```c
/*
    NAME: RODRIGO IGNACIO ROJAS GARCIA
    LAB#: 5
*/

// LIBRARY SECTION
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

// DEFINE SECTION
#define MAXLENGTH 256
#define MAXITERATION 30
#define SQUARE(x) ((x) * (x))
#define WINDOWSIZE 7

/* READS IN IMAGE */
unsigned char *read_in_image(int rows, int cols, FILE *image_file)
{
    // VARIABLE DECLARATION SECTION
    unsigned char *image;

    image = (unsigned char *)calloc(rows * cols, sizeof(unsigned char));

    fread(image, sizeof(unsigned char), rows * cols, image_file);

    fclose(image_file);

    return image;
}

/* CREATES AND SAVES FILE AS A PPM IMAGE */
void save_image(unsigned char *image, char *file_name, int rows, int cols)
{
    // VARIABLE DECLARATION SECTION
    FILE *file;

    file = fopen(file_name, "w");
    fprintf(file, "P5 %d %d 255\n", cols, rows);
    fwrite(image, rows * cols, sizeof(unsigned char), file);
    fclose(file);
}

/* EXTRACTS INFORMATION FROM INITIAL CONTOUR TEXT FILE*/
void read_initial_countour(char *file_name, int **contour_rows, int **contour_cols,
int *file_size)
{
    // VARIABLE DECLARATION SECTION
    FILE *file;
    int i = 0;
    int cols, rows;
    char c;
    cols = rows = 0;
    *file_size = 0;

    // OBTAINS FILE LENGTH AND REWINDS IT TO BEGINNING
    file = fopen(file_name, "r");
    if (file == NULL)
    {
        printf("Error, could not read in initial contour text file\n");
        exit(1);
```

```c
    }

    while((c = fgetc(file)) != EOF)
    {
        if (c == '\n')
        {
            *file_size += 1;
        }
    }
    rewind(file);

    // ALLOCATES MEMORY
    *contour_rows = calloc(*file_size, sizeof(int *));
    *contour_cols = calloc(*file_size, sizeof(int *));

    // EXTRACTS THE INITIAL COLUMNS AND ROWS OF INITIAL CONTOUR TEXT FILE
    while((fscanf(file, "%d %d\n", &cols, &rows)) != EOF)
    {
        (*contour_rows)[i] = rows;
        (*contour_cols)[i] = cols;
        i++;
    }

    fclose(file);
}

/* OUTPUTS INITIAL HAWK IMAGE WITH THE CONTOURS  */
void draw_contour(unsigned char *image, int image_rows, int image_cols, int
**contour_rows, int **contour_cols, int arr_length, char *file_name)
{
    // VARIABLE DECLARATION SECTION
    unsigned char *output_image;
    int rows, cols;
    int i = 0;

    output_image = (unsigned char *)calloc(image_rows * image_cols, sizeof(unsigned
char));

    // COPIES ORIGINAL IMAGE TO OUTPUT IMAGE
    for (i = 0; i < (image_rows * image_cols); i++)
    {
        output_image[i] = image[i];
    }

    // DRAW "+" ON IMAGE
    for (i = 0; i < arr_length; i++)
    {
        rows = (*contour_rows)[i];
        cols = (*contour_cols)[i];

        // "|" ON COLS
        output_image[(rows - 3)*image_cols + cols] = 0;
        output_image[(rows - 2)*image_cols + cols] = 0;
        output_image[(rows - 1)*image_cols + cols] = 0;
        output_image[(rows - 0)*image_cols + cols] = 0;
        output_image[(rows + 1)*image_cols + cols] = 0;
        output_image[(rows + 2)*image_cols + cols] = 0;
        output_image[(rows + 3)*image_cols + cols] = 0;

        // "-" ON ROWS
        output_image[(rows * image_cols) + (cols - 3)] = 0;
        output_image[(rows * image_cols) + (cols - 2)] = 0;
```

```
        output_image[(rows * image_cols) + (cols - 1)] = 0;
        output_image[(rows * image_cols) + (cols - 0)] = 0;
        output_image[(rows * image_cols) + (cols + 1)] = 0;
        output_image[(rows * image_cols) + (cols + 2)] = 0;
        output_image[(rows * image_cols) + (cols + 3)] = 0;
    }

