Report:

Task 1:

The main idea behind the median filtering is deciding the intensity of pixels only based on values of pixels in neighborhood. The kernel defined explicitly or convolution are not needed in this type of filtering. The intensity of pixels in the neighborhood are stored and sorted, then the middle element of sorted intensity values will assigned to respective pixel.

Both code and result images are provided below

Original

3x3

5x5

The function below in initial part of median filter the start position is set based on the size of kernel. It is worth noting that due initial positions are not set at zero and given some amount of offset, all the pixels along the vertical and horizontal corners(edges) of the image will be black because of the empty destination matrix.

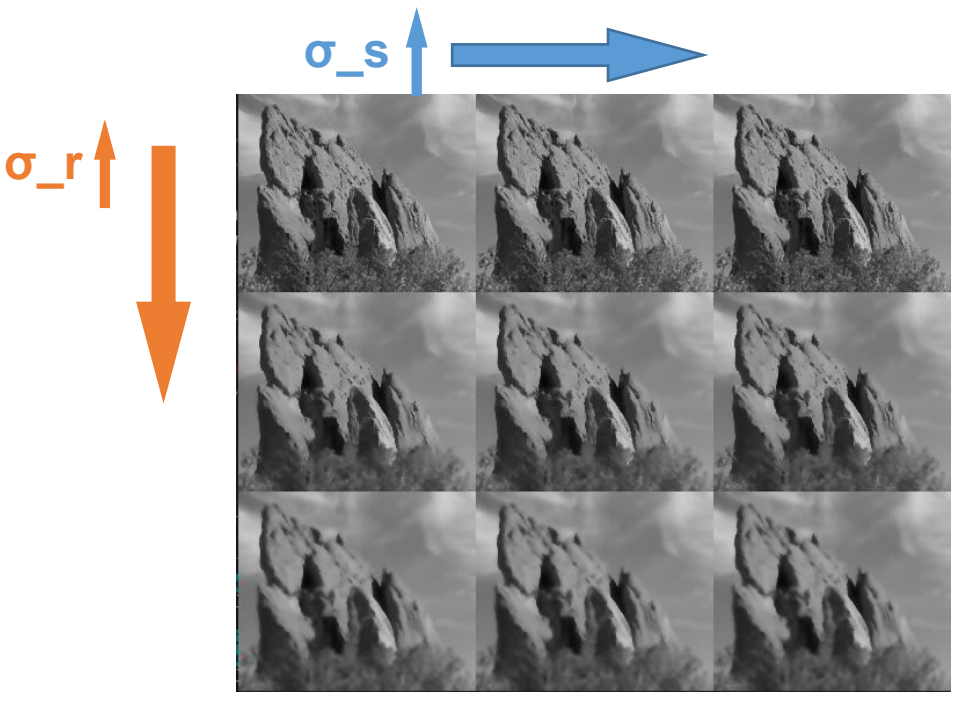
```
void apply_median(Mat& src, Mat& dst, int size)
{
    uchar* dstData = dst.data;
    uchar* srcData = src.data;
    int width = src.cols;
    int height = src.rows;

    for(int i = (size / 2); i < height - (size / 2); i++)
    {
        for(int j = (size / 2); j < width - (size / 2); j++)
        {
            dstData[i * width + j] = sort_n_find(srcData, j, i, width, height, size);
        }
    }
}
```

The function below the stores the intensities of nearby pixels into array. After this stage, array would be sorted via various sorting algorithms(In this case selection sort is being used). At last the the middle element of the array would be returned as the pixel value. Worth mentioning that, the intensity value of pixel being manipulated is not taken into account.

```
int sort_n_find(uchar* data, int x, int y, int width, int height, int size)
{
    int start = size / 2;
    int arr_size = size * size - 1;
    int arr[arr_size];
    memset(arr, 0, arr_size);
    for(int i = -start; i <= start; i++)
    {
        for(int j = -start; j <= start; j++)
        {
            if(i == 0 && j == 0) continue;
            arr[(i + start) * size + (j + start)] = data[(y + j) * width + (x + i)];
        }
    }
    for(int i = 0; i < arr_size ; i++)
    {
        int min_index = i;
        for(int j = i + 1; j < arr_size; j++)
        {
            if(arr[min_index] > arr[j])
            {
                min_index = j;
            }
        }
        swap(arr[min_index], arr[i]);
    }
    return arr[arr_size / 2];
```

