Sort and Filters
Assignment 5

Gwenaelle Cunha Sergio
ABR Lab – BEP
KNU 2014.1

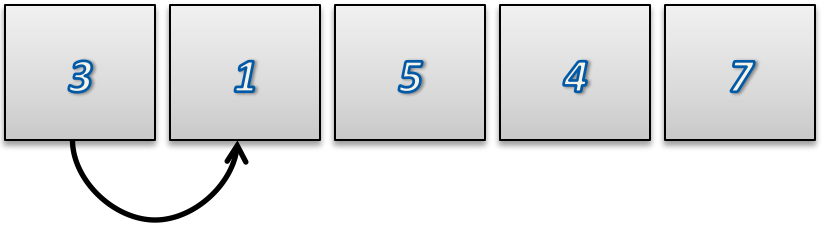
Contents

- Sort
  - Bubble sort
  - Quick sort
- Filters
  - Average
  - Median
  - Maximum
  - Minimum
  - Midpoint
  - Alpha-trimmed mean

Bubble sort

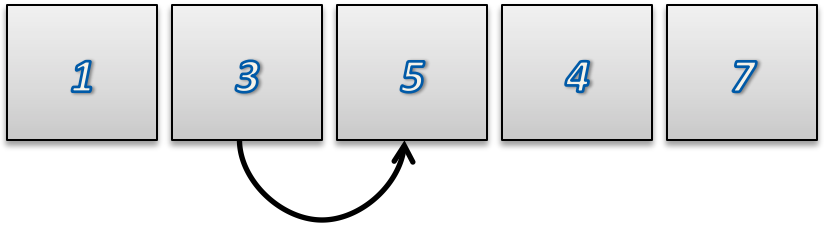
- Sorts array of data by changing adjacent values and thus bringing the ones with greater value to the end of the vector, like a *bubble*.

Demonstration



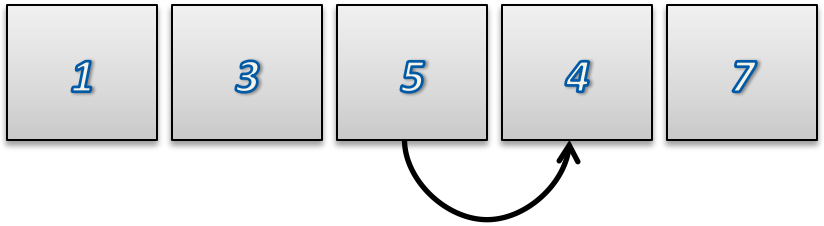
3 > 1?
Yes. Switch.

Demonstration



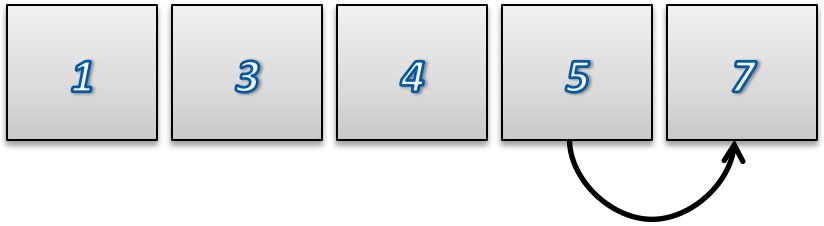
3 > 5?
No. Next.

Demonstration



5 > 4?
Yes. Swicth.

Demonstration



5 > 7?
No. Next.

