Report TP3 Computer Vision

Created by: Phan Mạnh Tùng & Louis Choules

Created at: October 2022

Contents

Table of Figure 1
Noise Reduction 2
Gradient 3
Edge Detection 5
Full Code 12
References 20

Table of Figure

Figure 1. Original Image 2
Figure 2. Pascal Tree 3
Figure 3. Gradient Magnitude Image 5
Figure 4. Image Threshold 10 6
Figure 5. Image Threshold 25 6
Figure 6. Image Threshold 50 6
Figure 7. Image Threshold 100 6
Figure 8. Image Threshold 200 7
Figure 9. Pixel Distribution for Double Thresholding 8
Figure 10. Direction For Double Thresholding 8
Figure 11. Non-Maximum Suppression 9
Figure 12. Double Thresholding 11
Figure 13. Hysteresis Thresholding 12

Original Image

Figure 1. Original Image

Noise Reduction

- Before all the steps for Canny Edge Detection, we apply Gaussian blur to smooth the image in order to get rid of noises. We tested the results without noise reduction, and these are not clean compared to using noise reduction. For this step, we use the below matrix for convolution.

$$Gaussian = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 1 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

This formula is created by using the pascal tree. Where we take [1 4 6 4 1] and multiplied by itself to create the matrix above.

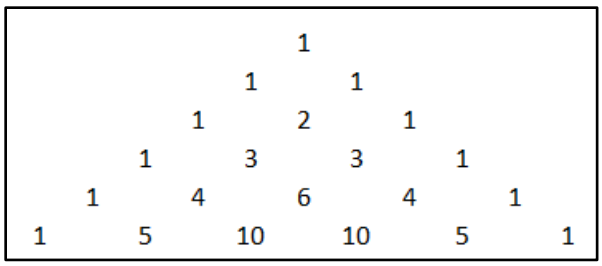


Figure 2. Pascal Tree

Gradient

- Write a program that computes the images I_x and I_y of the gradients in x and y of a PGM image (filtered) using the Scharr operators.

```
void get_scharr_value(int col, int row){
  int filter_size = 3;
  int Gx[] = {3, 0, -3, 10, 0, -10, 3, 0, -3};
  int Gy[] = {3, 10, 3, 0, 0, 0, -3, -10, -3};

  int total_x = 0;
  int total_y = 0;
  int start_col = col - (filter_size/2);
  int start_row = row - (filter_size/2);

  for(int i = 0; i < filter_size; i++){
    for(int j = 0; j < filter_size; j++){
      total_x += get_byte_by_pos(start_col + i, start_row + j) * Gx[i
*filter_size + j];
      total_y += get_byte_by_pos(start_col + i, start_row + j) * Gy[i
*filter_size + j];
    }
  }

  gradient = sqrt(total_x * total_x + total_y * total_y);
  theta[row * cols + cols] = atan((double)(total_y* 1.0f / total_x ))
* 180/ M_PI;
}
```

In this code, we calculate the gradient and the theta for the given pixel. We apply convolution for each pixel using the below Gx, Gy masking matrices, which are the Scharr filters.

$$G_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

After figuring out the derivatives Gx, Gy, we calculate the Gradient of each pixel and the theta with the formula given below.

$$G = \sqrt{{G_x}^2 * {G_y}^2}$$

$$\theta = \arctan\left(G_y / G_x\right) * 180 / \pi$$

After the calculation, the values range from $0 - 255\sqrt{2}$, we decided to normalize the value of the Gradient between $0 - 255$ in order to have a clear display, since the machine cannot display pixels that have value higher than 255. We use the following formula. However, by doing so, we aggressively narrow down the original range of the Gradient, causing many edge intensity values to become much smaller in the new range.

$$G = (G - min) * 255/max$$

- Display the image of the gradient magnitudes.

Figure 3. Gradient Magnitude Image

Edge Detection

- Display pixels with gradient magnitudes above a threshold defined by the user. Comment on the results.

  We try to make several thresholds (10, 25, 50, 100, 200). All pixels above the threshold are edges and vice versa. The best result is with threshold of 50, which is approximately the upper 20% of all the intensity values (this ratio will be used in the double thresholding).

