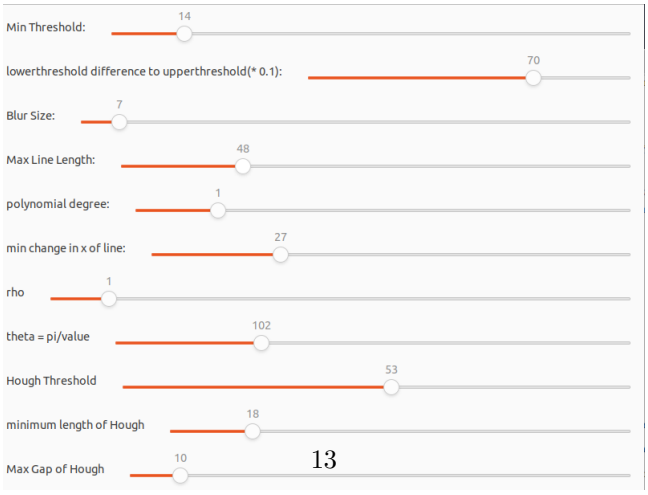
(b) Canny filter applied

(c) hough applied

(d) edge lines (including short and "vertical" lines)

(e) edge lines (including "vertical" and excluding short lines)

(f) horizon final line



(g) bar values

Figure 10: horizon 3

3 Appendix

A histogram.cpp

```
 #include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

using namespace cv;
using namespace std;

const int HIST_IMG_HEIGHT = 400;

// Declare global variables
Mat img; // Original image
Mat out; // Thresholded output image

void createHistogram(Mat &img, Mat &hist)
{
    int histSize = 256; // Number of bins (256 possible
        intensity values for grayscale)
    vector<int> histData(histSize, 0);
    // Create a black canvas for the histogram image
    Mat histImage(HIST_IMG_HEIGHT, 512, CV_8UC3,
        Scalar(255, 255, 255)); // Black canvas to draw
        the histogram

    bool uniform = true, accumulate = false;
    int imgRows = img.rows;
    int imgCols = img.cols;
    int max_height = 0;

    // 1. Iterate through the image to calculate the
        histogram (frequency of pixel intensities)
    for (int i = 0; i < imgRows; i++)
    {
        for (int j = 0; j < imgCols; j++)
        {
            // Get the pixel intensity value (grayscale
                value)
            int pixelValue = img.at<uchar>(i, j);
```

```
            // Increment the corresponding histogram bin
                for this pixel intensity
            histData[pixelValue]++;
            max_height = max(histData[pixelValue],
                max_height);
        }
    }

    // 2. Normalize the histogram
    int hist_w = 512;
    int hist_h = HIST_IMG_HEIGHT;
    int bin_w = cvRound((double)hist_w / histSize); //
        Width of each bin

    for (int i = 0; i < histSize; i++)
    {
        histData[i] = cvRound((float)histData[i] *
            hist_h / max_height); // Normalize based on
            max count
    }

    for (int i = 0; i < histSize; i++)
    {
        int binVal = histData[i]; // already normalized
            value
        // Calculate the x position and width of each bin
        int x = i * bin_w;
        // Draw a filled rectangle for each bin
        rectangle(histImage, Point(x, hist_h - binVal),
            Point(x + bin_w, hist_h),
                  Scalar(25, 25, 25), cv::FILLED);
    }

    // Add vertical lines at intensity 127 and 255
    int lineHeight = hist_h; // The line spans the full
        height of the histogram
    // Create an overlay from histImage
    Mat overlay = histImage.clone();

    // Define line color and thickness
    Scalar lineColor(50, 50, 50); // Dark gray
    int thickness = 1;
    int lineType = LINE_AA; // Anti-aliased line

    // Draw vertical lines at the desired intensities.