    // SAVES IMAGE WITH CONTOUR POINTS EXPRESSED AS "+"
    save_image(output_image, file_name, image_rows, image_cols);

    free(output_image);
}

/* CALCULATES THE MINIMUM AND MAXIMUM VALUE IN EACH PIXEL */
void find_min_and_max_int(int *convolution_image, int image_rows, int image_cols, int
*min, int *max)
{
    // VARIABLE DECLARATION SECTION
    int i;

    *min = convolution_image[0];
    *max = convolution_image[0];
    for (i = 1; i < (image_rows * image_cols); i++)
    {
        if (*min > convolution_image[i])
        {
            *min = convolution_image[i];
        }
        if (*max < convolution_image[i])
        {
            *max = convolution_image[i];
        }
    }
}

/* CALCULATES THE MINIMUM AND MAXIMUM VALUE OF EACH PIXEL ON WINDOW */
void find_min_and_max_float(float *convolution_image, int image_rows, int image_cols,
float *min, float *max)
{
    // VARIABLE DECLARATION SECTION
    int i, j, k;

    *min = convolution_image[0];
    *max = convolution_image[0];

    for(i = 1; i < (image_rows-1); i++)
    {
        for(j = 1; j < (image_cols-1); j++)
        {
            k = (i * image_cols) + j;
            if (*min > convolution_image[k])
            {
                *min = convolution_image[k];
            }
            if (*max < convolution_image[k])
            {
                *max = convolution_image[k];
            }
        }
    }
}
```

```
/* NORMALIZE UNSIGNED CHAR INPUT IMAGE, RETURNS NORMALIZED IMAGE */
unsigned char *normalize_unsigned_char(int *convolution_image, int image_rows, int
image_cols, int new_min, int new_max, int min, int max)
{
    // Variable Declaration Section
    unsigned char *normalized_image;
    int i;

    // Allocate memory
    normalized_image = (unsigned char *)calloc(image_rows * image_cols,
sizeof(unsigned char));

    for (i = 0; i < (image_rows * image_cols); i++)
    {
        if (min == 0 && max == 0)
        {
            normalized_image[i] = 0;
        }
        else
        {
            normalized_image[i] = ((convolution_image[i] - min)*(new_max - new_min)/
(max-min)) + new_min;
        }
    }

    return normalized_image;
}

/*  NORMALIZE FLOAT INPUT IMAGE, RETURNS NORMALIZED IMAGE */
float *normalize_float(float *convolution_image, int image_rows, int image_cols, float
new_min, float new_max, float min, float max)
{
    float *normalized_image;
    int i;

    normalized_image = (float *)calloc(image_rows * image_cols, sizeof(float));

    for (i = 0; i < (image_rows * image_cols); i++)
    {
        normalized_image[i] = ((convolution_image[i] - min)*(new_max - new_min)/(max-
min)) + new_min;
    }

    return normalized_image;
}

/* OUTPUTS NORMALIZED SOBEL IMAGE AND RETURNS UN-NORMALIZED SOBEL IMAGE */
float *sobel_edge_detector(unsigned char *image, int image_rows, int image_cols)
{
    // VARIABLE DECLARATION SECTION
    int *convolution_image;
    float *sobel_image;
    unsigned char *normalized_image;
    int i, j, rows, cols;
    int index1 = 0;
    int index2 = 0;
    int x = 0;
    int y = 0;
    int min = 0;
    int max = 0;

    // X and Y KERNELS
```

```
    int g_x[9] =    {-1, 0, 1,
                    -2, 0, 2,
                    -1, 0, 1};

    int g_y[9] =    {-1, -2, -1
                    , 0, 0, 0,
                    1, 2, 1};

    // ALLOCATE MEMORY
    convolution_image = (int *)calloc(image_rows * image_cols, sizeof(int));
    sobel_image = (float *)calloc(image_rows * image_cols, sizeof(float));

    // COPY ORIGINAL IMAGE
    for (i = 0; i < (image_rows * image_cols); i++)