Demonstration

Check vector again, since there won't be any switches,
the vector is sorted

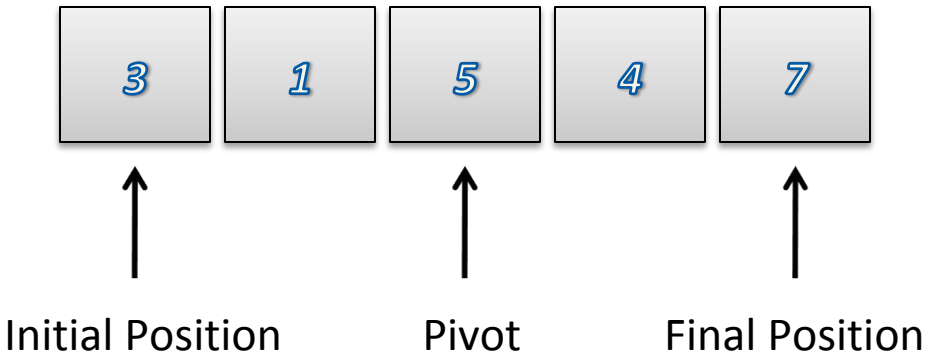
Bubble sort - Code

```
void bubblesort(int vector[], int size) {
    bool switched = true;
    while (switched) {
        switched = false;
        for (int i = 0; i < size - 1; i++) {
            if (V[i] > V[i + 1]) {
                aux = V[i];
                V[i] = V[i + 1];
                V[i + 1] = aux;
                switched = true;
      }
    }
  }
}
```

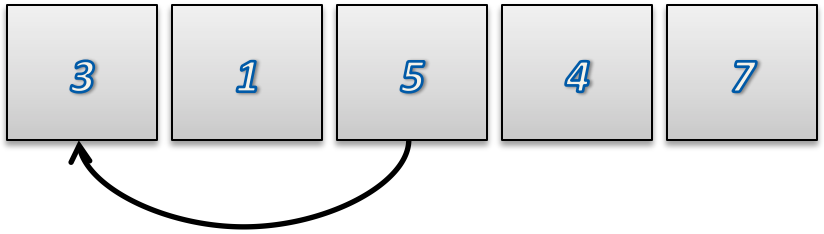
Quick sort

- Divide and conquer
- Pivot: index in the middle of the array
- Array is organized:
  - left of pivot: smaller numbers
  - right of pivot: bigger numbers
- Divide array in two
- Repeat the steps for the two new arrays until they are completely sorted out

Demonstration

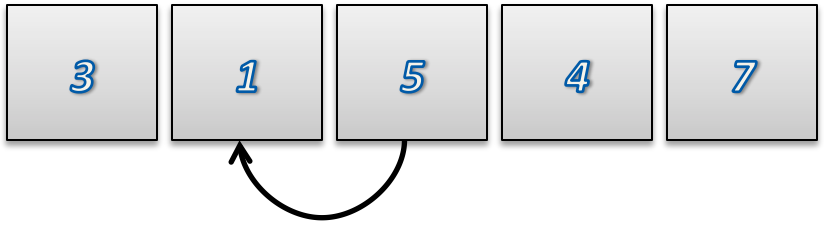


Demonstration



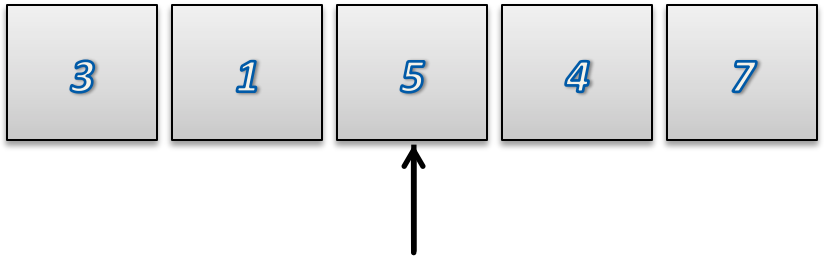
5 > 3?
Yes. Next.

Demonstration



5 > 1?
Yes. Next

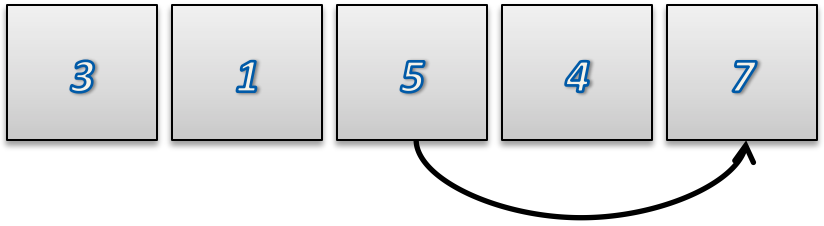
Demonstration



5 > 5?

