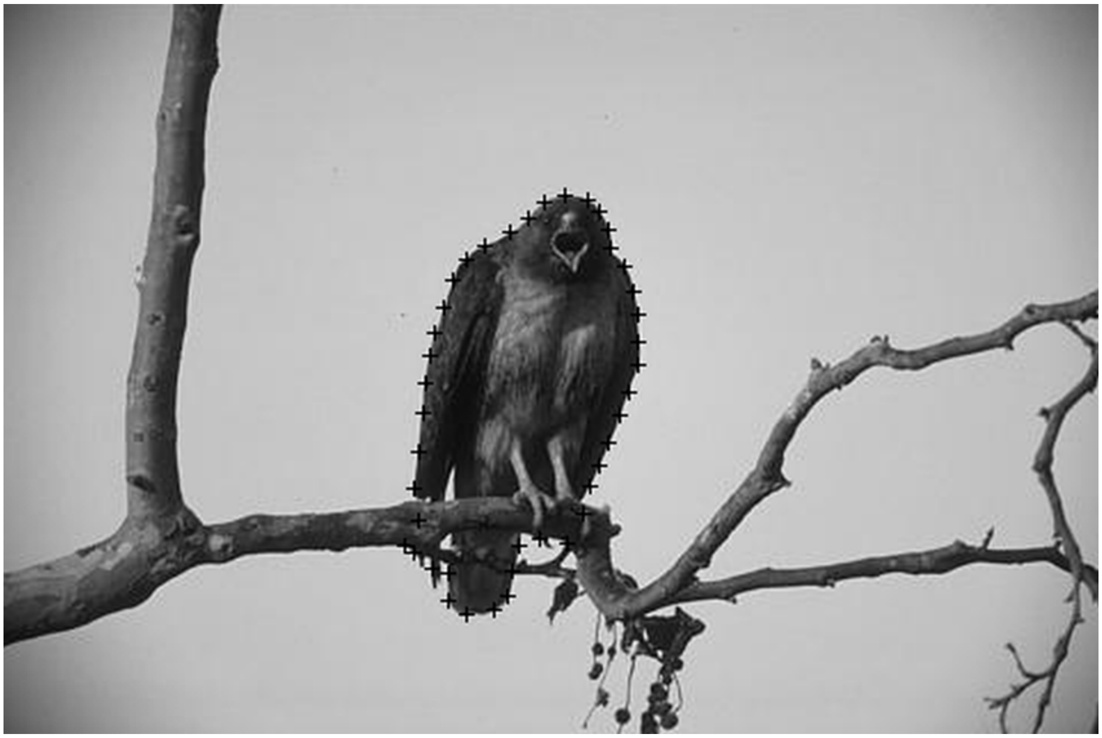
LAB 5 REPORT – Aaron Bruner

The purpose of this lab is to implement an active contours algorithm. After doing simple IO on the initial files we have our contours in an array of structures and the source image. Displaying a plus sign on the initial contour locations gives us the following:

Normalized Sobel edge gradient magnitude image:

Final contour locations after 30 iterations:



Final Coordinates – (Columns, Rows)

278 136	215 265	193 133
278 148	203 267	197 121
278 158	195 261	203 112
274 170	195 249	211 106
270 180	188 247	222 100
265 192	180 240	230 94
261 202	176 237	237 87
257 211	182 226	246 84
254 223	180 212	256 86
247 236	182 196	261 91
235 234	184 179	265 99
226 237	185 166	266 107
223 248	187 154	272 115
221 259	189 144	276 126

Source Code:

```
/* File  : lab5.c
   Author: Aaron Bruner
   Class : ECE - 4310 : Introduction to Computer Vision
   Term  : Fall 2022

   Description: This project must implement the active contour algorithm. The program
                must load a grayscale PPM image and a list of contour points. The contour points
must
                be processed through the active contour algorithm using the options given below.
The
                program must output a copy of the image with the initial contour drawn on top of
it, and a
                second image with the final contour drawn on top of it. The program must also
output a
                list of the final contour pixel coordinates.

