ITMO University

Faculty of Control Systems and Robotics

Report for laboratory work №3

course "Digital Image Processing"

Filtering and Edges Detection

Done by:

Ha The Long Vuong
Group: R42334c

Supervised by:

Shavetov Sergey Vasilievich

Saint Petersburg
2021

Contents

List of Figures

Listings

1 Purpose

Studying of the filtering images basic methods and edges detection.

2 Assignment

2.1 Noise Types

2.1.1 Impulse Noise

With impulse noise, the signal is distorted by spikes with very large negative or positive values of short duration and can arise, for example, due to decoding errors. This noise results in white ("salt") or black ("pepper") dots in the image, which is why it is often called dot noise. To describe it, one should take into account the fact that the appearance of a noise spike in each pixel $I(x, y)$ does not depend on the quality of the original image or on the presence of noise at other points and has the probability of p, and the value of the pixel intensity I(x,y) will be changed to value $d \in [0, 255]$:

$$I(x,y) = \begin{cases} d, \text{ with probability } p \\ s_{x,y}, \text{ with probability } (1-p) \end{cases} \tag{1}$$

OpenCV code

Listing 1: "Salt" and "pepper" noise using OpenCV and C++ programming language

```
void DIP::lab3::impluse_noise(const cv::Mat& im, cv::Mat& noise_im) {
  cv::Mat saltpepper_noise = cv::Mat::zeros(im.rows, im.cols, CV_8U);
  cv::randu(saltpepper_noise, 0, 255);

  cv::Mat black = saltpepper_noise < 30;
  cv::Mat white = saltpepper_noise > 225;

  noise_im = im.clone();
  noise_im.setTo(255, white);
  noise_im.setTo(0, black);
}
```

Result

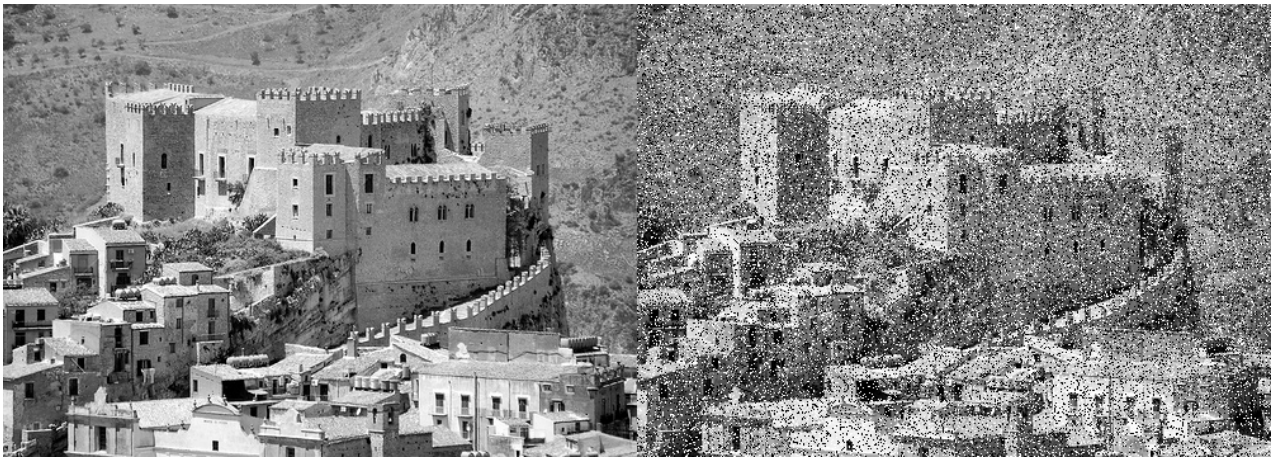


Figure 1: "Salt" and "pepper" noise using OpenCV and C++ programming

2.1.2 Additive Noise

Additive noise is described by the following expression

$$I_{new}(x, y) = I(x, y) + \eta(x, y) \tag{2}$$

where I_{new} - noisy image, I - original image, η - signal-independent additive noise with Gaussian or any other probability density function.

Python code

Listing 2: Additive noise using Python programming language

```
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage import io
from skimage.util import random_noise

filename = 'gray.png'
castel = io.imread(filename)

localvar_img = random_noise(castel,mode='localvar')
plt.imshow(localvar_img, cmap=plt.cm.gray)
plt.axis('off')
plt.show()
```

Result

(a) Original Image

(b) Noise image applied Additive noise

Figure 2: Additive noise using Python programming language

2.1.3 Multiplicative Noise

Multiplicative noise is described by the following expression:

$$I_{new}(x, y) = I(x, y) \cdot \eta(x, y) \tag{3}$$

A special case of multiplicative noise is speckle noise. This noise appears in images captured by coherent imaging devices such as medical scanners or radars.

OpenCV code

Listing 3: Speckle noise using OpenCV and C++ programming language

```
void lab3::speckle_noise(const cv::Mat& im, cv::Mat& noise_im) {
  noise_im = im.clone();
```

```
  int size[3] = {im.rows, im.cols, im.dims};
  cv::Mat mult_noise(im.dims, size, im.type());
  cv::theRNG().fill(mult_noise, cv::RNG::NORMAL, 0, 1);

  cv::multiply(noise_im, mult_noise, noise_im);
}
```

Result

(a) Original Image

(b) Noise image applied speckle noise

Figure 3: Additive noise using Python programming language

2.1.4 Gaussian Noise

The probability density distribution function $p(z)$ of the random variable z is described by the following expression:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-\mu)^2}{2\sigma^2}} \tag{4}$$

where

- z — the intensity of the image (for example, for a grayscale image $z \in [0, 255]$)
- μ — mean (mathematical expectation) of a random variable z,
- σ — standard deviation,
- Variance σ^2 determines the power of the introduced noise.