    // Note: Since each bin is 2 pixels wide (512/256),
```

```
    // the x position is calculated as intensity * bin_w.
    line(overlay, Point(255 * bin_w, 0), Point(255 *
        bin_w, lineHeight), lineColor, thickness,
        lineType, 0);
    line(overlay, Point(192 * bin_w, 0), Point(192 *
        bin_w, lineHeight), lineColor, thickness,
        lineType, 0);
    line(overlay, Point(128 * bin_w, 0), Point(128 *
        bin_w, lineHeight), lineColor, thickness,
        lineType, 0);
    line(overlay, Point(64 * bin_w, 0), Point(64 *
        bin_w, lineHeight), lineColor, thickness,
        lineType, 0);
    line(overlay, Point(0 * bin_w, 0), Point(0 * bin_w,
        lineHeight), lineColor, thickness, lineType, 0);

    // Blend the overlay with the original histogram
        image.
    // alpha controls the transparency of the lines (0.0
        - 1.0)
    double alpha = 0.7; // 50% transparency for the lines
    addWeighted(overlay, alpha, histImage, 1 - alpha, 0,
        histImage);
    // Set the calculated histogram image in the input
        parameter "hist"
    hist = histImage;
}

void thresh_onchange(int val, void *userdata)
{
    // Access the image pointer from userdata
    Mat *img_ptr = (Mat *)userdata;
    // Perform thresholding and store the result in 'out'
    cv::threshold(*img_ptr, out, val - 1, 256,
        cv::THRESH_BINARY);
    // Show the output image
    cv::imshow("Result", out);
}

int main(int argc, char *argv[])
{
    img = imread(argv[1], IMREAD_GRAYSCALE);
    // Set target width
    int targetWidth = 800;

    // Calculate new height to maintain aspect ratio
```

```
    double aspectRatio = static_cast<double>(img.cols) /
        img.rows;
    int newHeight = static_cast<int>(targetWidth /
        aspectRatio);

    // Resize both images
    cv::resize(img, img, cv::Size(targetWidth,
        newHeight));
    cv::imshow("a window", img);

    // Check if the image was successfully loaded
    if (img.empty())
    {
        printf("Failed to load image '%s'\n", argv[1]);
        return -1;
    }

    Mat hist;
    // Create image histogram
    createHistogram(img, hist);

    // Show the histogram in a window
    namedWindow("Histogram", WINDOW_NORMAL);
    imwrite("hist.jpg", hist); // Save the histogram
        image to file
    imshow("Histogram", hist);

    // Create a window for the result and set up a
        trackbar
    namedWindow("Result", WINDOW_NORMAL);

    // Pass the image pointer as the userdata to the
        callback function
    cv::createTrackbar("Threshold", "Result", NULL, 256,
        thresh_onchange, (void *)&img);

    // Call the callback function for the initial
        processing
    thresh_onchange(0, (void *)&img);

    // Wait for a key press before quitting
    waitKey(0);

    return 0;
}
```

