```cpp
#include <opencv2/opencv.hpp>
#include <iostream>
#include <string>
#include <stdlib.h>
#include <math.h>

using namespace cv;
using namespace std;

    /*********************************************************************
    *                           EXERCISE 3                               *
    *                          -----------                               *
    *                           FILTERING                                *
    *********************************************************************/

void func(char *, char *, int, short);
void lowPass(IplImage *, IplImage *);
void fractileFilter(IplImage *, IplImage *, int);
int doubleStream(CvMat *);
void xGradientFilter(IplImage *, IplImage *);
void yGradientFilter(IplImage *, IplImage *);

int main(int argc, char ** argv) {

    //double t = (double)getTickCount();                        // TIMING

                    ///    DON'T OVERWRITE THE ORIGINAL IMAGE!!!

    IplImage * original, * filtered;

        /// Upload in grey scale with a resolution 640x480
    original = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);

        /// To use 'lowPass'
    filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);
    cvSet(filtered, 0);   // Set to black


    /*********************************************************************
    *                           PARTS 1 & 2                              *
    *                          -------------                             *
    *                   Low- and High-Pass Filtering                     *
    *********************************************************************/

        /// My version (Low-Pass):
        /// Slower but it removes much more noise in uniform areas
    lowPass(original, filtered);

        /// Using OpenCV function (Low-Pass):
    IplImage * LP_filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);
    IplImage * HP_filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);

        /// Building the Kernels for the filters

        /// Low Pass
    float LP_mat [3][3];

    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            LP_mat[i][j] = 1.0 / 9;

    CvMat LP_kernel = cvMat(3,3,CV_32F, LP_mat);

        /// High-Pass
    float HP_mat [3][3] = {{    0    ,  -1.0/4  ,     0   },
                           { -1.0/4 ,    2.0    ,  -1.0/4},
                           {    0    ,  -1.0/4  ,     0   }};

    CvMat HP_kernel = cvMat(3, 3, CV_32F, HP_mat);

        /// Filtering
    cvFilter2D(original, HP_filtered, &HP_kernel, cvPoint(-1,-1));
    cvFilter2D(original, LP_filtered, &LP_kernel, cvPoint(-1,-1));


    /*********************************************************************
    *                              PART 3                                *
    *                             ---------                              *
    *                          Percentile Filter                         *
    *********************************************************************/

    /// First attempt - imaged cropped (rim rejected)
    /// Doesn't really work that well... Destroys some details and doesn't reduce
    /// much noise
```

```
    IplImage * perc_filtered = cvCreateImage(cvSize(original->width - 2, original->height
- 2), IPL_DEPTH_8U, 1);
    fractileFilter(original, perc_filtered, 50);  // Median filter: 50%


    /*********************************************************************
    *                               PART 4                               *
    *                             ---------                              *
    *                        Laplace-Gaussian Filter                     *
    *********************************************************************/
    IplImage * LG_filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);
    /*float LG_mat [9][9] = {{0,0,1,2,2,2,1,0,0},
                           {0,1,5,10,12,10,5,1,0},
                           {1,5,15,19,16,19,15,5,1},
                           {2,10,19,-19,-64,-19,19,10,2},
                           {2,12,16,-64,-148,-64,16,12,2},
                           {2,10,19,-19,-64,-19,19,10,2},
                           {1,5,15,19,16,19,15,5,1},
                           {0,1,5,10,12,10,5,1,0},
                           {0,0,1,2,2,2,1,0,0}};

    CvMat LG_kernel = cvMat(9, 9, CV_32F, LG_mat);*/

    /// This gives a much better result!!!