OpenCV code

Listing 4: Gaussian noise using OpenCV and C++ programming language

```
void lab3::gaussian_noise(const double mean, const double stddev,
                          const cv::Mat& im, cv::Mat& noise_im) {
  int size[3] = {im.rows, im.cols, im.dims};
  cv::Mat gaussian_noise(im.dims, size, im.type());
  noise_im = im.clone();
  cv::randn(gaussian_noise, mean, stddev);
  cv::addWeighted(noise_im, 1.0, gaussian_noise, 1.0, 0.0, noise_im);
}
```

Result

Figure 4: Gaussian noise using OpenCV and C++ programming

2.1.5 Quantization Noise

Depends on the selected quantization step and on the signal. Quantization noise can lead, for example, to the appearance of false contours around objects or to remove low-contrast details in the image. Such noise is not eliminated. Quantization noise can be approximately described by the *Poisson* distribution.

Python code

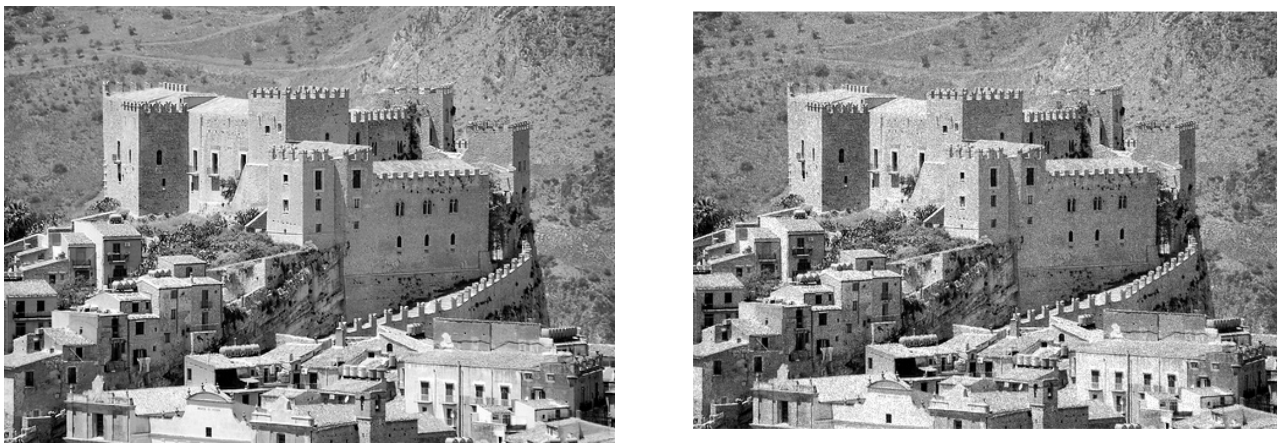
Listing 5: Poisson noise using Python programming language

```
import matplotlib.pyplot as plt
from skimage import io
from skimage.color import rgb2gray
from skimage.util import random_noise

filename = 'gray.png'
castel = io.imread(filename)

poisson_img = random_noise(castel,mode='poisson')
plt.imshow(poisson_img, cmap=plt.cm.gray)
plt.axis('off')
plt.show()
```

Result



(a) Original Image (b) Noise image applied Poisson noise

Figure 5: Poisson noise using Python programming language

2.2 Image Filtering

2.2.1 Low-pass Filter

Gaussian Filter

During images filtering, a two-dimensional Gaussian filter is used:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-y^2}{2\sigma^2}} \tag{5}$$

The larger the parameter σ, the more the image is blurred.

OpenCV code

Listing 6: Gaussian filtering using OpenCV and C++ programming language

```
void lab3::gaussian_filter(cv::Mat& noise_im, cv::Mat& denoise_im) {
  cv::GaussianBlur(noise_im, denoise_im, cv::Size(3, 3), 0);
}
```

Result

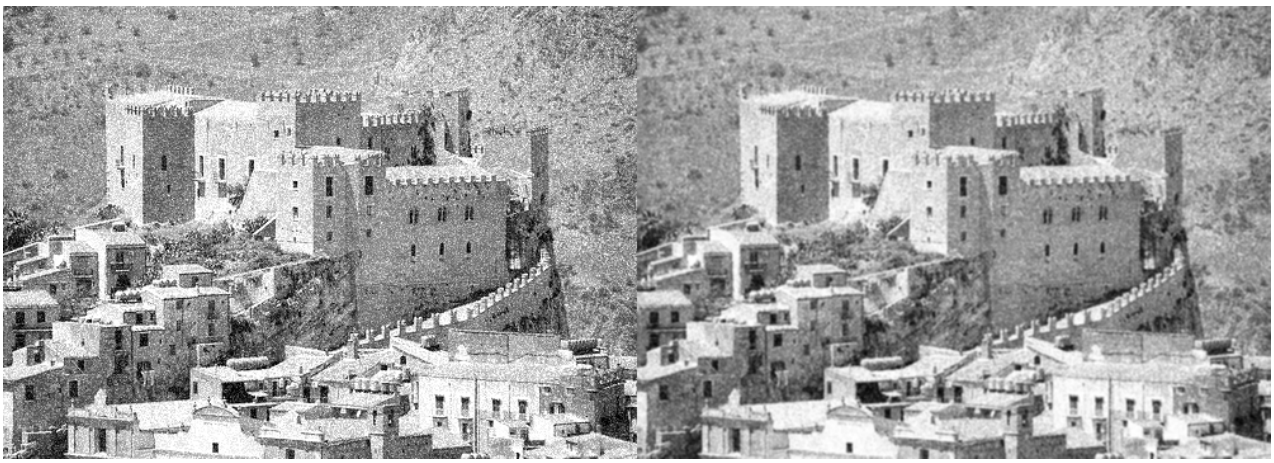


Figure 6: Gaussian denoise using OpenCV and C++ programming

Counter-harmonic Mean Filter

The filter is based on the expression:

$$I_{new}(x,y) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} I(i,j)^{Q+1}}{m \cdot n \cdot \sum_{i=0}^{m}\sum_{j=0}^{n} I(i,j)^{Q}} \tag{6}$$

where Q - filter order.