B horizon.cpp

```
#include <iostream>
#include <stdio.h>
#include <opencv2/core/core.hpp>
// #include <opencv2/highgui/highgui.hpp>
// #include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <cmath>

// Declare global variables
cv::Mat OG; // original image

cv::Ptr<cv::CLAHE> clahe = cv::createCLAHE();
cv::Mat contrastImage; // constrast image

cv::Mat img; // black-and-white image
cv::Mat out; // Output image
cv::Mat dst, copy_dst, detected_edges;
std::vector<cv::Vec4i> linesP;          // will hold the
    results of the detection
std::vector<cv::Vec4i> filteredLinesP; // Filtered lines
    based on length and orientation
std::vector<cv::Point> filteredPoints;
std::vector<cv::Point> shawtyLines;  // Filtered lines
    based on length and orientation
std::vector<cv::Vec4i> shawtyLinesP; // Filtered lines
    based on length and orientation

int lowThreshold = 0;
const int max_lowThreshold = 100;
const int ratio_thres = 3;
const int kernel_size = 3;
int Max_Line_Len = 50;
int MIN_DX = 10;
int DEGREE = 1;
int xBlur = 25;
int yBlur = 25;
const char *window_name = "Edge Map";
int tilesGridSize = 0;
int clipLimit = 0;
int polyDegree = 1;
```

```
int  thresholdWindow = 1;

int rho;
int theta;
int houghThreshold;
int minLen;
int maxGap;

// Polynomial regression function
std::vector<double> fitPoly(std::vector<cv::Point>
    points, int n)
{
  // Number of points
  int nPoints = points.size();

  // Vectors for allHistogram
  // Vectors for all the points xs and ys
  std::vector<float> xValues = std::vector<float>();
  std::vector<float> yValues = std::vector<float>();

  // Split the points into two vectors for x and y values
  for (int i = 0; i < nPoints; i++)
  {
    xValues.push_back(points[i].x);
    yValues.push_back(points[i].y);
  }

  // Augmented matrix
  double matrixSystem[n + 1][n + 2];
  for (int row = 0; row < n + 1; row++)
  {
    for (int col = 0; col < n + 1; col++)
    {
      matrixSystem[row][col] = 0;
      for (int i = 0; i < nPoints; i++)
        matrixSystem[row][col] += pow(xValues[i], row +
            col);
    }

    matrixSystem[row][n + 1] = 0;
    for (int i = 0; i < nPoints; i++)
      matrixSystem[row][n + 1] += pow(xValues[i], row) *
          yValues[i];
  }
```

```
  // Array that holds all the coefficients
  double coeffVec[n + 2] = {}; // the "= {}" is needed
      in visual studio, but not in Linux

  // Gauss reduction
  for (int i = 0; i <= n - 1; i++)
    for (int k = i + 1; k <= n; k++)
    {
      double t = matrixSystem[k][i] / matrixSystem[i][i];

      for (int j = 0; j <= n + 1; j++)
        matrixSystem[k][j] = matrixSystem[k][j] - t *
            matrixSystem[i][j];
    }

  // Back-substitution
  for (int i = n; i >= 0; i--)
  {
    coeffVec[i] = matrixSystem[i][n + 1];
    for (int j = 0; j <= n + 1; j++)
      if (j != i)
        coeffVec[i] = coeffVec[i] - matrixSystem[i][j] *
            coeffVec[j];

    coeffVec[i] = coeffVec[i] / matrixSystem[i][i];
  }

  // Construct the vector and return it
  std::vector<double> result = std::vector<double>();
  for (int i = 0; i < n + 1; i++)
    result.push_back(coeffVec[i]);
  return result;
}

// Returns the point for the equation determined
// by a vector of coefficents, at a certain x location
cv::Point pointAtX(std::vector<double> coeff, double x)
{
  double y = 0;
  for (int i = 0; i < coeff.size(); i++)
    y += pow(x, i) * coeff[i];
  return cv::Point(x, y);
}


void thresh_onchange(int val, void *userdata)
```

```
{
  // Access the image pointer from userdata
  cv::Mat *img_ptr = (cv::Mat *)userdata;

  // Perform thresholding and store the result in 'out'
  cv::threshold(*img_ptr, out, val - 1, 256,
      cv::THRESH_BINARY);

  // Show the output image
  cv::imshow("Result", out);
}

void threshold_img()
{
  // Create a window for the result and set up a trackbar