    {
        convolution_image[i] = image[i];
    }

    // CONVOLUTE INPUT IMAGE WITH X AND Y KERNELS
    for (rows = 1; rows < (image_rows - 1); rows++)
    {
        for (cols = 1; cols < (image_cols - 1); cols++)
        {
            x = 0;
            y = 0;
            for (i = -1; i < 2; i++)
            {
                for (j = -1; j < 2; j++)
                {
                    index1 = (image_cols * (rows + i)) + (cols + j);
                    index2 = 3*(i + 1) + (j + 1);
                    x += (image[index1] * g_x[index2]);
                    y += (image[index1] * g_y[index2]);
                }
            }
            index1 = (image_cols * rows) + cols;
            convolution_image[index1] = sqrt((SQUARE(x) + SQUARE(y)));
            sobel_image[index1] = sqrt((SQUARE(x) + SQUARE(y)));
        }
    }

    // FIND MINIMUM AND MAXIMUM VALUES IN CONVOLUTED IMAGE
    find_min_and_max_int(convolution_image, image_rows, image_cols, &min, &max);

    // NORMALIZES CONVOLUTED IMAGE FROM RANGE OF 0-255 IN ORDER TO SAVE AS PPM
    normalized_image = normalize_unsigned_char(convolution_image, image_rows,
image_cols, 0, 255, min, max);
    save_image(normalized_image, "hawk_sobel_image.ppm", image_rows, image_cols);

    // INVERTS NORMALIZED IMAGE IN ORDER TO SAVE AS PPM
    for (i = 0; i < (image_rows * image_cols); i++)
    {
        normalized_image[i] = 255 - normalized_image[i];
    }
    save_image(normalized_image, "hawk_sobel_inverted_image.ppm", image_rows,
image_cols);

    // FREE ALLOCATED MEMORY
    free(normalized_image);
    free(convolution_image);

    return sobel_image;
```

```
}

/* ACTIVE CONTOUR ALGORITHM APPLIED TO ORIGINAL IMAGE */
void active_contour(unsigned char *image, float *sobel_image, int image_rows, int
image_cols, int **contour_rows, int **contour_cols, int arr_length)
{
    // VARIABLE DECLARATION SECTION
    float *inverted_sobel;
    float *first_internal_energy;
    float *second_internal_energy;
    float *external_energy;
    float *sum_energies;
    float min, max, new_min, new_max;
    float *first_internal_energy_normalized, *second_internal_energy_normalized,
*external_energy_normalized;
    float average_distance_x = 0;
    float average_distance_y = 0;
    float average_distance = 0;
    int i, j, k, l, rows, cols;
    int index = 0;
    int index2 = 0;
    int index3 = 0;
    int new_x[arr_length];
    int new_y[arr_length];
    int temp = 0;
    new_min = 0.0;
    new_max = 1.0;


    // ALLOCATE MEMORY
    first_internal_energy = (float *)calloc(49, sizeof(float));
    second_internal_energy = (float *)calloc(49, sizeof(float));
    external_energy = (float *)calloc(49, sizeof(float));
    sum_energies = (float *)calloc(49, sizeof(float));
    inverted_sobel = (float *)calloc(image_rows * image_cols, sizeof(float));

    // FIND MINIMUM AND MAXIMUM VALUE OF SOBEL IMAGE
    find_min_and_max_float(sobel_image, image_rows, image_cols, &min, &max);

    // Creates an invereted Sobel image for External Energy calculation
    for ( i = 0; i < (image_rows * image_cols); i++)
    {
        inverted_sobel[i] = sobel_image[i];
        inverted_sobel[i] = (float)max - inverted_sobel[i];
    }

    // Calculates first Internal Energy
    for (l = 0; l < MAXITERATION; l++)
    {

        average_distance_x = 0.0;
        average_distance_y = 0.0;
        average_distance = 0.0;

        // CALCULATES THE AVERAGE DISTANCE BETWEEN CONTOUR POINTS
        for (i = 0; i < arr_length; i++)
        {