OpenCV Code

Listing 7: Counter-harmonic mean filtering using OpenCV and C++ programming language

```
void lab3::couterHarmonic_filter(cv::Mat& noise_im, cv::Mat& denoise_im,
                                 int kSize, double Q) {
  cv::Mat _src;
  //  cvtColor(noise_im, denoise_im, CV_BGR2GRAY);
  copyMakeBorder(noise_im, _src, (kSize / 2), (kSize / 2), (kSize / 2),
                 (kSize / 2), cv::BORDER_REPLICATE);

  float val;

  for (int i = (kSize / 2); i < _src.rows - (kSize / 2); i++) {
```

```
    for (int j = (kSize / 2); j < _src.cols - (kSize / 2); j++) {
      float a = 0;
      float b = 0;

      for (int k = i - (kSize / 2); k <= i + (kSize / 2); k++) {
        for (int w = j - (kSize / 2); w <= j + (kSize / 2); w++) {
          val = (float)_src.at<uchar>(k, w);
          a += (float)std::pow(val, (1.f + Q));
          b += (float)std::pow(val, Q);
        }
      }

      denoise_im.at<uchar>(i - (kSize / 2), j - (kSize / 2)) =
          cv::saturate_cast<uchar>(a / b);
    }
  }
}
```

Result

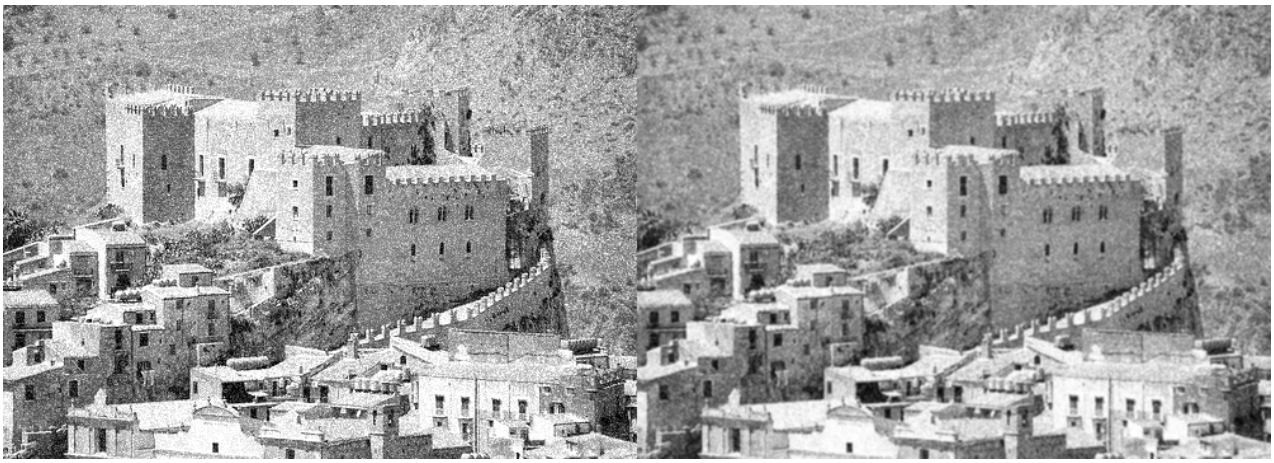


Figure 7: Counter-harmonic mean filtering using OpenCV and C++ programming

2.2.2 Nonlinear Filter

Median Filter

An arbitrary window shape can be set using zero coefficients. The pixel intensities in the window are represented as a column vector and sorted in ascending order. The filtered pixel is assigned the median (mean) intensity value in the series. The median element number after sorting can be calculated by the formula $n = \frac{N+1}{2}$, where N — the number of pixels involved in sorting.

OpenCV Code

Listing 8: Median filtering using OpenCV and C++ programming language

```
void lab3::median_filter(cv::Mat& noise_im, cv::Mat& denoise_im, int kSize)
    {
  medianBlur(noise_im, denoise_im, kSize);
}
```

Result

Weighted Median Filter

In this median filtering modification in the mask, weights are used (numbers 2, 3 etc.) to reflect

