```cpp
#include <opencv2/opencv.hpp>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <fstream>
#include <vector>
#include <iomanip>
#include <algorithm>

using namespace std;
using namespace cv;

IplImage * ReadBytes(const char *);
IplImage * preprocessing (IplImage *);
IplImage * thresholding (IplImage *);
void ROIs(IplImage *, IplImage *, int, ofstream&);
bool check(Mat);

double performPCA(const vector<Point> &pts, Mat &img) {
    //Construct matrix for data
    int sz = static_cast<int>(pts.size());
    Mat data_pts = Mat(sz, 2, CV_64F);

    for (int i = 0; i < data_pts.rows; i++) {
        data_pts.at<double>(i, 0) = pts[i].x;
        data_pts.at<double>(i, 1) = pts[i].y;
    }

    //Perform PCA analysis
    PCA pca_analysis(data_pts, Mat(), PCA::DATA_AS_ROW);

    //Store the eigenvalues and eigenvectors
    vector<Point2d> eigen_vecs(2);
    vector<double> eigen_val(2);
    for (int i = 0; i < 2; i++) {
        eigen_vecs[i] = Point2d(pca_analysis.eigenvectors.at<double>(i, 0),
        pca_analysis.eigenvectors.at<double>(i, 1));
        eigen_val[i] = pca_analysis.eigenvalues.at<double>(i);
    }

    double angle = atan2(eigen_vecs[0].y, eigen_vecs[0].x); // orientation in radians
    ofstream myfile;

    myfile.open("data.txt", ios_base::app);
    myfile << "Orientation " << angle * 180 / CV_PI << endl;
    myfile << "Ratio " << eigen_val[0] / eigen_val[1] << endl;
    myfile << "Eigenvec " << eigen_vecs[0] << endl;
    myfile << "Eigenval " << eigen_val[0] << endl;
    myfile << "Eigenvec " << eigen_vecs[1] << endl;
    myfile << "Eigenval " << eigen_val[1] << endl;
    myfile.close();

    //return (eigen_val[0] / eigen_val[1]);
    return eigen_val[0];
}

bool forsort(const vector<Point>& c1, const vector<Point>& c2) {
```

```cpp
    return (contourArea(c1, false) < contourArea(c2, false));
}

double analysis(Mat gray) {
    Mat binary;
    threshold(gray, binary, 0, 255, THRESH_BINARY | THRESH_OTSU);

    //Contours in binary image
    vector<vector<Point> > contours;
    findContours(binary, contours, RETR_LIST, CHAIN_APPROX_NONE);

    //Sort contour in ascending order
    stable_sort(contours.begin(), contours.end(), forsort);

    //PCA part
    //If only 1 contour simple take the last (largest)
    double RAT;
    if(contours.size() < 2)
        RAT = performPCA(contours[contours.size() - 1], gray);
    //If 2 contours take 2 last ones
    else
        for (int i = 1; i < 3; i++)
            RAT = performPCA(contours[contours.size() - i], gray);

    binary.release();
    return RAT;
}


int findBackground(int *, int);
void calcHistogram(IplImage *, int *);
void drawHistogram(const char *, int *);
IplImage * simpleThresholding(IplImage *, unsigned);
IplImage * getBinImage(IplImage *, unsigned);

int findContrast(IplImage *, int *);
bool multi(IplImage *, int *);


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

    // OPEN INPUT FILE

    string line;
    ifstream out(argv[1]);

    // READ LINE BY LINE

    int k = 1;      // Image number

    ofstream fileout(argv[2]);

    while(getline(out, line)) {
        const char * imgFileName = line.c_str();        // .UNC file name

        // READ IMAGE
        IplImage * img = ReadBytes(imgFileName);

```

```cpp
        // PREPROCESSING
        IplImage * imgP = preprocessing(img);

        // THRESHOLDING
        IplImage * imgPBin = thresholding(imgP);

        // FIND ROI
        ROIs(img, imgPBin, k, fileout);

        // RELEASE
        cvReleaseImage(&img);
        cvReleaseImage(&imgP);
        cvReleaseImage(&imgPBin);

        k++;
    }


    // CLOSE FILE
    fileout.close();
    out.close();

    // CLEAN

    return 0;
}



IplImage * ReadBytes(const char * filename) {
    FILE * f = fopen(filename, "rb");

    if(f == NULL)
        throw "Argument Exception";

    unsigned char info[34];
    fread(info, sizeof(unsigned char), 34, f);     // read the 34-byte header

    // extract image height and width from header
    int width = *(short*)&info[30];
    int height = *(short*)&info[28];
    int IMOD = *(short*)&info[32];

    uchar data[height * (IMOD + 1)][width];
    unsigned char temp;