            if ((i + 1) < arr_length)
            {
                average_distance_x = SQUARE((*contour_cols)[i] - (*contour_cols)[i +
1]);
                average_distance_y = SQUARE((*contour_rows)[i] - (*contour_rows)[i +
```

```
1]);
            }
            else
            {
                average_distance_x = SQUARE((*contour_cols)[i] - (*contour_cols)[0]);
                average_distance_y = SQUARE((*contour_rows)[i] - (*contour_rows)[0]);
            }
            average_distance +=sqrt(average_distance_x + average_distance_y);
            new_x[i] = 0;
            new_y[i] = 0;
        }
        average_distance /= arr_length;

        for (i = 0; i < arr_length; i++)
        {
            rows = (*contour_rows)[i];
            cols = (*contour_cols)[i];
            index = 0;

            // FIRST AND SECOND INTERNAL ENERGY AND EXTERNAL ENERGY CALCULATED
            for (j = (rows - 3); j <= (rows + 3); j++)
            {
                for (k = (cols - 3); k <= (cols + 3); k++)
                {
                    if ((i + 1) < arr_length)
                    {
                        first_internal_energy[index] = SQUARE(k - (*contour_cols)[i +
1]) + SQUARE(j - (*contour_rows)[i + 1]);
                        second_internal_energy[index] =
SQUARE(sqrt(first_internal_energy[index]) - average_distance);
                        index2 = (j * image_cols) + k;
                        external_energy[index] = SQUARE(inverted_sobel[index2]);
                    }
                    else
                    {
                        first_internal_energy[index] = SQUARE(k - (*contour_cols)[0])
+ SQUARE(j - (*contour_rows)[0]);
                        second_internal_energy[index] =
SQUARE(sqrt(first_internal_energy[index]) - average_distance);
                        index2 = (j * image_cols) + k;
                        external_energy[index] = SQUARE(inverted_sobel[index2]);

                    }
                    index++;
                }
            }

            // FINDS MINIMUM AND MAXIMUM VALUES OF EACH ENERGY AND NORMALIZES TO
VALUE OF 0 AND 1
            find_min_and_max_float(first_internal_energy, WINDOWSIZE, WINDOWSIZE,
&min, &max);
            first_internal_energy_normalized = normalize_float(first_internal_energy,
WINDOWSIZE, WINDOWSIZE, new_min, new_max, min, max);
            find_min_and_max_float(second_internal_energy, WINDOWSIZE, WINDOWSIZE,
&min, &max);
            second_internal_energy_normalized =
normalize_float(second_internal_energy, WINDOWSIZE, WINDOWSIZE, new_min, new_max,
min, max);
            find_min_and_max_float(external_energy, WINDOWSIZE, WINDOWSIZE, &min,
&max);
            external_energy_normalized = normalize_float(external_energy, WINDOWSIZE,
WINDOWSIZE, new_min, new_max, min, max);
```

```
            // CALCULATES THE ENERGY
            for (j = 0; j < 49; j++)
            {
                sum_energies[j] = first_internal_energy_normalized[j] +
second_internal_energy_normalized[j] + external_energy_normalized[j];
            }

            // FREES MEMORY
            free(first_internal_energy_normalized);
            free(second_internal_energy_normalized);
            free(external_energy_normalized);

            // DETERMINES THE LOWEST VALUE FOR NEW POINTS
            min = sum_energies[0];
            index = 0;
            for (j = 0; j < 49; j++)
            {
                if (min > sum_energies[j])
                {
                    min = sum_energies[j];
                    index = j;
                }
