(a) Original

(b) 2x2 Roberts Cross Operator
Execution Time: 0.2704 seconds

(c) 3x3 Sobel Operator
Execution Time: 0.2663 seconds

(d) 3x3 Mean Blur

(e) Mean Blur, Roberts
Execution Time: 0.2729 seconds

(f) Mean Blur, Sobel
Execution Time: 0.2856 seconds

(g) 3x3 Gaussian Blur

(h) Gaussian Blur, Roberts
Execution Time: 0.2753 seconds

(i) Gaussian Blur, Sobel
Execution Time: 0.3292 seconds

Figure 1: Cameraman

(a) Original

(b) 2x2 Roberts Cross Operator
Execution Time: 0.2463 seconds

(c) 3x3 Sobel Operator
Execution Time: 0.2543 seconds

(d) 3x3 Mean Blur

(e) Mean Blur, Roberts
Execution Time: 0.2796 seconds

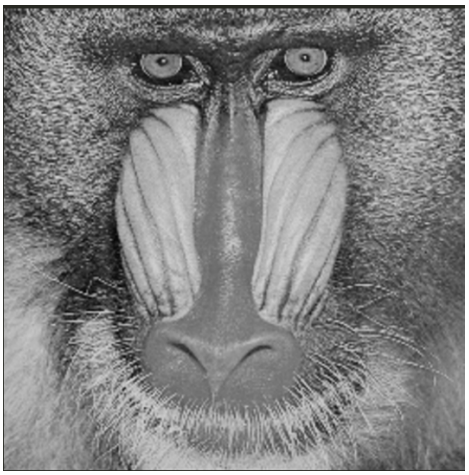
(f) Mean Blur, Sobel
Execution Time: 0.3032 seconds

(g) 3x3 Gaussian Blur

(h) Gaussian Blur, Roberts
Execution Time: 0.2892 seconds

(i) Gaussian Blur, Sobel
Execution Time: 0.2942 seconds

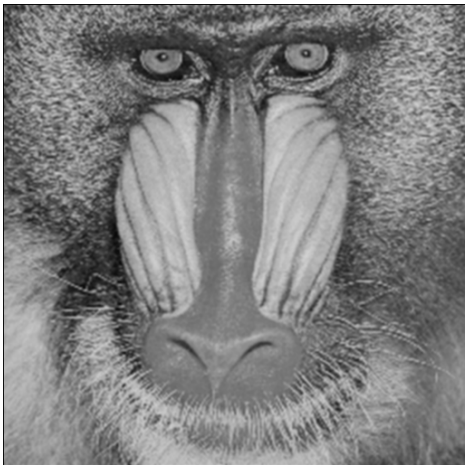
Figure 2: Lena



(a) Original

(b) 2x2 Roberts Cross Operator
Execution Time: 0.2424 seconds

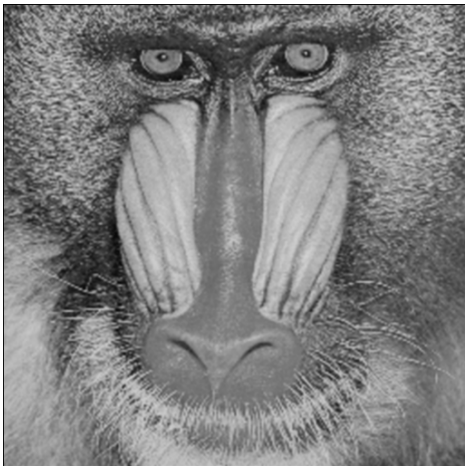
(c) 3x3 Sobel Operator
Execution Time: 0.2733 seconds