  cv::namedWindow("Result", cv::WINDOW_NORMAL);

  // Pass the image pointer as the userdata to the
      callback function
  cv::createTrackbar("Threshold", "Result", NULL, 256,
      thresh_onchange, (void *)&img);

  // Call the callback function for the initial
      processing
  thresh_onchange(0, (void *)&img);

  cv::Mat *img_ptr = (cv::Mat *)(void *)&img;

  // Perform thresholding and store the result in 'out'
  cv::threshold(*img_ptr, out, -1, 256,
      cv::THRESH_BINARY);

  // Show the output image
  cv::imshow("Result", out);
}

// Probabilistic Line Transform
void probabilisticLineTransform()
{
  cv::HoughLinesP(dst, linesP, rho, CV_PI / theta,
      houghThreshold, minLen, maxGap);
  cv::imshow("all probablisticLines", dst);
  std::cout << "Number of lines detected before
      filtering: " << linesP.size() << std::endl;

  filteredLinesP.clear();
```

```
filteredPoints.clear();
shawtyLinesP.clear();
shawtyLines.clear();
// Filter out short lines and near-vertical lines
for (size_t i = 0; i < linesP.size(); i++)
{
  cv::Vec4i l = linesP[i];
  cv::Point p1(l[0], l[1]);
  cv::Point p2(l[2], l[3]);
  int dx = abs(p1.x - p2.x);
  int dy = abs(p1.y - p2.y);
  double length = sqrt(dx * dx + dy * dy);

  // 5. Filter out short lines (using pythagoroues)

  // 6. Filter out vertical lines (eg. dx < 3 )

  // Keep lines longer than 50 pixels and with dx > 5
     (to avoid nearly vertical lines)
  if (dx > MIN_DX)
  {
    shawtyLinesP.push_back(l);
    shawtyLines.push_back(p1);
    shawtyLines.push_back(p2);
  }
  if (length > Max_Line_Len && dx > MIN_DX)
  {
    filteredLinesP.push_back(l);
    filteredPoints.push_back(p1);
    filteredPoints.push_back(p2);
  }
}
std::cout << "Number of lines after filtering:" <<
    filteredLinesP.size() << std::endl;

cv::Mat OG_copy;
// show shawty lines
OG_copy = OG.clone();
for (size_t i = 0; i < shawtyLinesP.size(); i++)
{
  cv::Vec4i l = shawtyLinesP[i];
  cv::line(OG_copy, cv::Point(l[0], l[1]),
      cv::Point(l[2], l[3]), cv::Scalar(0, 0, 255), 3,
      cv::LINE_AA);
}
cv::imshow("Shawty", OG_copy);
```

```
  // The image with only the (approximately) horizontal
      lines
  OG_copy = OG.clone();
  for (size_t i = 0; i < filteredLinesP.size(); i++)
  {
    cv::Vec4i l = filteredLinesP[i];
    cv::line(OG_copy, cv::Point(l[0], l[1]),
        cv::Point(l[2], l[3]), cv::Scalar(0, 0, 255), 3,
        cv::LINE_AA);
  }
  cv::imshow(window_name, OG_copy);
}

void drawPolynomial(cv::Mat &image, const
    std::vector<double> &coeffs)
{
  // We will build a list of points (x, y) along the
      polynomial
  std::vector<cv::Point> polyPoints;

  // Go through each x in [0, image.cols)
  // (You could also do a smaller range if you only want
      part of the image.)
  for (int x = 0; x < image.cols; x++)
  {
    // Evaluate polynomial y = a0 + a1*x + a2*x^2 + ...
    double y = 0.0;
    for (int i = 0; i < (int)coeffs.size(); i++)
    {
      y += std::pow(x, i) * coeffs[i];
    }

    // Convert to int
    int iy = cvRound(y);

    // Make sure the point is within the image bounds
    if (iy >= 0 && iy < image.rows)
    {
      polyPoints.push_back(cv::Point(x, iy));
    }
  }

  // Draw the polynomial by connecting consecutive points
  for (int i = 0; i < (int)polyPoints.size() - 1; i++)
  {
```

```
    cv::line(
        image,
        polyPoints[i],
        polyPoints[i + 1],
        cv::Scalar(0, 0, 255), // BGR color (red)
        2,                     // thickness
        cv::LINE_AA);
  }
}

static void CannyThreshold(int, void *)
{
  thresholdWindow = (thresholdWindow > 30) ?
      thresholdWindow : 30;
  rho = (rho > 0) ? rho : 1;
  theta = (theta>0) ? theta : 180;
  houghThreshold = (houghThreshold > 0) ? houghThreshold
      : 1;
  minLen = (minLen > 10) ? minLen : 10;