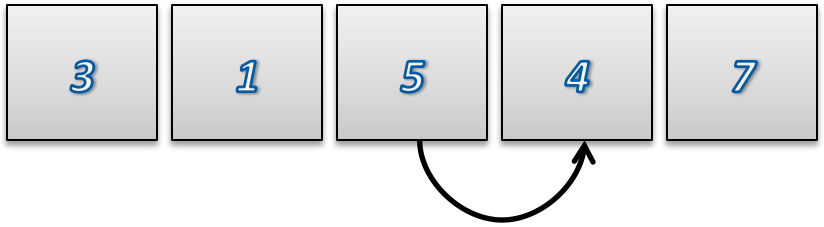
No. Pointer stays in this index.

Demonstration



5 < 7?
Yes. Next.

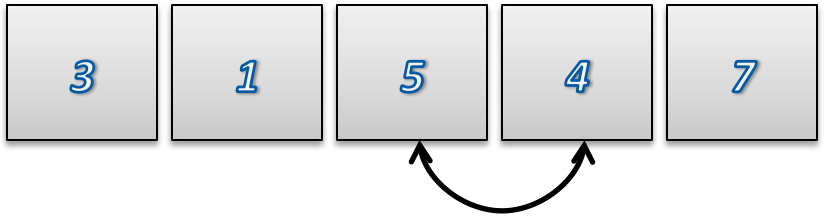
Demonstration



5 < 4?

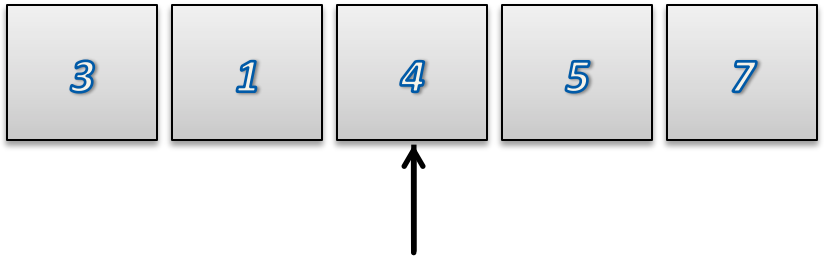
No. Pointer stays in this index.

Demonstration



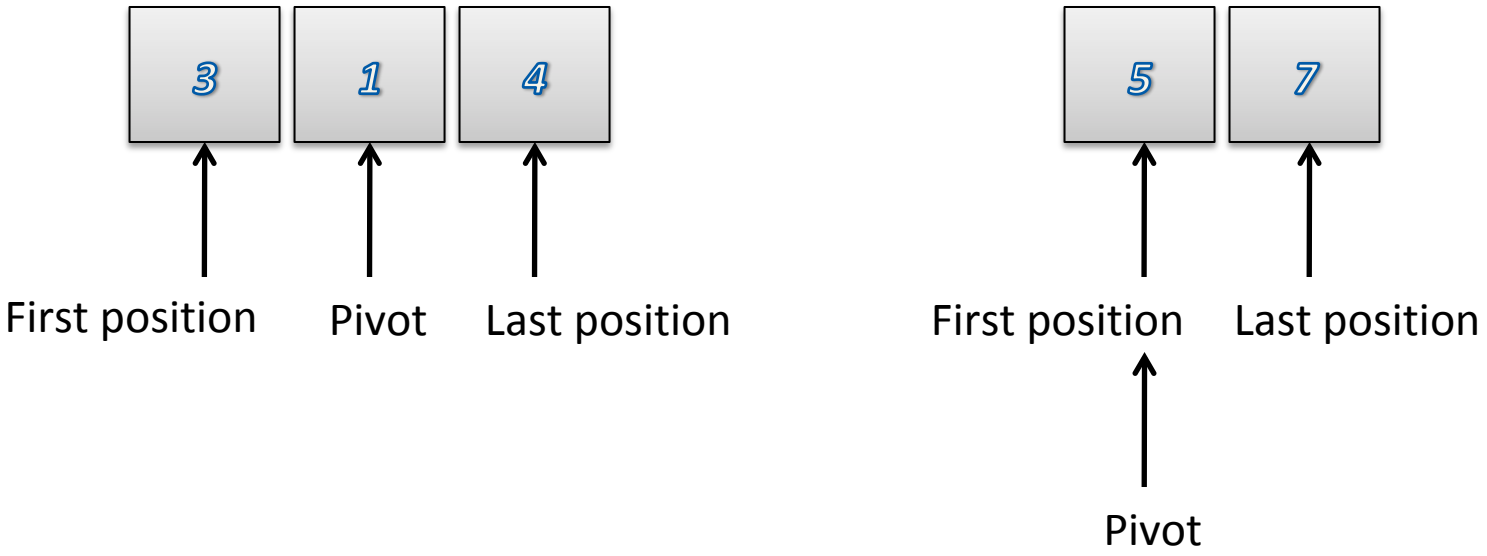
The pointers are in those two positions, so they are switched.