(d) 3x3 Mean Blur

(e) Mean Blur, Roberts
Execution Time: 0.3880 seconds

(f) Mean Blur, Sobel
Execution Time: 0.4073 seconds



(g) 3x3 Gaussian Blur

(h) Gaussian Blur, Roberts
Execution Time: 0.3293 seconds

(i) Gaussian Blur, Sobel
Execution Time: 0.3358 seconds

Figure 3: Mandril

(a) Original

(b) 2x2 Roberts Cross Operator
Execution Time: 0.2403 seconds

(c) 3x3 Sobel Operator
Execution Time: 0.2603 seconds

(d) 3x3 Mean Blur

(e) Mean Blur, Roberts
Execution Time: 0.2784 seconds

(f) Mean Blur, Sobel
Execution Time: 0.2994 seconds

(g) 3x3 Gaussian Blur

(h) Gaussian Blur, Roberts
Execution Time: 0.2838 seconds

(i) Gaussian Blur, Sobel
Execution Time: 0.3158 seconds

Figure 4: Pirate

```
/*
    File: edge_detection.h
*/

/* C++ Includes */
#include <chrono>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <utility>

/* Project Includes */
#include "convert/cameraman.h"
#include "convert/lena.h"
#include "convert/mandril.h"
#include "convert/pirate.h"

/* Output Directory */
static const std::string OUTPUT_DIR = "../output/";

/* Input Image Info */
typedef std::pair<std::string, const unsigned char *> Image;
static const std::vector<Image> IMAGES = {
  Image("cameraman", cameraman_image),
  Image("lena", lena_image),
  Image("mandril", mandril_image),
  Image("pirate", pirate_image)
};

/* Input Image Dimensions */
static const int IMAGE_WIDTH = 512;
static const int IMAGE_HEIGHT = 512;
static const int PIXEL_COUNT = IMAGE_WIDTH * IMAGE_HEIGHT;

/* Sobel Operator */
static const int SOBEL_V[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1 }};
static const int SOBEL_H[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

/* Roberts Cross Operators */
static const int ROBERTS_V[2][2] = {{1, 0}, {0, -1}};
static const int ROBERTS_H[2][2] = {{0, 1}, {-1, 0}};

/* Blur Operators */
static const int MEAN_BLUR[3][3] = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
static const int GAUSSIAN_BLUR[3][3] = {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};

/* Helper Functions */
int convolve_order_2(const int[2][2], const int, const int, int **);
int convolve_order_3(const int[3][3], const int, const int, int **);
void write_pgm(const std::string&, const std::string&, int **);

/* Execution Timer Class */
class Timer {
private:
    std::chrono::time_point<std::chrono::system_clock> start;
```

```
  std::string label;
public:
    Timer(const std::string& l) :
    start(std::chrono::system_clock::now()), label(l) {}
    ~Timer() {
        std::chrono::duration<double> elapsed_seconds =
      std::chrono::system_clock::now() - start;
        std::cout << label << " execution time: ";
    std::cout << elapsed_seconds.count() << " s" << std::endl;
    }
};