    /*float LG_mat [3][3] = {{ -1 , -1, -1},
                           { -1 ,  8, -1},
                           { -1 , -1, -1}};*/

    CvMat LG_kernel = cvMat(3, 3, CV_32F, LG_mat);
    /** NOTE: Probably because of how the function cvFilter2D is
    implemented it's better to use a matrix of float and to use
    a constructor of CvMat with CV_32F. Even when the values
    are integers, by specifying e.g. CV_16S results are weird and
    more processing is necessary to obtain the right visual result...*/

        /// Filtering
    cvFilter2D(original, LG_filtered, &LG_kernel, cvPoint(-1,-1));

    ///*****************************************************************
        /// Stream from camera in normal and filtered mode (2 separate windows)
    doubleStream(&LG_kernel);


    /*********************************************************************
    *                               PARTS 5                              *
    *                             ---------                              *
    *                         Triangular Filter                          *
    *********************************************************************/
    IplImage * x_filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);
    IplImage * y_filtered = cvCreateImage(cvGetSize(original), IPL_DEPTH_8U, 1);
        /// Emphasize the horizontal direction gradients
    xGradientFilter(original, x_filtered);
        /// Emphasize the vertical direction gradients
    yGradientFilter(original, y_filtered);


    /*********************************************************************
    *                          Display and Save                          *
    *********************************************************************/

    cvNamedWindow("Original Image (grey scale)", WINDOW_NORMAL);
    cvNamedWindow("Low-Pass Filtered Image (mine)", WINDOW_NORMAL);
    cvNamedWindow("Low-Pass Filtered Image", WINDOW_NORMAL);
    cvNamedWindow("High-Pass Filtered Image", WINDOW_NORMAL);
    cvNamedWindow("Fractile Filtered Image", WINDOW_NORMAL);
    cvNamedWindow("Laplace-Gauss Filtered Image", WINDOW_NORMAL);
    cvNamedWindow("x gradient", WINDOW_NORMAL);
    cvNamedWindow("y gradient", WINDOW_NORMAL);

    cvShowImage("Original Image (grey scale)", original);
    cvShowImage("Low-Pass Filtered Image (mine)", filtered);
    cvShowImage("Low-Pass Filtered Image", LP_filtered);
    cvShowImage("High-Pass Filtered Image", HP_filtered);
    cvShowImage("Fractile Filtered Image", perc_filtered);
    cvShowImage("Laplace-Gauss Filtered Image", LG_filtered);
    cvShowImage("x gradient", x_filtered);
    cvShowImage("y gradient", y_filtered);

    cvSaveImage("Original (grey scale).png", original);
    cvSaveImage("My_LP_Filtered.png", filtered);
    cvSaveImage("LP_Filtered.png", LP_filtered);
    cvSaveImage("HP_Filtered.png", HP_filtered);
    cvSaveImage("Fractile Filter.png", perc_filtered);
    cvSaveImage("Laplace-Gauss_Filter.png", LG_filtered);
```

```
    cvSaveImage("x_gradient.png", x_filtered);
    cvSaveImage("y_gradient.png", y_filtered);

    cvWaitKey(0);


    /*********************************************************************
    *                                Clean-Up                            *
    *********************************************************************/
    //t = ((double)getTickCount() - t)/getTickFrequency();
    //printf("\nExecution time = %.3f s", t);                // END TIMING

    cvReleaseImage(&original);
    cvReleaseImage(&filtered);
    cvReleaseImage(&LP_filtered);
    cvReleaseImage(&HP_filtered);
    cvReleaseImage(&perc_filtered);
    cvReleaseImage(&LG_filtered);
    cvReleaseImage(&x_filtered);
    cvReleaseImage(&y_filtered);

    cvDestroyAllWindows();

    return 0;
}



    /*********************************************************************
    * Filter an image using the function cvFilter2D and the kernel      *
    * specified to emphasize horizontal gradients.                      *
    *********************************************************************/
void xGradientFilter(IplImage * original, IplImage * x_filtered) {
    /*float x_mat [3][3] = {{ -1 , 0, 1},
                          { -2 , 0, 2},
                          { -1 , 0, 1}};*/
    CvMat x_kernel = cvMat(3, 3, CV_32F, x_mat);
    cvFilter2D(original, x_filtered, &x_kernel, cvPoint(-1,-1));
}


    /*********************************************************************
    * Filter an image using the function cvFilter2D and the kernel      *
    * specified to emphasize vertical gradients.                        *
    *********************************************************************/
void yGradientFilter(IplImage * original, IplImage * y_filtered) {
    /*float y_mat [3][3] = {{ -1, -2, -1},
                          {  0,  0,  0},
                          {  1,  2,  1}};*/

    CvMat y_kernel = cvMat(3, 3, CV_32F, y_mat);
    cvFilter2D(original, y_filtered, &y_kernel, cvPoint(-1,-1));
}


    /*********************************************************************
    * streams in 2 different windows the input from a webcam.           *
    * One window shows the stream from the camera directly.             *
    * The second window shows the processed stream.                     *
    * The kernel of the filter to be applied is passed as a pointer to  *
    * a CvMat so different type of filtering are possible.              *
    * The function returns 0 if everything goes well, -1 otherwise.     *
    *********************************************************************/
int doubleStream(CvMat * kernel) {
    VideoCapture cap(1);

    Mat frame, tframe, frame_grey;

    if (!cap.isOpened()) {
        printf("Cam could not be accessed\n");
        return -1;
    }

    namedWindow("Cam");
    cvNamedWindow("LG_Cam", 0);

    while(cap.read(frame)) {
        GaussianBlur(frame, tframe, Size(3,3), 0, 0, BORDER_DEFAULT);
        cvtColor(tframe, frame_grey, CV_BGR2GRAY);
        cvtColor(frame, frame_grey, CV_BGR2GRAY);
        IplImage * LG_im_cam = new IplImage(frame_grey);
        cvFilter2D(LG_im_cam, LG_im_cam, kernel, cvPoint(-1,-1));
        imshow("Cam", frame);
        cvShowImage("LG_Cam", LG_im_cam);
```

```
        if (waitKey(10) >= 0) {
            imwrite("Cam.png", frame);
            cvSaveImage("LG_Cam.png", LG_im_cam);
            break;
        }

        if(frame.empty()) {
            printf("End of stream\n");
            break;
        }
    }