  maxGap = (maxGap > 5) ? maxGap : 5;

  // Create a CLAHE object and set clip limit and grid
      size
  if (clipLimit > 0 && tilesGridSize > 0)
  {
    double tempClip = (clipLimit > 0) ? clipLimit : 1;
    clahe->setClipLimit(static_cast<double>(tempClip));
        // IncrprobabilisticLineTransform can adjust the
        grid size

    // Apply CLAHE to the grayscale image
    clahe->apply(img, contrastImage);
    cv::imshow("contrast", contrastImage);
  }
  else
  {
    contrastImage = img.clone();
  }


  int kernelSize = (xBlur > 0) ? xBlur : 1;
  cv::blur(contrastImage, detected_edges,
      cv::Size(kernelSize, kernelSize));
  cv::imshow("blurred", detected_edges);

  cv::Canny(detected_edges, detected_edges,
```

```
        lowThreshold,
        lowThreshold+static_cast<int>(thresholdWindow *
        0.1), kernel_size);
    dst = cv::Scalar::all(0);
    img.copyTo(dst, detected_edges);

    // 4. prob Hough transformation, will return a list of
        pairs of points
    // defining the start and end coors for line segments

    probabilisticLineTransform();
    // 7. find curve of best fit with appropriate order
        that fits all points
    DEGREE = (DEGREE > 0) ? DEGREE : 1;
    std::vector<double> coeffs = fitPoly(filteredPoints,
        DEGREE);

    for (auto x : coeffs)
    {
      std::cout << x << std::endl;
    }

    cv::Mat imgCopy = OG.clone();
    drawPolynomial(imgCopy, coeffs);
    cv::imshow("Polynomial Example", imgCopy);
}

void pipeline()
{
    dst.create(img.size(), img.type());
    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    cv::createTrackbar("Min Threshold:", window_name,
        &lowThreshold, max_lowThreshold, CannyThreshold);
    cv::createTrackbar("lowerthreshold difference to
        upperthreshold(* 0.1):", window_name,
        &thresholdWindow, 100, CannyThreshold);
    cv::createTrackbar("Blur Size:", window_name, &xBlur,
        100, CannyThreshold);
    cv::createTrackbar("Max Line Length:", window_name,
        &Max_Line_Len, 200, CannyThreshold);
    //cv::createTrackbar("Constrast grid size:",
        window_name, &tilesGridSize, 50, CannyThreshold);
    //cv::createTrackbar("Constrast clip limit:",
        window_name, &clipLimit, 50, CannyThreshold);
    cv::createTrackbar("polynomial degree:", window_name,
        &DEGREE, 6, CannyThreshold);
```

```
  cv::createTrackbar("min change in x of line:",
      window_name, &MIN_DX, 100, CannyThreshold);


  cv::createTrackbar("rho", window_name, &rho, 10,
      CannyThreshold);
  cv::createTrackbar("theta = pi/value", window_name,
      &theta, 360, CannyThreshold);
  cv::createTrackbar("Hough Threshold", window_name,
      &houghThreshold, 100, CannyThreshold);
  cv::createTrackbar("minimum length of Hough",
      window_name, &minLen, 100, CannyThreshold);
  cv::createTrackbar("Max Gap of Hough", window_name,
      &minLen, 100, CannyThreshold);
  // 2. blur
  // 3. Canny Filter -> leaving use with edges of image
  CannyThreshold(0, 0);
}

int main(int argc, char *argv[])
{
  std::string filename(argv[1]);
  DEGREE = 1;

  // 1. convert to greyscale
  img = cv::imread(filename, cv::IMREAD_GRAYSCALE);
  OG = cv::imread(filename);

  // Check if the image was successfully loaded
  if (img.empty())
  {
    printf("Failed to load image '%s'\n", argv[1]);
    return -1;
  }

  // Set target width
  int targetWidth = 800;

  // Calculate new height to maintain aspect ratio
  double aspectRatio = static_cast<double>(OG.cols) /
      OG.rows;
  int newHeight = static_cast<int>(targetWidth /
      aspectRatio);

  // Resize both images
  cv::resize(img, img, cv::Size(targetWidth, newHeight));
```

```
    cv::resize(OG, OG, cv::Size(targetWidth, newHeight));

    cv::imshow("original image", OG);
    pipeline();

    cv::waitKey(0);

    return 0;
}
```