    for(int i = 0; i < height * (IMOD + 1); i++) {
        for(int j=0; j < width; j++) {
            fread(&temp, sizeof(unsigned char), 1, f);
            data[i][j] = temp;
        }
    }

    fclose(f);

    Mat imageMat = Mat(height * (IMOD + 1), width, CV_8U, data);

    IplImage * img = cvCreateImage(cvSize(imageMat.cols - 10, imageMat.rows),
```

```
        IPL_DEPTH_8U, 1);
    IplImage ipltemp = imageMat(Rect(9, 0, width - 10, height * (IMOD + 1)));
    cvCopy(&ipltemp, img);

    imageMat.release();

    return img;
}



IplImage * preprocessing(IplImage * img) {
    IplImage * I = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    cvSmooth(img, I, CV_BILATERAL, 0.05, 20);

    int IT = 1;
    cvDilate(I, I, NULL, IT);

    return I;
}



typedef struct THR {
    int B;
    int x;
} THR;



IplImage * thresholding(IplImage * img) {
    ///                       HISTOGRAM
    int dep = 256;
    int hist[dep];
    calcHistogram(img, hist);


    ///                COMPUTE THE BACKGROUND = B
    int B = findBackground(hist, img->height * img->width);

    ///                 COMPUTE THE THRESHOLD = t
    int t = 10;

    ///                 FINAL THRESHOLD T = B + t
    int T = t + B;

    ///                    USE T TO THRESHOLD
    IplImage * bin;
    bin = simpleThresholding(img, T);

    return bin;
}



void ROIs(IplImage * IPLimg, IplImage * IPLbin, int k, ofstream& fileout) {
    Mat img, bin;
    img = cvarrToMat(IPLimg);                   // Binary Image
```

```
    bin = cvarrToMat(IPLbin);                   // Original Image

    Mat bincpy = bin.clone();                   // Copy used by 'findContour'

    blur(bincpy, bincpy, CvSize(3, 3));

    double xc = IPLimg->width / 2 - 10;         // Remember 10 pixels are cut out on the
                                                // left side.
    double yc = IPLimg->height / 2;

    /// /////////////////////////////////////////////////////////////////////////////////////////
    ///                                  FIND CONTOURS                                         ///
    /// /////////////////////////////////////////////////////////////////////////////////////////

    vector<vector<Point> > contours;             // vector of vectors of Point --> Table of x and y coordinates

    findContours(bincpy, contours, CV_RETR_TREE, CV_CHAIN_APPROX_NONE, Point(0, 0));


    /// /////////////////////////////////////////////////////////////////////////////////////////
    ///   COMPUTE MOMENTS FOR EACH CONTOURS AND EXTRACT ONLY THE NORMALIZED 1ST ORDER MOMENTS ///
    /// /////////////////////////////////////////////////////////////////////////////////////////

    int N = contours.size();                      // Easier and less confusing to use N than calling the method every time

    /// /////////////////////////////////////////////////////////////////////////////////////////
    ///                         COUNT CONTOURS THAT HAVE ENOUGH POINTS                         ///
    /// /////////////////////////////////////////////////////////////////////////////////////////

    /// NOTE: Tests determined that 5 points is the minimum to be sure that moments can be computed correctly

    int K = 0;
    for(int i = 0; i < N; i++)
        if (contours[i].size() >= 5)
            K++;


    /// /////////////////////////////////////////////////////////////////////////////////////////
    ///                                RECAP OF THE COUNT                                      ///
    /// /////////////////////////////////////////////////////////////////////////////////////////

    //cout << endl << "Total number of objects: " << N << endl;
    //cout << endl << "Number of useful objects: " << K << endl;
```

```cpp


    /// ////////////////////////////////////////////////////////////////////////////////////
    ///                      FOR EACH USEFUL CONTOUR DETERMINES ALL MOMENTS                      ///
    /// ////////////////////////////////////////////////////////////////////////////////////

    vector<Moments> mmt(K);                    // Vector of objects of 'Moments'

    /// Keep the following 4 instructionS in mind and read the note below...