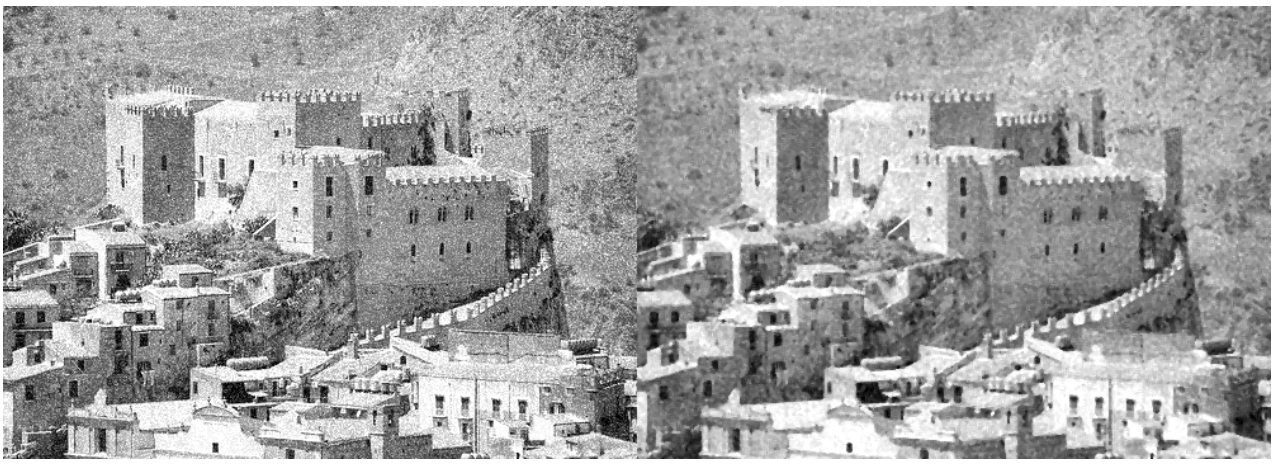


Figure 8: Median filtering using OpenCV and C++ programming

more influence on the filtering result of pixels located closer to the element to be filtered.

OpenCV Code

Listing 9: Weighted median filtering using OpenCV and C++ programming language

```
void lab3::weightedMedian_filter(cv::Mat& noise_im, cv::Mat& denoise_im,
                                 int kSize) {
  cv::ximgproc::weightedMedianFilter(noise_im, noise_im, denoise_im, kSize);
}
```

Result

Figure 9: Weighted median filtering using OpenCV and C++ programming

Adaptive Median Filter

In this filter modification, a sliding window of size s × s adaptively increases depending on the filtering result. The main idea is to increase the window size until the algorithm finds a median value that is not impulse noise, or until it reaches the maximum window size. In the latter case, the algorithm will return the value $z_{i,j}$.

OpenCV Code

Listing 10: An iteration of adaptive median filtering using OpenCV and C++ programming language

```
uchar lab3::adaptive_iterate(const cv::Mat& im, int row, int col,
                             int kernelSize, int maxSize) {
```

```
  std::vector<uchar> pixels;
  for (int a = -kernelSize / 2; a <= kernelSize / 2; a++) {
    for (int b = -kernelSize / 2; b <= kernelSize / 2; b++) {
      pixels.push_back(im.at<uchar>(row + a, col + b));
    }
  }
  sort(pixels.begin(), pixels.end());
  auto min = pixels[0];
  auto max = pixels[kernelSize * kernelSize - 1];
  auto med = pixels[kernelSize * kernelSize / 2];
  auto zxy = im.at<uchar>(row, col);
  if (med > min && med < max) {
    if (zxy > min && zxy < max) {
      return zxy;
    } else {
      return med;
    }
  } else {
    kernelSize += 2;
    if (kernelSize <= maxSize)
      return adaptive_iterate(im, row, col, kernelSize, maxSize);
    else
      return med;
  }
}
```

Listing 11: Adaptive median filtering using OpenCV and C++ programming language

```
void lab3::adaptive_filter(int minSize, int maxSize, cv::Mat& noise_im,
                           cv::Mat& denoise_im) {
  copyMakeBorder(noise_im, denoise_im, maxSize / 2, maxSize / 2, maxSize /
   2,
                 maxSize / 2, cv::BORDER_REFLECT);
  int rows = denoise_im.rows;
  int cols = denoise_im.cols;
  for (int j = maxSize / 2; j < rows - maxSize / 2; j++) {
    for (int i = maxSize / 2; i < cols * denoise_im.channels() - maxSize /
   2;
         i++) {
      denoise_im.at<uchar>(j, i) =
          adaptive_iterate(denoise_im, j, i, minSize, maxSize);
    }
  }
}
```

Result

Rank Filter

Rank filter selects a pixel with the number from the resulting column vector of mask elements $r \in [1, N]$, which will be result of filtering

1. If N is odd and $r = \frac{N+1}{2}$, then rank filter is *median*, the code and result of this filter is present in 2.2.2,
2. If $r = 1$, the filter selects the lowest intensity value and called *min-filter*,
3. If $r = N$, the filter selects the maximum intensity value and called *max-filter*.