   Required Files:
    * hawk.ppm
    * hawk_init.txt

   Bugs:
    * Currently none
*/

#pragma region definitions

//#define DEBUG False
#define BLACK 0
#define WHITE 255
#define ROVERMAX 30  // Number of iterations for our algorithm
#define FILTERCOLS 7 // This is the 7x7 columns count

#define SQR(x) ((x)*(x))

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>

struct contourPoints {
    //char letter;
    int  x; // COLUMN
    int  y; // ROW
};

void sobel(float* output, unsigned char* input, int COLS, int ROWS);
void normalize(unsigned char* output, float* input, int max, int min, int COLS, int ROWS);
void MaxMin(float* srcImage, float *max, float *min, int COLS, int ROWS);
void outputImage(unsigned char* source, char* fileName, int col, int row);
unsigned char* readImage(int* COLS, int* ROWS, char* source);
unsigned char* createImage(int size);
float* normalizeBinary(float* energy);
struct contourPoints* readCSV(char* contourPointsDir, int* fileRows);

char* sourceImageDir = "hawk.ppm";
char* contoursPointsDir = "hawk_init.txt";

#pragma endregion
```

```c
int main(int argc, char* argv[])
{
    unsigned char* sourceImage, *sourceWithContours, *normalizedImage, *result;
    char resultStr[17];
    int* gradientImage;
    float* sobelImage;
    struct contourPoints* contours,* newContours;

    int r, c, j = 0, fileRows, sourceROWS, sourceCOLS, location;
    float min, max;

    /* ──────────────────────────────────────────────────────────────────────────
*/
    /*                  STEP 1: Read in source image and contour pixels
*/
    /*       * User provides no arguments (argc == 1) then we default to specified files
*/
    /*       * User provides 2 arguments  (argc == 3) then we open provided files
*/
    /* ──────────────────────────────────────────────────────────────────────────
*/

    if (argc == 1) {
        sourceImage = readImage(&sourceCOLS, &sourceROWS, sourceImageDir);
        contours = readCSV(contoursPointsDir, &fileRows);
    }
    else if (argc == 3)
    {
        sourceImage = readImage(&sourceCOLS, &sourceROWS, argv[1]);
        contours = readCSV(argv[2], &fileRows);
    }
    else
    {
        printf("Incorrect number of arguments...\nUsage: ./lab5 (sourceImage.ppm)
(ContourPoints.txt)\n");
        exit(0);
    }

    /* ──────────────────────────────────────────────────────────────────────────
*/
    /*           STEP 2: Print plus signs on source image at contour locations
*/
    /* The image with the initial contours drawn as an arrow at each location
    *
    *  _   _   _   X-3 _   _   _
    *  _   _   _   X-2 _   _   _
    *  _   _   _   X-1 _   _   _
    *  Y-3 Y-2 Y-1 YX  Y+1 Y+2 Y+3
    *  _   _   _   X+1 _   _   _
    *  _   _   _   X+2 _   _   _
    *  _   _   _   X+3 _   _   _
    *
    *  Create a copy of the original image */
    /* ──────────────────────────────────────────────────────────────────────────
*/
    sourceWithContours = createImage(sourceCOLS * sourceROWS);
    // Duplicate the source image and then apply contours as plus signs at x,y locations
    for (int i = 0; i < sourceROWS * sourceCOLS; i++) sourceWithContours[i] = sourceImage[i];
    for (int i = 0; i < fileRows; i++)
```

```
    {
        for (j = -3; j <= 3; j++)
        {
            sourceWithContours[(contours[i].y + j) * sourceCOLS + contours[i].x] = BLACK; //
Vertical Line
            sourceWithContours[contours[i].y * sourceCOLS + (contours[i].x + j)] = BLACK; //
Horizontal Line
        }
    }
    outputImage(sourceWithContours, "hawk_sourceArrows.ppm", sourceCOLS, sourceROWS);

    /* ─────────────────────────────────────────────────────────────────────────────────────
*/
    /*                  STEP 3: Get the Sobel edge gradient magnitude image
*/
    /* ─────────────────────────────────────────────────────────────────────────────────────
*/
    sobelImage = calloc(sourceROWS * sourceCOLS, sizeof(float));
    sobel(sobelImage, sourceImage, sourceCOLS, sourceROWS);

    // Find maximum and minimum values
    MaxMin(sobelImage, &max, &min, sourceCOLS, sourceROWS);

    // Normalize the image using min and max values
    normalizedImage = createImage(sourceROWS * sourceCOLS);
    normalize(normalizedImage, sobelImage, max, min, sourceCOLS, sourceROWS);
    outputImage(normalizedImage, "hawk_normalized.ppm", sourceCOLS, sourceROWS);

    /* ─────────────────────────────────────────────────────────────────────────────────────
*/
    /*                   STEP 4: Internal and external energy
*/
    /*  You must experiment with different window sizes and weightings of each energy term, to
*/
    /*  find which gives the best result. Each energy term can be normalized by rescaling from
*/
    /*  min-max value to 0-1, to assist with weighting. The active contour algorithm should run
*/
    /*  for a maximum of 30 iterations, but you should experiment with fewer iterations.