Demonstration

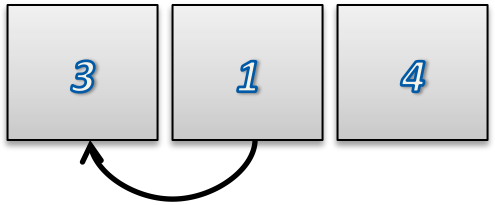


Now all the values to the left of the pivot are smaller than it and the ones to the right are bigger. Now, divide The vector.

Demonstration



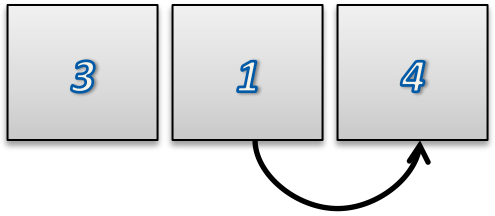
Demonstration



3 < 1?
No. Pointer stays in this position.

Sorted

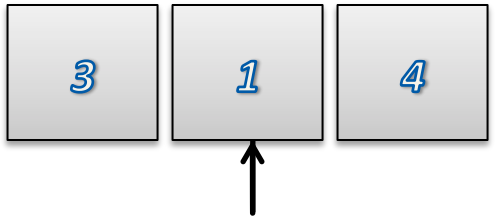
Demonstration



4 > 1?
Yes. Next.

Sorted

Demonstration

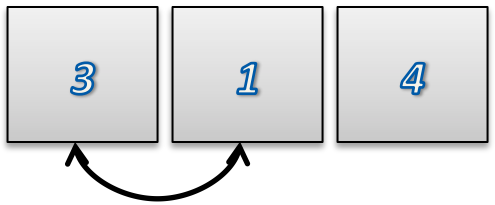


1 > 1?

No. Pointer stays in this position.

Sorted

Demonstration



Switch

Sorted

Demonstration

Sorted

Sorted

Demonstration

Quick sort - Code

```
void quicksort(int vector[], int inivet, int endvet) {
   int i = inivet;
   int j = endvet;
   int pivo = vector[(int) ((inivet + endvet) / 2)];
   int aux;
   while (i < j) {
      while (vector[i] < pivo) i++;
      while (vector[j] > pivo) j--;
      if (i <= j) {
         aux = vector[i];
         vector[i] = vector[j];
         vector[j] = aux;
         i++;
         j--;
      }
  }
   if (j > inivet) quicksort(vector, inivet, j);
   if (i < endvet) quicksort(vector, i, endvet);
}
```

Filters

- Goal: remove noise in an image.
- The way it functions is by taking each pixel and calculating its new value using a spatial mask and its neighboring pixels.
- Mask size: usually an odd number, i.e. 3x3, 5x5, 7x7, etc.
- Image convolution: the mask moves around the picture from the up-left corner to the down-right corner, covering all the pixels.
- Techniques that allows the analysis of pixels in the border of the image: adding a new boundary to the original image with the color black (used here) or copying the the same boundary again.

Average

- Average of values inside the mask

$$M_{3x3} = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

$$M_{5x5} = \begin{pmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{pmatrix}$$

Average - Code

```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int radius = 1, pixelValue;
for (int i = 0; i < mat.rows; i++){
   for (int j = 0; j < mat.cols; j++){
      pixelValue = 0;
      for (int ki = i-radius; ki <= i+radius; ki++){
         for (int kj = j-radius; kj <= j+radius; kj++){
            if ((ki < 0 || kj < 0) ||
               (ki >= mat.rows || kj >= mat.cols)) {
               vectorPixelsValue += 0;
            } else {
               vectorPixelsValue += mat.at<uchar>(ki,kj);
            }
      }
    }
      filter3x3.at<uchar>(i,j) = vectorPixelsValue / kernelArea ;
   }
}
```

Average - Result

original

3x3

5x5

7x7

Median

- Used for understory edge part to remove impulse noise.