OpenCV Code

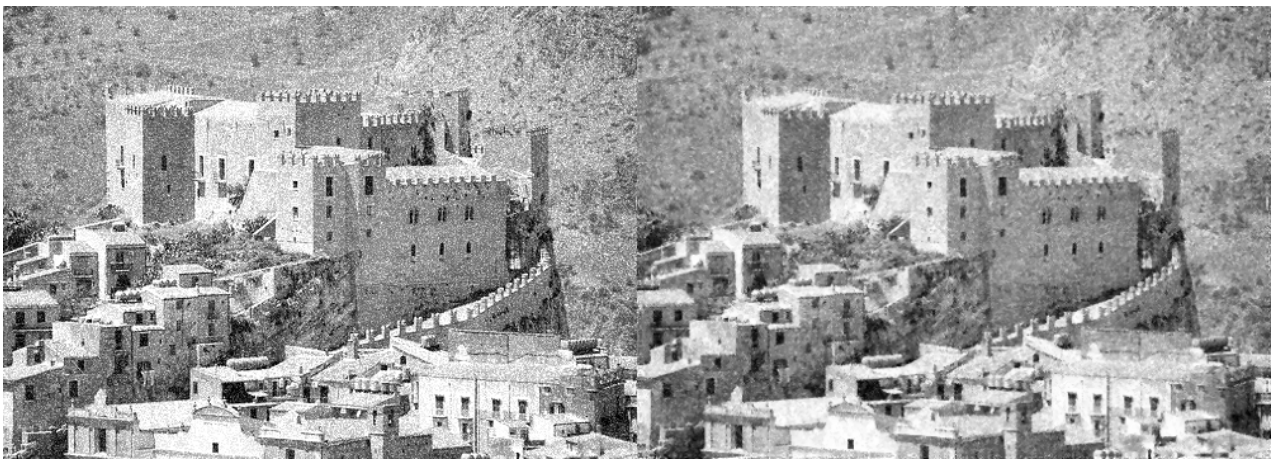


Figure 10: Adaptive median filtering using OpenCV and C++ programming

Listing 12: Rank filtering using OpenCV and C++ programming language

```
void lab3::rank_filter(const std::string& method, int kSize, cv::Mat&
    noise_im,cv::Mat& denoise_im) {
  int iMethod;
  if (method == "min") iMethod = 0;
  if (method == "max") iMethod = 1;
  cv::Mat kernel = cv::getStructuringElement(cv::MorphShapes::MORPH_RECT,
                                             cv::Size(kSize, kSize));
  switch (iMethod) {
    case 0: {
      cv::erode(noise_im, denoise_im, kernel);
      break;
    }
    case 1: {
      cv::dilate(noise_im, denoise_im, kernel);
      break;
    }
    default:
      std::cout << "Default " << std::endl;
  }
}
```

Result

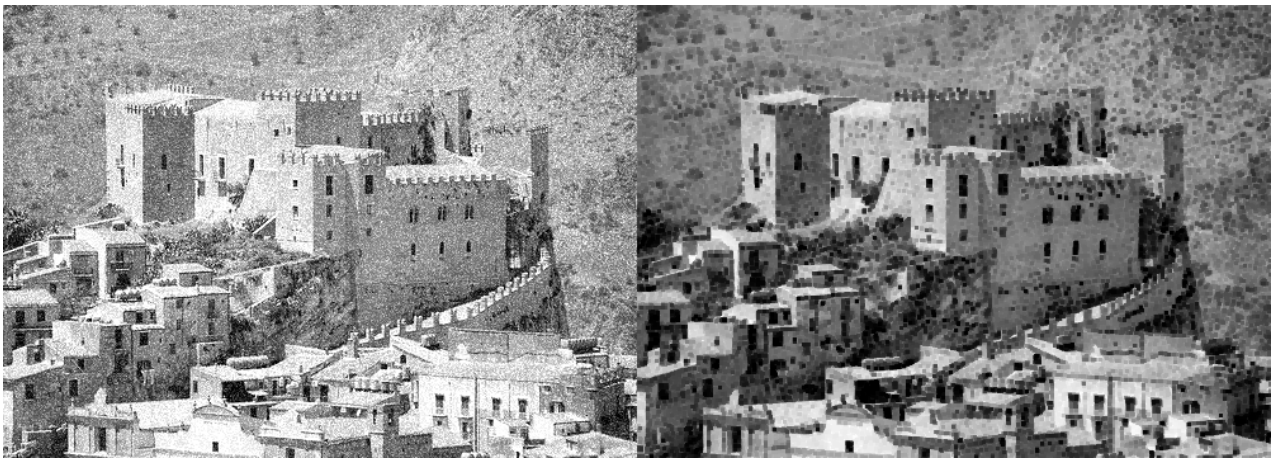


Figure 11: Min filtering using OpenCV and C++ programming

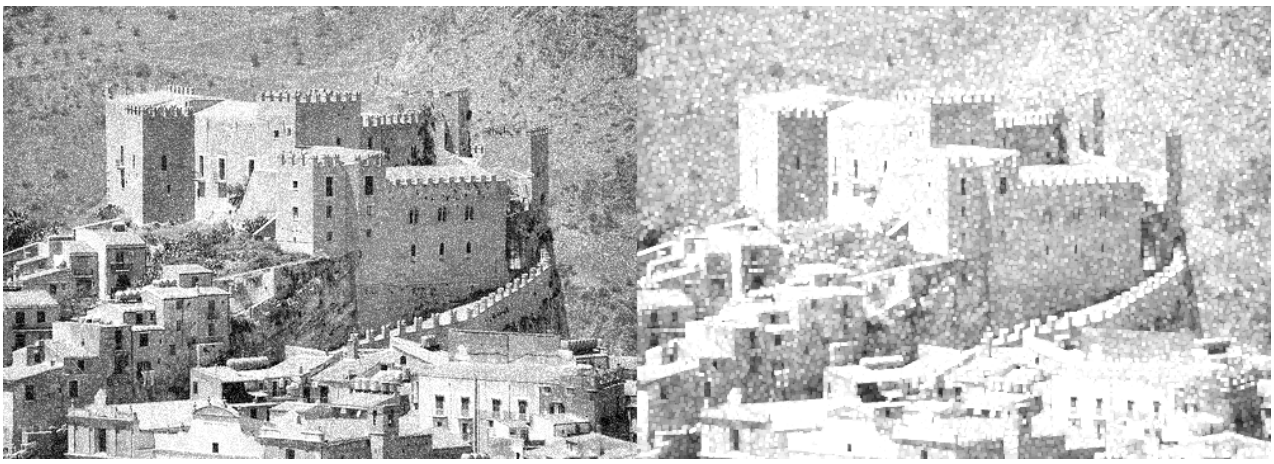


Figure 12: Max filtering using OpenCV and C++ programming

2.3 Edges Detector

2.3.1 Roberts Edges Detector

The Roberts filter works with the minimum dimensionality mask allowed for the derivative calculation 2×2, therefore it is fast and quite efficient. Possible options for masks for finding the gradient along the axes O_x and O_y:

$$G_x = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} \tag{7}$$

or

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{8}$$

As a result of applying the Roberts differential operator, we obtain an estimate of the gradient in the directions G_x and G_y. The all edge detectors gradient modulus can be calculated by the formula $G = \sqrt{G_x^2 + G_y^2} = |G_x| + |G_y|$, and gradient direction 0 the formula $\arctan\left(\frac{G_y}{G_x}\right)$.