*/
    /* ─────────────────────────────────────────────────────────────────────────────────────
*/

    // Initialize our energy variables
    float* inEnergyOne = calloc(SQR(FILTERCOLS), sizeof(float));
    float* inEnergyTwo = calloc(SQR(FILTERCOLS), sizeof(float));
    float* exEnergy    = calloc(SQR(FILTERCOLS), sizeof(float));
    float* totalEnergy = calloc(SQR(FILTERCOLS), sizeof(float));

    float avgDist = 0, * normOne, * normTwo, * normEx;
    newContours = calloc(fileRows, sizeof(struct contourPoints));
    result = createImage(sourceCOLS*sourceROWS);

    for (int rover = 0; rover < ROVERMAX; rover++, avgDist = 0)
    {
        // Calculate the average distance
        for (int a = 0; a < fileRows; a++)
        {
            a < fileRows - 1 ? newContours[a].x = newContours[a].y = 0 : 0;
```

```
            avgDist += sqrt(SQR(contours[a].y - contours[a == (fileRows - 1) ? 0 : (a + 1)].y) + 
SQR(contours[a].x - contours[a == (fileRows - 1) ? 0 : (a + 1)].x));
            avgDist = (a == (fileRows - 1)) ? avgDist / fileRows : avgDist;
        }

        for (int b = 0; b < fileRows; b++)
        {
            // Calculate the energy for each contour point
            // (y - y+1) * (x - x+1) for every values except the last one. The last value is (y - 
0) * (x - 0)
            for (r = -3; r <= 3; r++)
            {
                for (c = -3; c <= 3; c++)
                {
                    inEnergyOne[(r + 3) * 7 + (c + 3)] = SQR((contours[b].y + r) - contours[b == 
(fileRows - 1) ? 0 : (b + 1)].y) + SQR((contours[b].x + c) - contours[b == (fileRows - 1) ? 0 : (b 
+ 1)].x);
                    inEnergyTwo[(r + 3) * 7 + (c + 3)] = SQR(sqrt(inEnergyOne[(r + 3) * 7 + (c + 
3)]) - avgDist);
                       exEnergy[(r + 3) * 7 + (c + 3)] = SQR(max - sobelImage[(contours[b].y + r) 
* sourceCOLS + (contours[b].x + c)]);
                }
            }

            // Normalize the energy to values from 0 to 1
            normOne = normalizeBinary(inEnergyOne);
            normTwo = normalizeBinary(inEnergyTwo);
            normEx  = normalizeBinary(exEnergy);

            // Get Total Energy and location
            min = location = 0;
            for (int d = 0; d < SQR(FILTERCOLS); d++)
            {
                totalEnergy[d] = 2 * normOne[d] + normTwo[d] + normEx[d];
                (d == 0) ? min = totalEnergy[d] : (totalEnergy[d] < min ? min = totalEnergy[d], 
location = d : false);
            }

            // Now that we have the total energy and location we can find new contour positions
            location / 7 > 3 ? newContours[b].x = contours[b].y + abs(location / 7 - 3) : 
(location / 7 < 3 ? newContours[b].x = contours[b].y - abs(location / 7 - 3) : (newContours[b].x = 
contours[b].y));
            location % 7 > 3 ? newContours[b].y = contours[b].x + abs(location % 7 - 3) : 
(location % 7 < 3 ? newContours[b].y = contours[b].x - abs(location % 7 - 3) : (newContours[b].y = 
contours[b].x));
        }

        // Update the location of the contours
        for (int e = 0; e < fileRows; e++)
        {
            contours[e].x = newContours[e].y;
            contours[e].y = newContours[e].x;
        }


        // Output every 5 iterations
        if (rover % 5 == 0 || rover == 29)
        {
            for (int f = 0; f < sourceROWS * sourceCOLS; f++) result[f] = sourceImage[f];
            for (int g = 0; g < fileRows; g++)
            {
```

```
                for (int h = -3; h <= 3; h++)
                {
                    result[(contours[g].y + h) * sourceCOLS + contours[g].x] = BLACK; // Vertical
Line
                    result[contours[g].y * sourceCOLS + (contours[g].x + h)] = BLACK; //
Horizontal Line