- Representation of an order filter: it sorts out the mask values and takes the middle element as the pixel’s new value.

Median - Code

```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int radius = 1, middleIdx = (kernelArea-1)/2, vectorPixelsValue[9];
for (int i = 0; i < mat.rows; i++){
   for (int j = 0; j < mat.cols; j++){
      int cont = 0;
      for (int ki = i-radius; ki <= i+radius; ki++){
         for (int kj = j-radius; kj <= j+radius; kj++){
            if ((ki < 0 || kj < 0) ||
               (ki >= mat.rows || kj >= mat.cols)) {
               vectorPixelsValue[cont++] = 0;
            } else {
               vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
            }
         }
      }
      sort(vectorPixelsValue, kernelArea);
      filter3x3.at<uchar>(i,j) = vectorPixelsValue[middleIdx];
   }
}
```

Median - Result

original

3x3

5x5

7x7

Maximum

- Similar to median filter, but instead of taking the middle value, it takes the largest value as the pixel's new value.

Maximum - Code

```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int radius = 1, vectorPixelsValue[9];
for (int i = 0; i < mat.rows; i++){
   for (int j = 0; j < mat.cols; j++){
      int cont = 0;
      for (int ki = i-radius; ki <= i+radius; ki++){
         for (int kj = j-radius; kj <= j+radius; kj++){
            if ((ki < 0 || kj < 0) ||
               (ki >= mat.rows || kj >= mat.cols)) {
               vectorPixelsValue[cont++] = 0;
            } else {
               vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
            }
       }
    }
      sort(vectorPixelsValue, kernelArea);
      filter3x3.at<uchar>(i,j) = vectorPixelsValue[kernelArea-1];
    }
}
```

Maximum - Result

original

3x3

5x5

7x7

Minimum

- Also similar to median filter, but instead of taking the middle value, it takes the smallest value as the pixel's new value.

Minimum - Code

```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int radius = 1, vectorPixelsValue[9];
for (int i = 0; i < mat.rows; i++){
   for (int j = 0; j < mat.cols; j++){
      int cont = 0;
      for (int ki = i-radius; ki <= i+radius; ki++){
         for (int kj = j-radius; kj <= j+radius; kj++){
            if ((ki < 0 || kj < 0) ||
               (ki >= mat.rows || kj >= mat.cols)) {
               vectorPixelsValue[cont++] = 0;
            } else {
               vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
            }
         }
      }
      sort(vectorPixelsValue, kernelArea);
      filter3x3.at<uchar>(i,j) = vectorPixelsValue[0];
   }
}
```

Minimum - Result

original

3x3

5x5

7x7

Midpoint

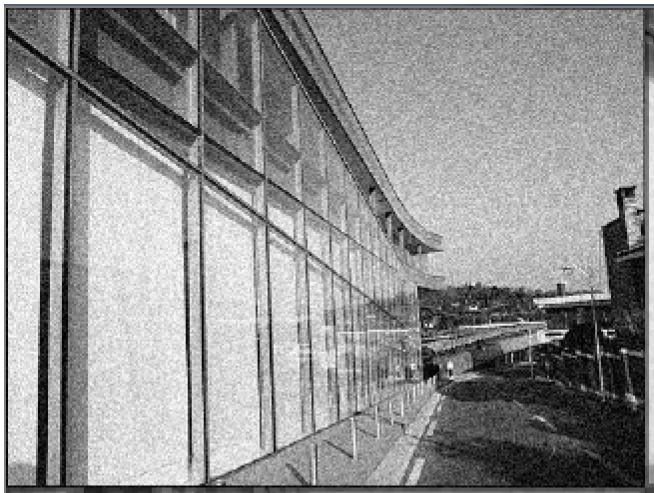
- Takes the average value between the smallest and the largest value in the sorted mask as the new pixel value.