OpenCV code

Listing 13: Rank detector using OpenCV and C++ programming language

```
void lab3::roberts_detector(const cv::Mat& im, cv::Mat& dst) {
  // reduce noise
  cv::Mat denoise_im;
  cv::GaussianBlur(im, denoise_im, cv::Size(3, 3), 0);
  dst = denoise_im.clone();

  // Roberts operator
  int nRows = dst.rows;
  int nCols = dst.cols;
  for (int i = 0; i < nRows - 1; i++) {
    for (int j = 0; j < nCols - 1; j++) {
      int t1 = (im.at<uchar>(i, j) - im.at<uchar>(i + 1, j + 1)) *
               (im.at<uchar>(i, j) - im.at<uchar>(i + 1, j + 1));
      int t2 = (im.at<uchar>(i + 1, j) - im.at<uchar>(i, j + 1)) *
               (im.at<uchar>(i + 1, j) - im.at<uchar>(i, j + 1));
      dst.at<uchar>(i, j) = (uchar)sqrt(t1 + t2);
    }
  }
}
```

Result

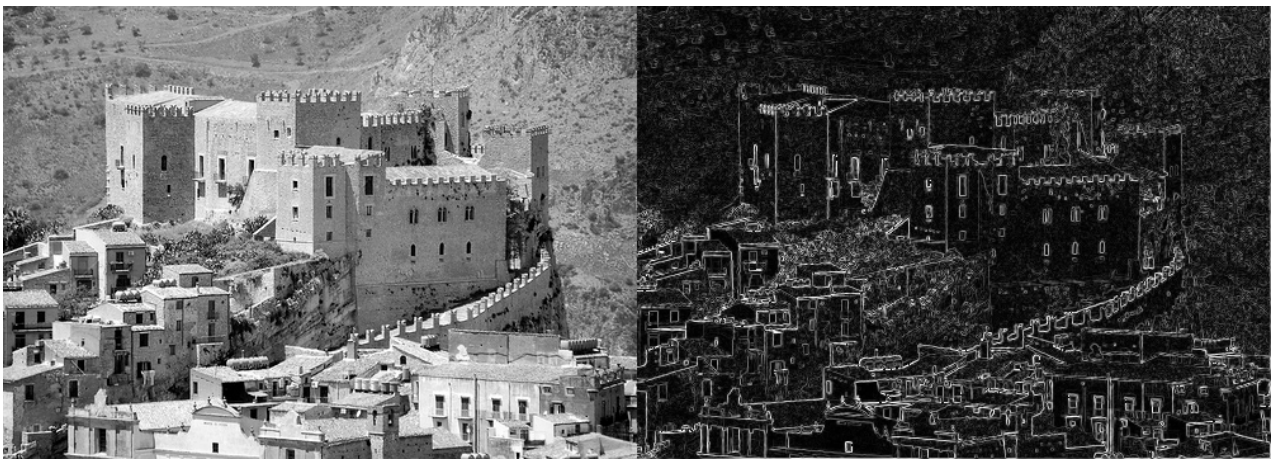


Figure 13: Roberts edges detector using OpenCV and C++ programming

2.3.2 Prewitt Edges Detector

This approach uses two orthogonal masks of size 3×3, allowing you to more accurately calculate the derivatives along the axes O_x and O_y:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{9}$$

OpenCV code

Listing 14: Prewitt edges detector using OpenCV and C++ programming language

```
void lab3::prewitt_detector(const cv::Mat& im, cv::Mat& dst) {
  // reduce noise
  cv::Mat denoise_im;
  cv::GaussianBlur(im, denoise_im, cv::Size(3, 3), 0);

  // Prewitt operator
  cv::Mat Gx = (cv::Mat_<int>(3, 3) << -1, 0, 1, -1, 0, 1, -1, 0, 1);
  cv::Mat Gy = (cv::Mat_<int>(3, 3) << -1, -1, -1, 0, 0, 0, 1, 1, 1);
  cv::Mat dst_x, dst_y;
  cv::filter2D(denoise_im, dst_x, -1, Gx);
  cv::filter2D(denoise_im, dst_y, -1, Gy);

  // Merge image
  cv::addWeighted(dst_x, 0.5, dst_y, 0.5, 0.0, dst);
}
```

Result

2.3.3 Sobel Edges Detector

This approach is similar to the Roberts filter, but different mask weights are used. A typical example of a Sobel filter:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{10}$$