Figure 4. Image Threshold 10

Figure 5. Image Threshold 25

Figure 6. Image Threshold 50

Figure 7. Image Threshold 100

Figure 8. Image Threshold 200

The result of this naïve approach is pretty good at first glance, but some of the edges are too thick and we capture too much details if the threshold is low. To solve these issues, we use Canny's approach with 2 parts: non-maximum suppression and Hysteresis thresholding.

- Canny's approach:

1. Non maximum suppression: extract local gradient extrema in the gradient direction.

The algorithm is to find the maximum intensity pixel in a certain edge direction which is determined using the theta values that we get from the Gradient section. Based on it we will define the value of each pixel by taking the angle from theta.

NORTH WEST	NORTH	NORTH EAST
WEST	CURRENT	EAST
SOUTH WEST	SOUTH	SOUTH EAST

Figure 9. Pixel Distribution for Double Thresholding

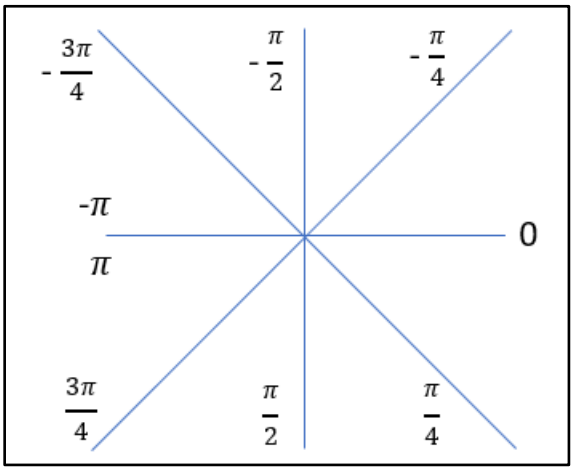


Figure 10. Direction For Double Thresholding

Angle	Clamp Angle	Pixel A	Pixel B
-22.5 – 22.5 157.5 – -157.5	0	West	East
-22.5 – -67.5 112.5 – 157.5	45	South West	North East

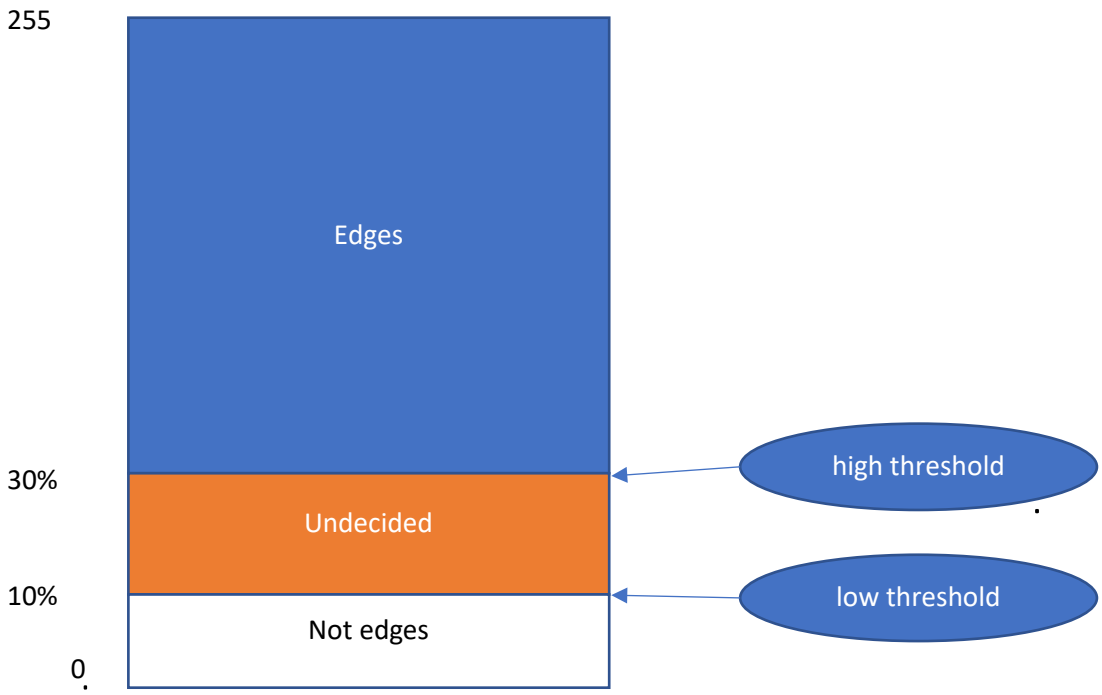
-67.5 – -112.5 67.5 – 112.5	90	South	North
-112.5 – -157.5 22.5 – 67.5	135	South East	North West