Midpoint - Code

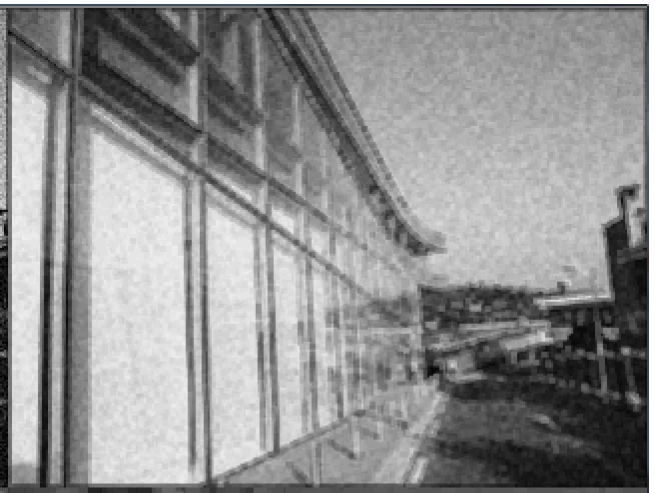
```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int radius = 1, vectorPixelsValue[9];
for (int i = 0; i < mat.rows; i++){
   for (int j = 0; j < mat.cols; j++){
      int cont = 0;
      for (int ki = i-radius; ki <= i+radius; ki++){
         for (int kj = j-radius; kj <= j+radius; kj++){
            if ((ki < 0 || kj < 0) ||
               (ki >= mat.rows || kj >= mat.cols)) {
               vectorPixelsValue[cont++] = 0;
            } else {
               vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
            }
         }
      }
      sort(vectorPixelsValue, kernelArea);
      filter3x3.at<uchar>(i,j) = cvRound((vectorPixelsValue[0]+
   vectorPixelsValue[kernelArea-1])/2);
      }
}
```

Midpoint - Result

original

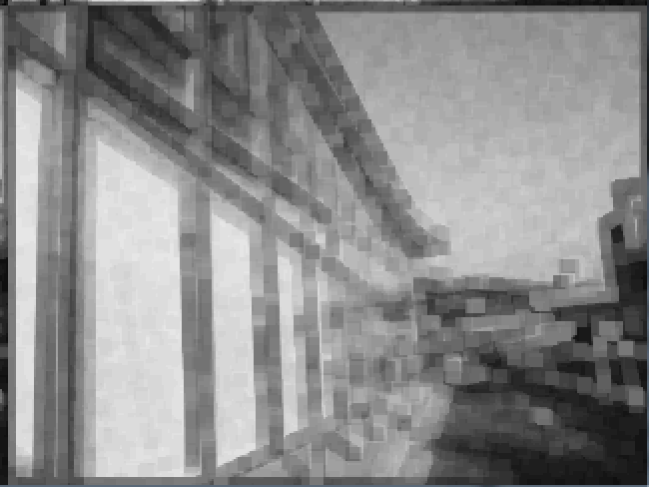


3x3



5x5

7x7



Alpha-trimmed Mean

- Opposite of midpoint filter: it takes all the values, except the smallest and largest to calculate the average value, according to alpha:

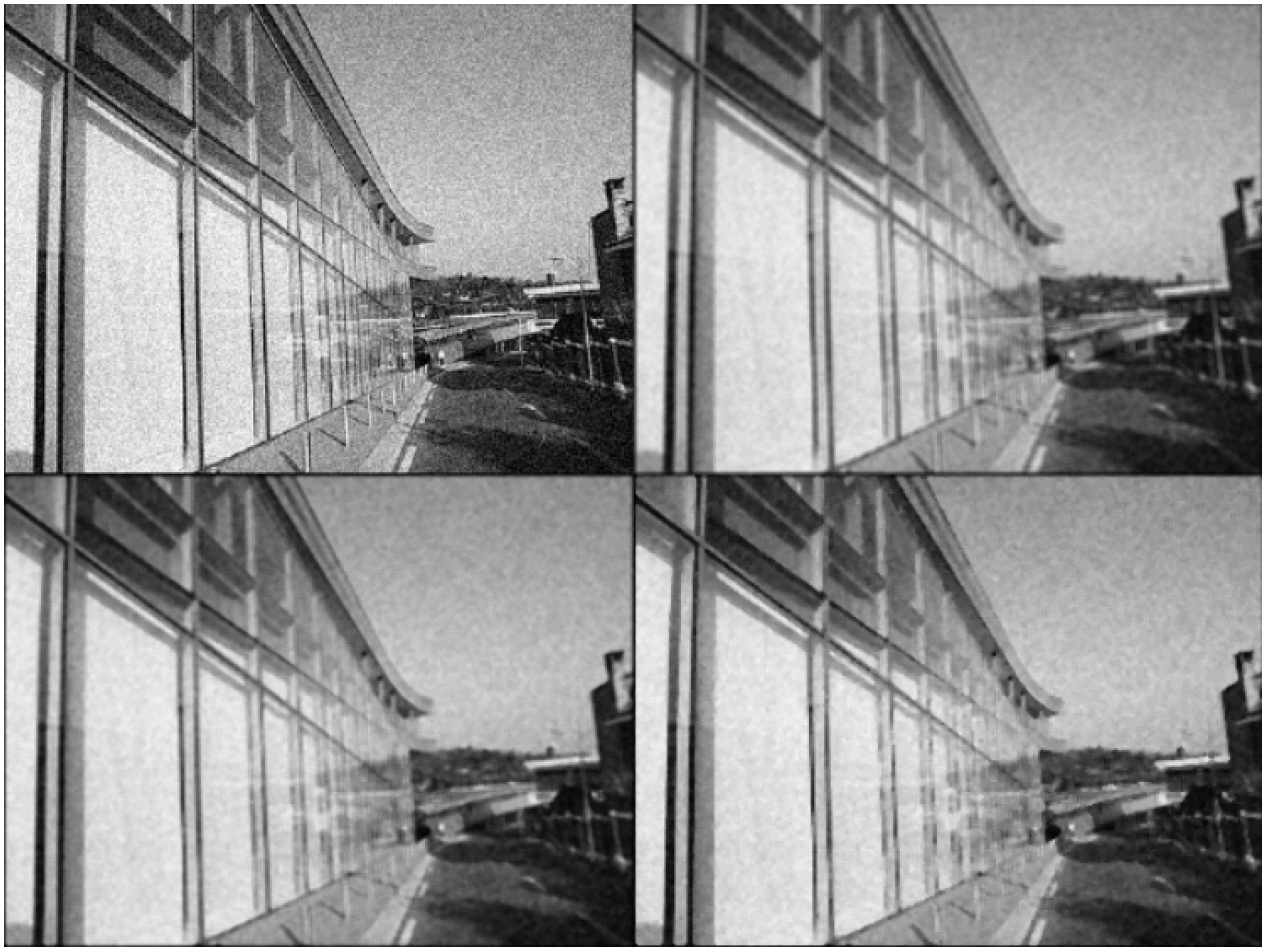
$$\frac{\sum_{\alpha}^{N-\alpha} elements}{N-2\alpha}$$

Alpha-trimmed Mean - Code

```
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
   int radius = 1, vectorPixelsValue[9];
   for (int i = 0; i < mat.rows; i++){
      for (int j = 0; j < mat.cols; j++){
         int cont = 0;
         for (int ki = i-radius; ki <= i+radius; ki++){
            for (int kj = j-radius; kj <= j+radius; kj++){
               if ((ki < 0 || kj < 0) ||
                  (ki >= mat.rows || kj >= mat.cols)) {
                  vectorPixelsValue[cont++] = 0;
               } else {
                  vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
               }
            }
         }
         sort(vectorPixelsValue, kernelArea);
          for (int k = 0; k < 3; k++){
                alpha = k*2;
                int elementValue = 0;
                for (int vi = alpha; vi < kernelArea-alpha; vi++){
          elementValue += vectorPixelsValue[vi];
                }
                int N = kernelArea-2*alpha; //all elements from alpha to N-alpha-1
                filter3x3[k].at<uchar>(i,j) = cvRound(elementValue/N);
          }
   }
```

Alpha-trimmed Mean - Result

original



α = 0

α = 2

α = 4

Filter = 3x3