```

```cpp
/*
  File: main.cpp
*/
#include "edge_detection.h"

int main(void) {  // Start Total Timer
  Timer total_time("Total");

  /* Allocate memory to hold input/output images. */
  int **input_image = new int*[IMAGE_HEIGHT];
  int **blurred_image = new int*[IMAGE_HEIGHT];
  int **output_image = new int*[IMAGE_HEIGHT - 2];
  for (int i = 0; i < IMAGE_HEIGHT; i++) {
    input_image[i] = new int[IMAGE_WIDTH];
    blurred_image[i] = new int[IMAGE_WIDTH];
  }
  for (int i = 0; i < IMAGE_HEIGHT - 2; i++) {
    output_image[i] = new int[IMAGE_WIDTH - 2];
  }

  /* Iterate through each image. */
  for (Image image_pair : IMAGES) {
    /* Unpack image parameters. */
    std::string image_label = image_pair.first;
    const unsigned char *image = image_pair.second;

    /* Read flat 1D image array to 2D array. */
    for (int i = 0; i < IMAGE_HEIGHT; i++) {
      for (int j = 0; j < IMAGE_WIDTH; j++) {
        input_image[i][j] = static_cast<int>(
          image[i * IMAGE_HEIGHT + j]);
      }
    }

    {Timer sobel_time(image_label + "_sobel");

    /* Detect image's edges using the Sobel operator. */
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_3(SOBEL_V, i, j, input_image);
        int H = convolve_order_3(SOBEL_H, i, j, input_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
    }
    write_pgm(image_label, image_label + "_sobel", output_image);
    }  // End Sobel Timer

    {Timer roberts_time(image_label + "_roberts");

    /* Detect image's edges using the Roberts cross. */
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_2(ROBERTS_V, i, j, input_image);
        int H = convolve_order_2(ROBERTS_H, i, j, input_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
```

```
    }
    write_pgm(image_label, image_label + "_roberts", output_image);
    }  // End Roberts Timer

    {Timer mean_sobel_time(image_label + "_mean_sobel");
    /* Blur image with mean blur, then apply Sobel. */
    for (int i = 0; i < IMAGE_HEIGHT - 0; i++) {
      for (int j = 0; j < IMAGE_WIDTH - 0; j++) {
        if (i == 0 || i == IMAGE_HEIGHT - 1 ||  j == 0 || j == IMAGE_WIDTH - 1) {
          blurred_image[i][j] = 0;
        } else {
          blurred_image[i][j] = static_cast<int>(
            round(convolve_order_3(MEAN_BLUR, i, j, input_image) / 9));
        }
      }
    }
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_3(SOBEL_V, i, j, blurred_image);
        int H = convolve_order_3(SOBEL_H, i, j, blurred_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
    }
    write_pgm(image_label, image_label + "_mean_sobel", output_image);
    }  // End Mean Sobel Timer

    {Timer mean_roberts_time(image_label + "_mean_roberts");
    /* Blur image with mean blur, then apply Roberts. */
    for (int i = 0; i < IMAGE_HEIGHT - 0; i++) {
      for (int j = 0; j < IMAGE_WIDTH - 0; j++) {
        if (i == 0 || i == IMAGE_HEIGHT - 1 || j == 0 || j == IMAGE_WIDTH - 1) {
          blurred_image[i][j] = 0;
        } else {
          blurred_image[i][j] = static_cast<int>(
            round(convolve_order_3(MEAN_BLUR, i, j, input_image) / 9));
        }
      }
    }
    write_pgm(image_label, image_label + "_mean_blur", blurred_image);
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_2(ROBERTS_V, i, j, blurred_image);
        int H = convolve_order_2(ROBERTS_H, i, j, blurred_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
    }
    write_pgm(image_label, image_label + "_mean_roberts", output_image);
  }  // End Mean Roberts Timer

    {Timer gaussian_sobel_time(image_label + "_gaussian_sobel");
    /* Blur image with Gaussian blur, then apply Sobel. */
    for (int i = 0; i < IMAGE_HEIGHT - 0; i++) {
      for (int j = 0; j < IMAGE_WIDTH - 0; j++) {
        if (i == 0 || i == IMAGE_HEIGHT - 1 || j == 0 || j == IMAGE_WIDTH - 1) {
          blurred_image[i][j] = 0;
        } else {
```

```
          blurred_image[i][j] = static_cast<int>(
            round(convolve_order_3(GAUSSIAN_BLUR, i, j, input_image) / 16));
        }
      }
    }
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_3(SOBEL_V, i, j, blurred_image);
        int H = convolve_order_3(SOBEL_H, i, j, blurred_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
    }
    write_pgm(image_label, image_label + "_gaussian_sobel", output_image);
    }  // End Gaussian Sobel Timer

    {Timer mean_roberts_time(image_label + "_gaussian_roberts");
    /* Blur image with Gaussian blur, then apply Roberts cross. */
    for (int i = 0; i < IMAGE_HEIGHT - 0; i++) {
      for (int j = 0; j < IMAGE_WIDTH - 0; j++) {
        if (i == 0 || i == IMAGE_HEIGHT - 1 || j == 0 || j == IMAGE_WIDTH - 1) {
          blurred_image[i][j] = 0;
        } else {
          blurred_image[i][j] = static_cast<int>(
            round(convolve_order_3(GAUSSIAN_BLUR, i, j, input_image) / 16));
        }
      }
    }
    write_pgm(image_label, image_label + "_gaussian_blur", blurred_image);
    for (int i = 1; i < IMAGE_HEIGHT - 1; i++) {
      for (int j = 1; j < IMAGE_WIDTH - 1; j++) {
        int V = convolve_order_2(ROBERTS_V, i, j, blurred_image);
        int H = convolve_order_2(ROBERTS_H, i, j, blurred_image);
        output_image[i - 1][j - 1] = static_cast<int>(
          round(sqrt(pow(V, 2) + pow(H, 2))));
      }
    }
    write_pgm(image_label, image_label + "_gaussian_roberts", output_image);
    }  // End Gaussian Roberts Timer
  }

  /* Clean up allocated memory. */
  for (int i = 0; i < IMAGE_HEIGHT; i++) {
    delete[] input_image[i];
    delete[] blurred_image[i];
  }
  for (int i = 0; i < IMAGE_HEIGHT - 2; i++) {
    delete[] output_image[i];
  }
  delete[] input_image;
  delete[] blurred_image;
  delete[] output_image;

  return 0;
}  // End Total Timer

int convolve_order_2(const int mask[2][2], const int i, const int j, int **I) {
  return I[i][j] * mask[0][0]
```

```
    + I[i][j + 1] * mask[0][1]
    + I[i + 1][j] * mask[1][0]
    + I[i + 1][j + 1] * mask[1][1];
}

int convolve_order_3(const int mask[3][3], const int i, const int j, int **I) {
  return I[i - 1][j - 1] * mask[0][0]
    + I[i - 1][j] * mask[0][1]
    + I[i - 1][j + 1] * mask[0][2]
    + I[i][j - 1] * mask[1][0]
    + I[i][j] * mask[1][1]
    + I[i][j + 1] * mask[1][2]
    + I[i + 1][j - 1] * mask[2][0]
    + I[i + 1][j] * mask[2][1]
    + I[i + 1][j + 1] * mask[2][2];
}

void write_pgm(const std::string& label, const std::string& mask, int **W) {
  std::ofstream pgm(OUTPUT_DIR + label + "/" + mask + ".pgm");

  if (!pgm) {
    std::cerr << "Error occurred opening " << label << std::endl;
    return;
  }

  /* Write header, dimensions, and maximum value to file. */
  pgm << "P2" << std::endl;
  pgm << IMAGE_WIDTH - 2 << " " << IMAGE_HEIGHT - 2 << std::endl;
  pgm << "255" << std::endl;

  for (int i = 0; i < IMAGE_HEIGHT - 2; i++) {
    for (int j = 0; j < IMAGE_WIDTH - 2; j++) {
      pgm << W[i][j] << " ";
    }
    pgm << std::endl;
  }
}

```