When we know which block that we will compare (Current, Pixel A, Pixel B). We check if the current pixel is smaller than Pixel A and Pixel B, then we will change Current into 0.

Figure 11. Non-Maximum Suppression

2. Hysteresis thresholding

Step 1: Double Thresholding



This is a more advanced process than the heuristic approach of naively setting one certain threshold, which results in thick edges and unwanted details. We attempt to create a cleaner result with relevant and thinner edges.

From the results of the gradient magnitudes after normalization, ranging from $0 - 255$, we carry out the double thresholding by setting up 2 thresholds (high and low) in order to divide pixels into 3 different group:

- The "edges" group: The intensity is really high (>=30%) that we are sure they contribute to the final edges. The number 30% is determined using the value 20% from the naïve approach, which contain too many details. We choose 30% in order to only identify the main edges. We then set the intensity to 255 (white color)
- The "undecided" group: The intensity is high, but not enough to make sure they are the edges (>=10% and <30%). We set the intensity to 100, having lighter visibility compared to the final edges.
- The "not edges" group: The intensity is low; thus, these pixels are not the edges. We set the intensity to 0 – black.

Figure 12. Double Thresholding

We can see the result image only contain scattered edges (white) and too much detailed information in the undecided group (gray). We want to complete the edges by turning the undecided pixels into the final edge but get rid of the too-much-detailed pixels.

Step 2: Hysteresis thresholding

The algorithm is simple: if the pixel is in the undecided group and one of 8 surrounded pixels is an edge, turn the pixel into edge. This algorithm assures that only undecided pixels relating to edges can turn into final edges.

The result is shown below, which is the best result we could get after applying both non-maximum suppression and Hysteresis thresholding.

Figure 13. Hysteresis Thresholding

Full Code

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include "Util.h"

int rows, cols;
gray *graymap;
int *filter;
int divider;

/*
  Get_byte_pos
  it will return the value of the pixel based on the position
*/
int get_byte_by_pos(int col, int row){
  if(row < 0)
    row = 0;
  if(col < 0)
    col = 0;
  if(col >= cols)
    col = cols-1;
  if(row >= rows)
    row = rows-1;

  return graymap[row * cols + col];
```

```
}

/*
  Create Filter
  it will create the pascal tree filter and define the value of
divider

  Create Pascal triangle
  Courtesy: https://www.faceprep.in/c/program-to-print-pascals-
triangle/
*/
void create_filter(int filter_size){
  int coef, i, j, temp_pascal[filter_size+2];
  divider = 0;
  for(i = 0; i<= filter_size; i++){
    if(i==0)
      coef = 1;
    else
      coef = coef * (filter_size-i)/i;

    temp_pascal[i] = coef;
  }

  filter = malloc(sizeof(int) * filter_size * filter_size);
  for(i = 0; i < filter_size; i++){
    for(j = 0; j < filter_size; j++){
      int x = temp_pascal[i] * temp_pascal[j];
      filter[i*filter_size+j] = x;
      divider += x;
    }
  }
}