                }
            }

            memset(resultStr, 0, strlen(resultStr));
            sprintf(resultStr, "hawk_final_%d.ppm", rover);
            outputImage(result, resultStr, sourceCOLS, sourceROWS);
        }

    }

    FILE *fpt;
    fpt = fopen("coordinates.txt", "w");
    // Output final coordinates
    for (int j = 0; j < fileRows; j++)
    {
        if (j == 0) fprintf(fpt, "Columns Rows\n");
        fprintf(fpt, "%d %d\n", contours[j].x, contours[j].y);
    }
    fclose(fpt);

    return 0;
}

/// <summary>
/// Normalize the source image to 255 using the maximum and minimum pixel values provided
/// </summary>
/// <param name="output"></param>
/// <param name="input"></param>
/// <param name="max"></param>
/// <param name="min"></param>
/// <param name="COLS"></param>
/// <param name="ROWS"></param>
void normalize(unsigned char* output, float* input, int max, int min, int COLS, int ROWS)
{
    // https://en.wikipedia.org/wiki/Normalization_(image_processing)
    for (int i = 0; i < COLS * ROWS; i++)
    {
        output[i] = (input[i] - min) * 255 / (max - min);
    }

    return;
}

/// <summary>
/// Normalize the image to values from 0 to 1
/// </summary>
/// <param name="energy">Input image to be normalized</param>
/// <returns>Normalized image</returns>
float* normalizeBinary(float * energy)
{
    float* result = calloc(SQR(FILTERCOLS), sizeof(float));
    float min = energy[0], max = energy[0];

    for (int i = 0; i < SQR(FILTERCOLS); i++)
    {
```

```
        max < energy[i] ? max = energy[i] : max;
        min > energy[i] ? min = energy[i] : min;
    }

    for (int i = 0; i < SQR(FILTERCOLS); i++)
    {
        result[i] = (energy[i] - min) * 1 / (max - min);
    }

    return result;
}

/// <summary>
/// Find the maximum and minimum pixel value for the source image
/// </summary>
/// <param name="srcImage"></param>
/// <param name="max"></param>
/// <param name="min"></param>
void MaxMin(float* input, float *max, float *min, int COLS, int ROWS)
{
    (*min) = (*max) = input[0];

    for (int i = 0; i < COLS * ROWS; i++)
    {
        (*max) < input[i] ? (*max) = input[i] : (*max);
        (*min) > input[i] ? (*min) = input[i] : (*min);
    }

    return;
}

/// <summary>
/// Sobel Edge Detection algorithm
/// https://en.wikipedia.org/wiki/Sobel_operator
/// _________________________________________________
/// Horizontal Edge | Vertical Edge | Sobel Template |
///   -1   -2   -1  | -1   0   1    |  w1   w2   w3  |
///    0    0    0  | -2   0   2    |  w4   w5   w6  |
///    1    2    1  | -1   0   1    |  w7   w8   w9  |
/// -------------------------------------------------
/// </summary>
/// <param name="output"></param>
/// <param name="input"></param>
/// <param name="COLS"></param>
/// <param name="ROWS"></param>
void sobel(float* output, unsigned char* input, int COLS, int ROWS)
{
    int horizontalEdge[9] = { -1, -2, -1,  0, 0, 0,  1, 2, 1 };
    int verticalEdge[9]   = { -1,  0,  1, -2, 0, 2, -1, 0, 1 };
    float x = 0, y = 0;

    // Duplicate the input image into the output
    for (int i = 0; i < COLS * ROWS; i++) output[i] = input[i];

    // Apply Sobel Filter
    for (int r = 1; r < ROWS - 1; r++)
    {
        for (int c = 1; c < COLS - 1; c++, x = 0, y = 0)
        {
            for (int a = -1; a <= 1; a++)
            {
```

```c
                for (int b = -1; b <= 1; b++)
                {
                    x += horizontalEdge[(a + 1) * 3 + (b + 1)] * input[(a + r) * COLS + (b + c)];
                    y +=   verticalEdge[(a + 1) * 3 + (b + 1)] * input[(a + r) * COLS + (b + c)];
                }
            }
            output[r * COLS + c] = sqrt(SQR(x) + SQR(y));
        }
    }

    return;