OpenCV code

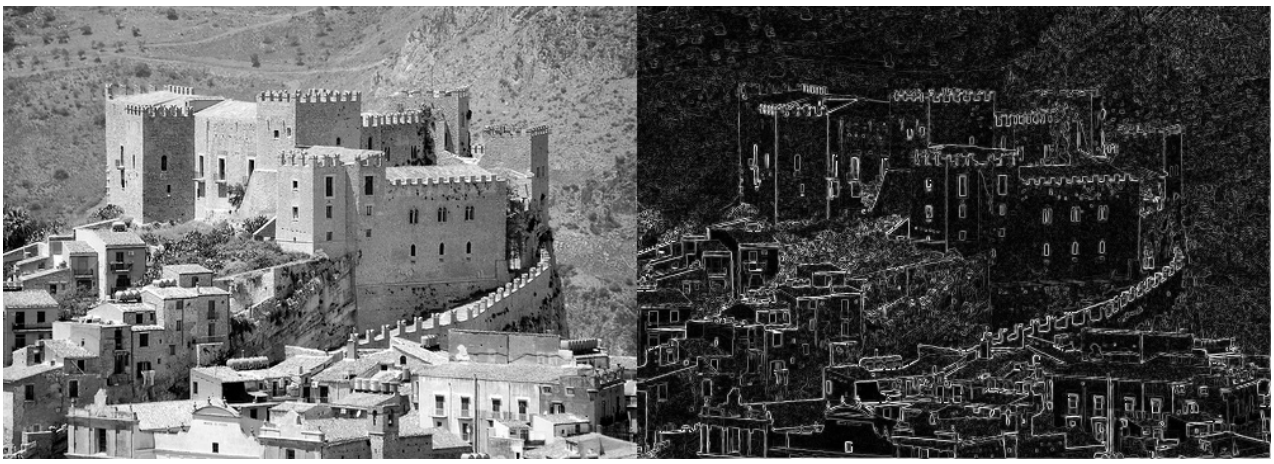


Figure 14: Prewitt edges detector using OpenCV and C++ programming

Listing 15: Sobel edges detector using OpenCV and C++ programming language

```
void lab3::sobel_detector(int ksize, const cv::Mat& im, cv::Mat& dst) {
  // reduce noise
  cv::Mat denoise_im;
  cv::GaussianBlur(im, denoise_im, cv::Size(3, 3), 0);

  // Sobel operator
  cv::Mat grad_x, grad_y;
  cv::Mat abs_grad_x, abs_grad_y;
  cv::Sobel(denoise_im, grad_x, CV_16S, 1, 0, ksize, 1, 0, cv::
   BORDER_DEFAULT);
  cv::Sobel(denoise_im, grad_y, CV_16S, 0, 1, ksize, 1, 0, cv::
   BORDER_DEFAULT);

  // converting back to CV_8U
  cv::convertScaleAbs(grad_x, abs_grad_x);
  cv::convertScaleAbs(grad_y, abs_grad_y);

  // Merge image
  cv::addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0, dst);
}
```

Result

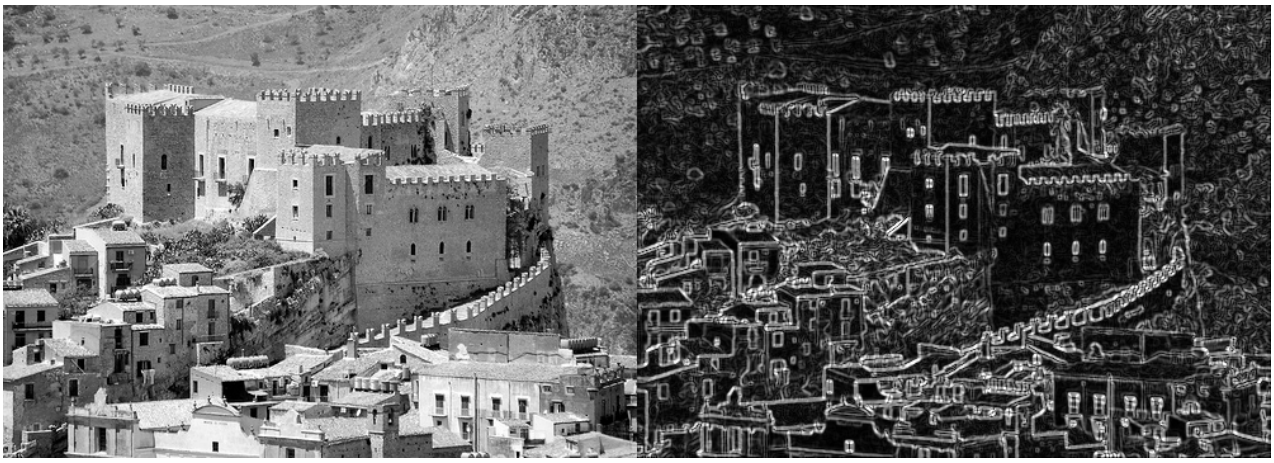


Figure 15: Sobel edges detector using OpenCV and C++ programming

2.3.4 Laplace Edges Detector

Laplace filter uses approximation of the second derivatives along the axes O_x and O_y

$$L(I(x,y)) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{11}$$

Formula 11 can be approximated by the following mask:

$$w = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{12}$$

OpenCV code

Listing 16: Laplace edges detector using OpenCV and C++ programming language

```
void lab3::laplacian_detector(int kernel_size, const cv::Mat& im,
                              cv::Mat& dst) {
  // reduce noise
  cv::Mat denoise_im;
  cv::GaussianBlur(im, denoise_im, cv::Size(3, 3), 0);

  int scale = 1;
  int delta = 0;
  int ddepth = CV_16S;

  // Laplacian operator
  cv::Laplacian(denoise_im, dst, ddepth, kernel_size, scale, delta,
            cv::BORDER_DEFAULT);

  // converting back to CV_8U
  cv::convertScaleAbs(dst, dst);
}
```