            }

            // DETERMINES ROW AND COLUMN FOR NEW POINT BASED ON INDEX
            temp = 0;
            index2 = (index / WINDOWSIZE); // row
            if (index2 < 3)
            {
                temp = (*contour_rows)[i] - abs(index2 - 3);
                new_y[i] = temp;
            }
            else if (index2 > 3)
            {
                temp = (*contour_rows)[i] + abs(index2 - 3);
                new_y[i]  = temp;
            }
            else
            {
                new_y[i] = (*contour_rows)[i];
            }


            index3 = (index % WINDOWSIZE); // col
            if (index3 < 3)
            {
                new_x[i] = (*contour_cols)[i] - abs(index3 - 3);

            }
            else if (index3 > 3)
            {
                new_x[i] = (*contour_cols)[i] + abs(index3 - 3);
            }
            else
            {
                new_x[i] = (*contour_cols)[i];
            }
        }

        // SETS NEW POINTS
        for (i = 0; i < arr_length; i++)
```

```
        {
            (*contour_cols)[i] = new_x[i];
            (*contour_rows)[i] = new_y[i];
        }

    }

    // DRAWS CONTOUR WITH FINAL POINTS
    draw_contour(image, image_rows, image_cols, contour_rows, contour_cols,
arr_length, "hawk_final_contour.ppm");

    // CREATE FILE WITH FINAL CONTOUR POINTS
    FILE *file;
    file = fopen("final_contour_points.csv", "w");
    fprintf(file, "COLS,ROWS\n");
    for(i = 0; i < arr_length; i++)
    {
        fprintf(file, "%d,%d\n", (*contour_cols)[i], (*contour_rows)[i]);
    }
    fclose(file);

    // FREE ALLOCATED MEMORY
    free(first_internal_energy);
    free(second_internal_energy);
    free(external_energy);
    free(sum_energies);
    free(inverted_sobel);
}

int main(int argc, char *argv[])
{
    // VARIABLE DECLARATION SECTION
    FILE *image_file;
    int IMAGE_ROWS, IMAGE_COLS, IMAGE_BYTES;
    char file_header[MAXLENGTH];
    unsigned char *input_image;
    float *sobel_image;
    int *contour_rows, *contour_cols;
    int file_size;

    /* CHECKS THAT USER ENTERED THE CORRECT NUMBER OF PARAMETERS */
    if (argc != 3)
    {
        printf("Usage: ./executable image_file.ppm initial_contour_file.txt");
        exit(1);
    }

    /* OPEN IMAGE */
    image_file = fopen(argv[1], "rb");
    if (image_file == NULL)
    {
        printf("Error, could not read input image\n");
        exit(1);
    }
    fscanf(image_file, "%s %d %d %d\n", file_header, &IMAGE_COLS, &IMAGE_ROWS,
&IMAGE_BYTES);
    if ((strcmp(file_header, "P5") != 0) || (IMAGE_BYTES !=  255))
    {
        printf("Error, not a greyscale 8-bit PPM image\n");
        fclose(image_file);
        exit(1);
    }
```

```
    /* ALLOCATE MEMORY AND READ IN INPUT IMAGE */
    input_image = read_in_image(IMAGE_ROWS, IMAGE_COLS, image_file);

    /* EXTRACT INFORMATION FROM INITIAL CONTOUR TEXT FILE */
    read_initial_countour(argv[2], &contour_rows, &contour_cols, &file_size);

    /* DRAW INITIAL CONTOUR "+" ON INPUT IMAGE */
    draw_contour(input_image, IMAGE_ROWS, IMAGE_COLS, &contour_rows, &contour_cols,
file_size, "hawk_initial_contour.ppm");

    /* UN-NORMALIZED SOBEL IMAGE */
    sobel_image = sobel_edge_detector(input_image, IMAGE_ROWS, IMAGE_COLS);

    /* CONTOUR ALGORITHM */
    active_contour(input_image, sobel_image, IMAGE_ROWS, IMAGE_COLS, &contour_rows,
&contour_cols, file_size);

    /* FREE ALLOCATED MEMORY */
    free(input_image);
    free(sobel_image);
    free(contour_rows);
    free(contour_cols);

    return 0;
}
```