}

/// <summary>
/// Read in integer values from CSV file. Except the delimiter is a space
/// </summary>
/// <param name="contourPointsDir">File directory for CSV file</param>
/// <returns>An array of structures which contain the columns and rows from the file</returns>
struct contourPoints* readCSV(char* contourPointsDir, int *fileRows)
{
    int i = 0, r = 0, c = 0;
    (*fileRows) = 0;
    struct contourPoints* contours;
    FILE* FPT;

    // Open the file for reading
    FPT = fopen(contourPointsDir, "r");
    FPT == NULL ? printf("Failed to open %s.\n", contourPointsDir), exit(0) : false;

    // Determine the number of rows in the file
    while ((i = fscanf(FPT, "%d %d\n", &c, &r)) && !feof(FPT))
        if (i == 2) (*fileRows) += 1;
    // Number of rows + 1 since last row isn't counted
    (*fileRows)++;

    // Allocate space for array of structures
    contours = calloc((*fileRows), sizeof(struct contourPoints));

    // Return to the beginning of the file
    rewind(FPT);

    // Scan in all columns and rows
    for (i = 0; i <= (*fileRows) && !feof(FPT); i++)
        fscanf(FPT, "%d %d\n", &contours[i].x, &contours[i].y);
    fclose(FPT);

    return contours;
}

/// <summary>
/// The readImage function is designed to take a file name as the source and reads all of the data
into a new image.
/// </summary>
/// <param name="ROWS"> Number of rows in the source image </param>
/// <param name="COLS"> Number of columns in the source image </param>
/// <param name="source"> File name that we're needing to open and read data from </param>
/// <returns> The function returns an array of values which makes up our image </returns>
unsigned char* readImage(int* COLS, int* ROWS, char* source)
{
    int BYTES, readHeaderReturn;
    static char header[80];
```

```
    // Open image for reading
    FILE* fpt = fopen(source, "rb");
    if (fpt == NULL) {
        printf("Failed to open file (%s) for reading.\n", source);
        exit(0);
    }

    /* read image header (simple 8-bit greyscale PPM only) */
    if (fscanf(fpt, "%s %d %d %d\n", header, &*COLS, &*ROWS, &BYTES) != 4 || strcmp(header, "P5")
!= 0 || BYTES != 255)
    {
        fclose(fpt);
        printf("Image header corrupted.\n");
        exit(0);
    }

    // Create an empty image that is large enough for ROWS x COLS bytes
    unsigned char* destination = createImage((*ROWS) * (*COLS));

    fread(destination, 1, (*ROWS) * (*COLS), fpt);
    fclose(fpt);

    return destination;
}

/// <summary>
/// Output the image to the fileName provided.
/// </summary>
/// <param name="source">The image needing to be output to the directory fileName</param>
/// <param name="fileName">Directory where the image needs to be printed to</param>
/// <param name="col">Number of columns in source image</param>
/// <param name="row">Number of rows in source image</param>
void outputImage(unsigned char* source, char* fileName, int col, int row)
{

    FILE* FPT = fopen(fileName, "w"); FPT == NULL ? printf("Unable to open %s for writing.\n",
source), exit(0) : false;
    fprintf(FPT, "P5 %d %d 255\n", col, row);
    fwrite(source, col * row, 1, FPT);
    fclose(FPT);

    return;
}

/// <summary>
/// createImage allocates memory for our image array.
/// </summary>
/// <param name="size"> Number of bytes that are needing to be allocated for our image </param>
/// <returns> An array with 'size' number of bytes allocated for our image use</returns>
unsigned char* createImage(int size)
{
    unsigned char* newImage = (unsigned char*)calloc(size, sizeof(unsigned char));
    if (newImage == NULL) {
        printf("Unable to allocate %d bytes of memory.\n", size);
        exit(0);
    }

    return newImage;
}
```