Task 2:

In comparison with Gaussian Filter, Bilateral filter considers the rapid change in intensity values while performing the blur. The result image and code can be observed below:



σ_r – Range (Intensity)

σ_s - Spatial (Distance)

The function is a just a entry point to main filtering process and applies the intensity change to all the pixel.

```
void apply_bilateral_filter(Mat& src, Mat& dst, double sigma_s, double sigma_i, int size)
{
    uchar* srcData = src.data;
    uchar* dstData = dst.data;
    for(int i = 0; i < src.rows; i++)
    {
        for(int j = 0; j < src.cols; j++)
        {
            dstData[i * src.cols + j] = bilateral_filter(srcData, j, i, src.cols, src.rows, sigma_i, sigma_s,
            size);
        }
    }
}
```

The function below performs the actual filtering. First it calculates the amount of contribution based on the distance from the center pixel then it finds the amount of contribution based on the intensity difference between center pixel neighboring pixels.After the values respective to distance and intensity obtained, these are then used find and decide new intensity value for center pixel. This specific behavior of filter allows it to perform the blurring without losing the sharpness of the edges(Unlike Gaussian Filter).

```
double bilateral_filter(uchar* data, int x, int y, int width, int height, double sigma_i, double sigma_s, int size)
{
    double iFiltered = 0;
    double wp = 0, res = 0, total = 0;
    int sta double gi  2;
    double gi, gs;
    int dif;
    for(int i = -start; i <= start; i++)
    {
        for(int j = -start; j <= start; j++)
        {
            if(x + j < width && x + j > 0 && y + i < height && y + i > 0)
            {
                gs = exp(-(i * i + j * j) / (2.0 * sigma_s * sigma_s)) / (2 * CV_PI * sigma_s * sigma_s);
                dif = data[y * width + x] - data[(y + i) * width + (x + j)];
                gi = exp(-(dif * dif) / (2.0 * sigma_i * sigma_i)) / (2 * CV_PI * sigma_i * sigma_i);
                wp = gi * gs;
                res += data[(y + i) * width + (x + j)] * wp;
                total += wp;
            }
        }
    }
    return res / total;
}
```

Task 3:

The main reason, why the reason the sobel edge detection flaws in real world application is that, even if slight intensity change is detected algorithm going to consider this slight change as an edge, that further would lead to the high number of unwanted, useless edges. In order to resolve this issue the Canny edge detection is used, by performing double thresholding excluding unwanted or thick edges leaving only thin edges.

Sobel operator: As we see a lot unwanted edges

Canny operator: Less edges

Table below demonstrates the result of canny filtering with different threshold values with execution time;

The time taken for execution of different threshold values

```
[INFO]Time-duration for thershold1 = 120.000000 thershold2 = 200.000000: 0.010519
[INFO]Time-duration for thershold1 = 150.000000 thershold2 = 200.000000: 0.004175
[INFO]Time-duration for thershold1 = 180.000000 thershold2 = 200.000000: 0.003452
[INFO]Time-duration for thershold1 = 120.000000 thershold2 = 220.000000: 0.004596
[INFO]Time-duration for thershold1 = 150.000000 thershold2 = 220.000000: 0.005133
[INFO]Time-duration for thershold1 = 180.000000 thershold2 = 220.000000: 0.004120
[INFO]Time-duration for thershold1 = 120.000000 thershold2 = 240.000000: 0.003903
[INFO]Time-duration for thershold1 = 150.000000 thershold2 = 240.000000: 0.004684
[INFO]Time-duration for thershold1 = 180.000000 thershold2 = 240.000000: 0.003096
```

Observing the result given above it can be clearly stated that as the values of threshold1 is decreasing the amount of execution time increases and as the values of threshold2 is increasing the execution time decreases. Such pattern occurs because, the low threshold values will detect more edges, more edges mean more non-maximum suppression and edge tracking operations, potentially leading to increased execution time. The high value of threshold leads that less edges would be detected, reducing the amount of computation needed for edge tracking.

Student Name: Javid Guliyev

Student ID: 12235421