/*
  Get_new_byte
  it will return the value of new pixel based on the filter
automatically.
*/
int get_new_byte(int col, int row, int filter_size){
  int temp = 0;
  int start_col = col - (filter_size/2);
  int start_row = row - (filter_size/2);
  for(int i = 0; i < filter_size; i++){
    for(int j = 0; j < filter_size; j++){
      temp += get_byte_by_pos(start_col + i, start_row + j) *
filter[i *filter_size + j];
    }
  }

  return temp / divider;
}
```

```
/*
  Repeat_gaussian
  It will apply the Gaussian blur with "total_run" repetition and
size from "filter_size"
*/
void repeat_gaussian(int total_runs, int filter_size){
  create_filter(filter_size);
  gray temp_graymap[cols * rows];
  int k;
  int i, j;
  for(k = 0; k < total_runs; k++){
    for(i=0; i < rows; i++){
      for(j=0; j < cols ; j++){
        temp_graymap[i * cols + j] = get_new_byte(j,i, filter_size);
      }
    }

    for(i=0; i < rows * cols; i++){
      graymap[i] = temp_graymap[i];
    }
  }
}

int gradient;
double *theta;

/*
  get_scharr_filter
  It produces the theta and Gradient for the coordinate that passed
in the arguments
*/
void get_scharr_value(int col, int row){
  int filter_size = 3;
  int Gx[] = {3, 0, -3, 10, 0, -10, 3, 0, -3};
  int Gy[] = {3, 10, 3, 0, 0, 0, -3, -10, -3};

  int total_x = 0;
  int total_y = 0;
  int start_col = col - (filter_size/2);
  int start_row = row - (filter_size/2);

  for(int i = 0; i < filter_size; i++){
    for(int j = 0; j < filter_size; j++){
      total_x += get_byte_by_pos(start_col + i, start_row + j) * Gx[i
*filter_size + j];
      total_y += get_byte_by_pos(start_col + i, start_row + j) * Gy[i
*filter_size + j];
    }
  }
```

```
  gradient = sqrt(total_x * total_x + total_y * total_y);
  theta[row * cols + cols] = atan((double)(total_y* 1.0f / total_x ))
* 180/ M_PI;
}

/*
  Scharr_filter
  Apply the scharr filter to the image
*/
void scharr_filter(int threshold){
  int i, j;
  int max = __INT_MAX__ *-1;
  int min = __INT_MAX__;
  int temp[cols * rows];
  theta = (double *) malloc(sizeof(double) * cols * rows);

  for(i=0; i < rows; i++){
    for(j=0; j < cols ; j++){
      get_scharr_value(j,i);
      temp[i*cols + j] = gradient;
      if(gradient > max)
        max = gradient;
      if(gradient < min)
        min = gradient;
    }
  }

  for(i=0; i < rows * cols ; i++){
    int x = (temp[i]-min) * 255 /max;
    graymap[i] = x;

    /*
      For getting the image by using threshold
    */
    // graymap[i] = (x <= threshold) ? 0 : 255;
  }
}

/*
  Non Maxima Suppression
  is for reducing the size of the edge
*/
void non_maxima_suppression(){
  int temp[cols * rows];
  int i, j;

  for(i=0; i < rows * cols; i++){
    temp[i] = graymap[i];
  }

  for(i=1; i < rows-1; i++){
```

```
    for(j=1; j < cols-1; j++){
      int my_pos = temp[i * cols + j];
      double t = theta[i * cols + j];
      if(t < 0)
        t += 180;

      int a = 0;
      int b = 0;

      if((157.5 <= t && t <= 180) || (0 <= t && t < 22.5)){
        a = temp[i * cols + (j+1)];
        b = temp[i * cols + (j-1)];
      }
      if(22.5 <= t && t < 67.5){
        a = temp[(i+1) * cols + (j-1)];
        b = temp[(i-1) * cols + (j+1)];
      }
      if(67.5 <= t && t < 112.5){
        a = temp[(i+1) * cols + j];
        b = temp[(i-1) * cols + j];
      }
      if(112.5 <= t && t < 157.5){
        a = temp[(i-1) * cols + (j-1)];
        b = temp[(i+1) * cols + (j+1)];
      }

      if (my_pos < a || my_pos < b)
        graymap[i * cols + j] = 0;
    }
  }
}

/*
  Define the value of edge, undecided, and not_edge
*/
int edge = 255; //white
int undecided = 100; //white gray
int not_edge = 0;//black: not relate to the edge

/*
  Double Thresholding
  Classify into 3 categories: edge, undecided, and not_edge
*/
void double_thresholding(){
  int i;
  int size = cols*rows;

