COMP27112
Computer Graphics and Image Processing
Laboratory Exercise 4
Horizon Detection

Dr Tim Morris

2018-2019

1 Introduction

Today's lab exercise is your second (and last) marked lab for the image processing part of the unit. We will be taking a look at detecting the horizon in images, making use of NASA's High Definition Earth Viewing experiment. A number of concepts will be covered, including edge detection using Canny transformations and polynomial regression.

2 Setting up

Create a directory for the exercise, **ex4**, and copy materials to it. In addition to any code you may want to reuse, you'll need the images from Blackboard.

3 Let's dive right in

First and foremost, you need to load an image. For each image, you will have to go through a series of transformations to end up with a line separating the Earth from the sky. Let's take a closer look at what you need to do for each frame:

1. Convert the frame into greyscale
2. Apply a Canny filter on the frame, leaving us with an image of the edges
3. Apply a probabilistic Hough transformation that will return a list of pairs of Points defining the start and end coordinates for line segments. Here, you may have to filter out the vertical lines. You could do that by calculating the inverse tangent of each line, finding its angle from the horizontal, or check whether the x co-ordinates of the segment are similar.
4. Now that we're left with all the (nearly) horizontal lines' points, we need to draw a curve that best fits all those points. This is called polynomial regression. It takes some points and calculates the best polynomial of any order that you choose, fitting all the points. Be careful not to overfit the points though, as the horizon curve best matches a quadratic function; choosing a higher order polynomial can give you unstable results, i.e. a very wavy line.

Since polynomial regression is outside the scope of this course, you have been provided with two functions that do this for you. fitPoly returns a vector of coefficients for the polynomial and pointAtX returns the point on the polynomial at a certain x position.

Listing 1: C++ code to compute a polynomial line's coefficients and points on a polynomial line

```

//Polynomial regression function
cv::vector<double> fitPoly(cv::vector<cv::Point> points, int n)
{
  //Number of points
```

```
int nPoints = points.size();

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

//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 cv vector and return it
    cv::vector<double> result = cv::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(cv::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);
}
```

Here are some of the things that you'll need:

1. **cv::vector<type>** is a vector that can hold multiple values of the same type (cf array)
2. **cv::Canny(source, destination, lowerThreshold, upperThreshold)** is a function that applies the Canny transformation on an image
3. **cv::HoughLinesP(sourceMat, destVector, rho, theta, threshold, minLen, minGap)** detects the line segments in the sourceMat and adds them to the vector as **cv::Vec4i** objects, containing the coordinates for the segment (x1, y1, x2, y2). Rho is the distance resolution of the accumulator (in pixels). Theta is the angle resolution of the accumulator (in pixels). The threshold is the accumulator threshold parameter. Only the lines that get enough votes (>threshold) are returned. minLen is the minimum line length, while maxGap is the maximum allowed gap between points.
4. **cv::circle(destination, point, radius, colour)** draws a circle on a cv::Mat. The point parameter is a cv::Point containing the coordinates of the circle and the colour is a cv::Scalar containing the three BGR values (0-255).
5. **cv::moveWindow(name, x, y)** moves a window to the specified x and y coords. Not necessary, but it's easier not having to arrange the windows every time you run the program.

You should experiment with

1. omitting the thresholding step
2. the parameters of the Canny function
3. the parameters of the Hough function

4 Submit

You now have one code file. Zip it and **submit**. You may have saved the result images but you don't have to submit them. When you have your work marked you will either show the saved image(s) or run the programme for the TA.

5 Marking scheme

Correct code for threshold	1 mark
Correct code for Canny	2 marks
Correct code for probabilistic Hough	2 marks
Correct code for curve fitting	1 mark
Parameter optimisation	2 marks
Correct horizon:	2 marks

References

[1] OpenCV website, https://opencv.org/about.html

[2] OpenCV documentation, https://docs.opencv.org/2.4/index.html