    cvDestroyAllWindows();

    return 0;
}


    /**********************************************************************
    * Starting from an image this function applies a percentile filter  *
    * The percentage is passed as a parameter.                           *
    * A 3x3 window is assumed.                                           *
    **********************************************************************/
/// prototypes of the supporting functions used by the following
void MtoV(char *, int, uchar []);
int cmpfunc (const void *, const void *);

void fractileFilter(IplImage * src, IplImage * dest, int p) {
    uchar v[9];
    int percentile = 9 * p / 100;
    for(int i = 0; i < dest->height; i++) {
        char * psrc = src->imageData + i*src->widthStep;
        char * pdest = dest->imageData + i*dest->widthStep;
        for(int j = 0; j < dest->width; j++) {
            MtoV(psrc, src->widthStep, v);       // pixels within window in an array
            qsort(v, 9, sizeof(uchar), cmpfunc); // Quicksort
            *pdest = v[percentile];
            psrc++;
            pdest++;
        }
    }
}

    /**********************************************************************
    * Comparing function used in the call to 'qsort' in 'fractileFilter *
    **********************************************************************/

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

    /**********************************************************************
    * starting from a 3x3 ROI of a IplImage (so of type char), this     *
    * function creates a 9-elements array.                               *
    *                                                                    *
    * NB. In this context the order of the elements is not relevant.    *
    **********************************************************************/

void MtoV(char * src, int step, uchar v[]) {
    for(int i = 0; i < 3; i++) {
        char * psrc = src + i*step;
        for(int j = 0; j < 3; j++) {
            v[3 * i + j] = (uchar)(*psrc);
            psrc++;
        }
    }
}

    /**************************************************************************
    * Receives pointers to the source and destination images and produces    *
    * The filtered image in the destination.                                 *
    *                                                                        *
    * It uses a "frame" images to deal with the border of the image in a not *
    * too complicated way. The frame is a black image whose dimensions are   *
    * such that there is a 1-pixel large "frame" around the source image.    *
    *                                                                        *
    * When filtering one of the 4 corner pixels, 5 pixels out of 9 from the  *
    * Kernel stick out of the image and go over the "frame". These will not  *
    * contribute to the averaging (being equal to 0), but the divisor must   *
    * be changed to 4 (useful pixels in the Kernel) instead of 9. There is   *
    * no need to know which specific corner pixel is considered, it works in *
    * in general!                                                            *
    *                                                                        *
    * Similarly, for pixels in one of the borders (but not in the corners)   *
```

```
336     * 3 pixels of the Kernel stick out of the image and 6 must be used as    *
337     * divider for averaging.                                                  *
338     *                                                                         *
339     * IMPORTANT: Most of the pixels are of course in the inner part of the    *
340     * image, so that condition must be checked first and it will avoid other *
341     * conditions to be checked all the time for nothing. This improves the    *
342     * efficiency and the execution time!!!                                    *
343     * For the same reason, it would be better checking for the border before *
344     * checking for the corners, but the condition for the borders is very     *
345     * long and hard to read.                                                  *
346     ***************************************************************************/
347
348 #define INNER 9
349 #define CORNER 4
350 #define SIDE 6
351 //
352 void lowPass(IplImage * src, IplImage * dest) {
353         /// Container set to black (Frame)
354     IplImage * temp = cvCreateImage(cvSize(src->width + 2, src->height + 2),
    IPL_DEPTH_8U, 1);
355     cvSet(temp, 0);
356         /// Image "framed"
357     cvSetImageROI(temp, cvRect(1, 1, src->width, src->height));
358     cvCopy(src, temp);
359     cvResetImageROI(temp);
360
361     int N = dest->height, M = dest->width;
362     for(int i = 0; i < N; i++) {
363         char * pdest = dest->imageData + i*dest->widthStep;
364         char * ptemp = temp->imageData + i*temp->widthStep;
365         for(int j = 0; j < M; j++) {
366             if ((i > 0 && i < N - 1) && (j > 0 && j < M - 1))            // Inner Image
367                 func(ptemp, pdest, temp->widthStep, INNER);
368             else if ((i == 0 || i == N - 1) && (j == 0 || j == M - 1))   // Corners
369                 func(ptemp, pdest, temp->widthStep, CORNER);
370             else                                                        // Borders
371                 func(ptemp, pdest, temp->widthStep, SIDE);
372             pdest++;
373             ptemp++;
374         }
375     }
376     cvReleaseImage(&temp);
377 }
378 //*/
379
380     /***************************************************************************
381     * Support function for 'lowPass'. It does the dirty job!!!                *
382     * Takes 2 pointers to the source (framed) and destination images, the     *
383     * 'widthStep' value for the framed image to scan it and a divider dk      *
384     * that takes 3 possible values: 9 (inner pixels), 6 (border pixels), and *
385     * 4 (corner pixels).                                                      *
386     * It produces the value of a single pixel of the final image, pointed to *
387     * by pd, by averaging over the Kernel and taking into account the rim.    *
388     ***************************************************************************/
389 //
390 void func(char * pf, char * pd, int step, short dk) {
391     float k = 1.0 / dk;
392     *pd = 0;                                // pixel in the final image.