Result

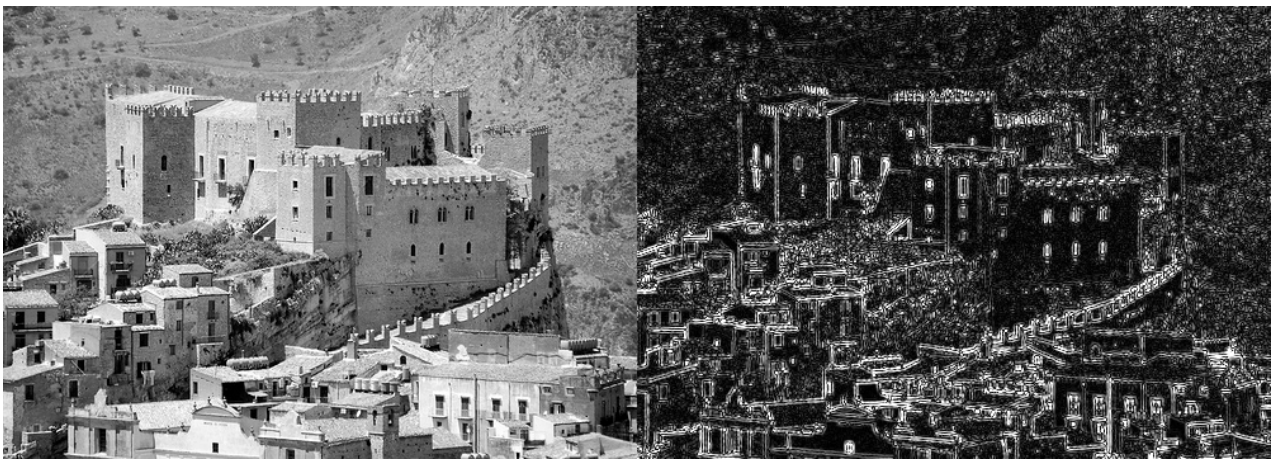


Figure 16: Laplace edges detector using OpenCV and C++ programming

2.3.5 Canny Algorithm

Canny Edge Detection is a popular edge detection algorithm. It was developed by John F. Canny. It is a multi-stage algorithm and we will go through each stages:

1. Noise Reduction
   Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter. We have already seen this in previous chapters.

2. Finding Intensity Gradient of the Image Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (G_x) and vertical direction (G_x). From these two images, we can find edge gradient and direction for each pixel as follows:

$$\begin{aligned} G &= \sqrt{G_x^2 + G_y^2} \\ \theta &= \tan^{-1}\left(\frac{G_y}{G_x}\right) \end{aligned} \tag{13}$$

   Gradient direction is always perpendicular to edges.

3. After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighborhood in the direction of gradient.

4. This stage decides which are all edges are really edges and which are not. For this, we need two threshold values, minVal and maxVal. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded.

OpenCV code

Listing 17: Canny edges detection algorithm using OpenCV and C++ programming language

```
void lab3::canny_detector(int low_threshold, int kernel_size, const cv::Mat&
    im, cv::Mat& dst) {
  const int ratio = 3;
  // reduce noise
  cv::Mat denoise_im;
  cv::GaussianBlur(im, denoise_im, cv::Size(3, 3), 0);
  // Canny Alogrithm in OpenCV
  cv::Canny( denoise_im, dst, low_threshold, low_threshold*ratio,
   kernel_size );
}
```

Result

3 Conclusion

In this work, we implemented several type of noise into images, with the generated noise images, we performed different filters to denoise the image. In the end, by applied convolution kernel, we able to implement several edge detector, Canny algorithm gives the best result by using multi-stages of filtering.

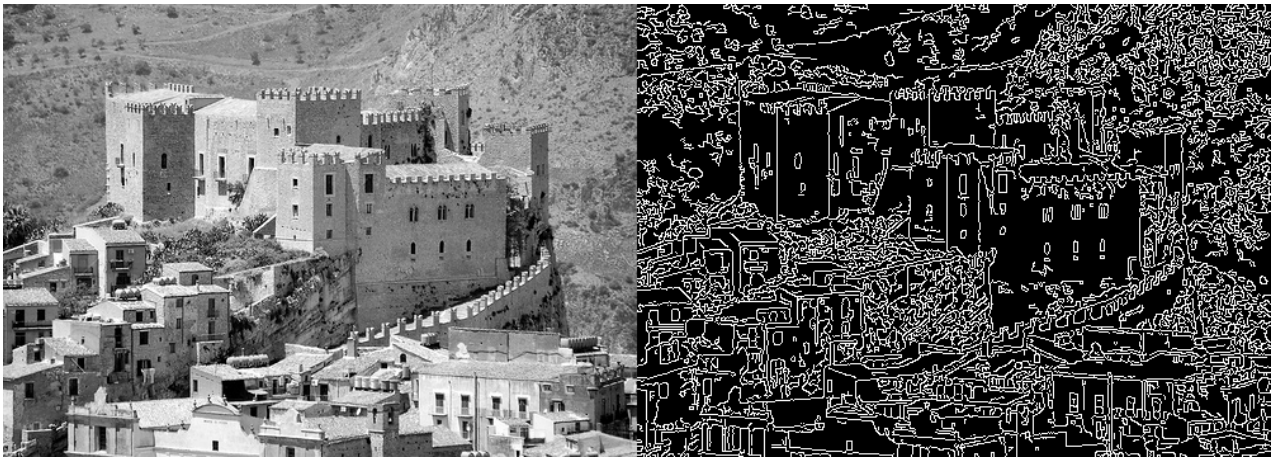


Figure 17: Canny edges detection algorithm using OpenCV and C++ programming