  // Find max value of the image
  int max = -1 * __INT_MAX__;
  for(i=0; i < size; i++){
    if(max < graymap[i])
```

```
      max = graymap[i];
  }

  int high_threshold = max * 0.3;
  int low_threshold = max * 0.1;

  for(i=0; i<size;i++){
    if(graymap[i] >= high_threshold)
      graymap[i] = edge;
    else if(graymap[i] >= low_threshold)
      graymap[i] = undecided;
    else
      graymap[i] = not_edge;
  };
};

/*
  is_neighbor_an_edge
  it return true if one of the neighbors is an edge
*/
bool is_neighbor_an_edge(int col, int row){
  return
    get_byte_by_pos(col - 1, row -1) == edge || get_byte_by_pos(col -
1, row) == edge || get_byte_by_pos(col - 1, row + 1) == edge ||
    get_byte_by_pos(col, row -1) ==
edge     ||                                          get_byte_by_pos(
col, row + 1) == edge ||
    get_byte_by_pos(col + 1, row -1) == edge || get_byte_by_pos(col +
1, row) == edge || get_byte_by_pos(col + 1, row + 1) == edge;
}

/*
  hysteresis
  Change undecided value into an edge if one of the neighbors is an
edge
*/
void hysteresis(){
  int i,j;

  for(i=0; i < rows; i++){
    for(j=0; j < cols; j++){
      if (graymap[i * cols + j] == undecided){
        if (is_neighbor_an_edge(j, i))
          graymap[i * cols + j] = edge;
        else
          graymap[i * cols + j] = not_edge;
      }
    }
  }
}
```

```
int main(int argc, char* argv[]){
  FILE* ifp;
  int ich1, ich2, maxval=255, pgmraw, filter_size = 5, total_runs=1,
threshold = 50;
  int i;

  /*
    Argument Handler
  */
  if ( argc != 2 ){
    printf("\nUsage: %s file\n\n", argv[0]);
    exit(0);
  }

  ifp = fopen(argv[1],"r");
  if (ifp == NULL) {
    printf("error in opening file %s\n", argv[1]);
    exit(1);
  }

  ich1 = getc( ifp );
  if ( ich1 == EOF )
    pm_erreur( "EOF / read error / magic number" );
  ich2 = getc( ifp );
  if ( ich2 == EOF )
    pm_erreur( "EOF /read error / magic number" );
  if(ich2 != '2' && ich2 != '5')
    pm_erreur(" wrong file type ");
  else
    if(ich2 == '2')
      pgmraw = 0;
    else
      pgmraw = 1;

  cols = pm_getint( ifp );
  rows = pm_getint( ifp );
  maxval = pm_getint( ifp );

  graymap = (gray *) malloc(cols * rows * sizeof(gray));

  for(i=0; i < rows * cols; i++)
    if(pgmraw)
      graymap[i] = pm_getrawbyte(ifp) ;
    else
      graymap[i] = pm_getint(ifp);

  /*
    Processing
    Courtesy: https://towardsdatascience.com/canny-edge-detection-
step-by-step-in-python-computer-vision-b49c3a2d8123
  */
```

```
  // Noise Reduction;
  repeat_gaussian(total_runs, filter_size);

  // Gradient calculation;
  scharr_filter(threshold);

  // Non-maximum suppression;
  non_maxima_suppression();

  // Double threshold;
  double_thresholding();

  // Edge Tracking by Hysteresis.
  hysteresis();

  if(pgmraw)
    printf("P2\n");
  else
    printf("P5\n");

  printf("%d %d \n", cols, rows);
  printf("%d\n",maxval);

  for(i=0; i < rows * cols ; i++){
    if(pgmraw)
      printf("%d ", graymap[i]);
    else
      printf("%c", graymap[i]);
  }

  fclose(ifp);
  return 0;
}
```

References

Program to print Pascal's Triangle | FACE Prep. (2020, March 10). Retrieved from https://www.faceprep.in/c/program-to-print-pascals-triangle/

Sahir, S. (2019, January 25). *Canny Edge Detection Step by Step in Python — Computer Vision*. Retrieved from https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123