393     for(int i = 0; i < 3; i++) {
394         char * lpf = pf + i*step;           // Local pointer to framed image
395         for(int j = 0; j < 3; j++) {
396             /// After many tests I found that this sequence of casting
397             /// keeps the correct result at the end...
398             *pd += (uchar)(k * (float)((uchar)(*lpf)));
399             lpf++;
400         }
401     }
402 }
403 //*/
404
405
406     /***************************************************************************
407     * The following group of functions do exactly the same thing as the       *
408     * combination 'lowPass' + 'func' above.                                   *
409     * It was intended to be more efficient since every function does only     *
410     * what is strictly needed for a specific task, avoiding unnecessary       *
411     * computation. It's not as easy to read and it turned out to be just as   *
412     * efficient as above in terms of computation time. The nice thing, and    *
413     * the main reason why I keep it is that it is more efficient in terms of *
414     * memory requirements since it does not need a framed intermediate image.*
415     ***************************************************************************/
416 /*
417 void lowPass(IplImage * src, IplImage * dest) {
418     int N = dest->height, M = dest->width;
```

```
    for(int i = 0; i < N; i++) {
        char * p = dest->imageData + i*dest->widthStep;
        char * pp = src->imageData + i*src->widthStep;
        for(int j = 0; j < M; j++) {
            if ((i > 0 && i < N - 1) && (j > 0 && j < M - 1))       // Inner image
                func(pp, p, src->widthStep);

            else if (j == 0 && i > 0 && i < N - 1)                  // left border
                func_3((pp - src->widthStep), p, src->widthStep);
            else if (j == M - 1 && i > 0 && i < N - 1)              // right border
                func_3((pp - src->widthStep - 1), p, src->widthStep);
            else if (i == 0 && j > 0 && i < M - 1)                  // top border
                func_2((pp - 1), p, src->widthStep);
            else if (i == N - 1 && j > 0 && i < M- 1)               // bottom border
                func_2((pp - src->widthStep - 1), p, src->widthStep);

            else if (i == 0 && j == 0)                              // top-left
                func_1(pp, p, src->widthStep);
            else if (i == 0 && j == M - 1)                          // top-right
                func_1((pp - 1), p, src->widthStep);
            else if (i == N - 1 && j == 0)                          // bottom-left
                func_1((pp -  src->widthStep), p, src->widthStep);
            else if (i == N - 1 && j == M - 1)                      // bottom-right
                func_1((pp - src->widthStep - 1), p, src->widthStep);
            p++;
            pp++;
        }
    }
}

void func(char * ps, char * pd, int step) {
    float k = 1.0 / 9;
    *pd = 0;
    char * lps = ps - step - 1;
    for(int i = 0; i < 3; i++) {
        lps += i*step;
        for(int j = 0; j < 3; j++) {
            *pd += (uchar)(k * (float)((uchar)(*lps)));
            lps++;
        }
    }
}

void func_1(char * pp, char * p, int step) {
    *p = 0;
    *p += (uchar)((((float)((uchar)(*pp))) + ((float)((uchar)(*(pp + 1)))) +
          ((float)((uchar)(*(pp + step)))) + ((float)((uchar)(*(pp + step + 1))))) / 4);
}

void func_2(char * pp, char * p, int step) {
    *p = 0;
    *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1))) +
          (float)((uchar)(*(p + 2)))) / 6);
    pp += step;
    *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1))) +
          (float)((uchar)(*(p + 2)))) / 6);
}

void func_3(char * pp, char * p, int step) {
    *p = 0;
    for(int i = 0; i < 3; i++) {
        *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1)))) / 6);
        p += i*step;
    }
}
//*/
```