    int j = 0;
    for(int i = 0; i < N; i++)
        if (contours[i].size() >= 5)
            mmt[j++] = moments(contours[i], false);


    /// ////////////////////////////////////////////////////////////////////////////////////
    ///                                 NORMALIZED FIRST MOMENTS                                 ///
    /// ////////////////////////////////////////////////////////////////////////////////////

    /// NOTE: Necessary because the class 'Moments' doesn't have normalized first moments...
    vector<Point2f> norm1stMoments(K);

    for (int i = 0; i < K; i++)
        norm1stMoments[i] = Point2f((mmt[i].m10 / mmt[i].m00), (mmt[i].m01 / mmt[i].m00));


    /// NOTE: exactly the same for cycle is repeated twice: first, to compute K, and a second time to
    ///       calculate the moments. In the present implementation this is necessary because the value
    ///       K is used to declare 'mmt'.
    ///       The most elegant (and probably correct) method is allocating dynamically 'mmnts'.


    /// ////////////////////////////////////////////////////////////////////////////////////
    ///                         SCAN THROUGH COORDINATES OF EACH CONTOUR                         ///
    /// ////////////////////////////////////////////////////////////////////////////////////

    // These are used to form the name of the ROI images.
    string name = "ROI_";
    //string sufix = ".png"
    char numIm[21];
    sprintf(numIm, "%d", k);
    char numROI[21];
    int n = 0;

```

```cpp
    int X = bin.cols, Y = bin.rows;             // Image sizes
    int S = 10;                                 // Half side of the ROI


    /* Show which objects have been "seen" from the original image
    IplImage * imgcpy1 = cvCloneImage(IPLimg);
    IplImage * I1 = cvCreateImage(cvGetSize(imgcpy1), IPL_DEPTH_8U, 3);
    cvCvtColor(imgcpy1, I1, COLOR_GRAY2RGB);
    IplImage * imgcpy2 = cvCloneImage(IPLimg);
    IplImage * I2 = cvCreateImage(cvGetSize(imgcpy2), IPL_DEPTH_8U, 3);
    cvCvtColor(imgcpy2, I2, COLOR_GRAY2RGB);
    unsigned short A = 8;*/
    /// ////////////////

    string Rname;

    for (int i = 0; i < K; i++) {
        // Discard ROIs that fall out of the image
        if(norm1stMoments[i].x - S >= 0 && norm1stMoments[i].y - S >= 0 && norm1stMoments[i].x + S < X && norm1stMoments[i].y + S < Y) {


            //cvCircle(I1, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y), A, cvScalar(255,0,255), 1, 4);
            /// //////////////////////

            // Definition of ROI
            Rect rect(norm1stMoments[i].x - S, norm1stMoments[i].y - S, 2 * S, 2 * S);

            // Creation of ROI image with reasonable size
            Mat Roi, RoiCpy;
            resize(img(rect), Roi, Size(400, 400), 1, 1, INTER_LANCZOS4);

            //bilateralFilter(RoiCpy, Roi, 3, 1, 20);

            if (!check(Roi)) {
                n++;
                Roi.release();
                continue;
            }

            //cvCircle(I2, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y), A, cvScalar(0,0,255), 1, 4);

            // Ensure a coherent numeration (i.e. Use 'n' instead of 'i' as part of the name)
            sprintf(numROI, "%d", n);

            double EV = analysis(Roi);

            double R = sqrt(pow((norm1stMoments[i].x - xc), 2) + pow((norm1stMoments[i].y - yc), 2));

            Rname = name + numIm + "_" + numROI;

            fileout << setfill(' ') << setw(16) << left << Rname;
            fileout << " \t" << setw(7) << std::internal << R << "\t\t \t";
            fileout << EV << "\t\t";
```

```
            fileout << norm1stMoments[i] << "\n";

            //string result;
            //result = Rname + sufix;
            //imwrite(result, Roi);
            Roi.release();

            n++;
        }
        //else
            //cout << "Image # " << k << "   -  Discarded:  " << norm1stMoments[i] << 
              endl;
    }

    /*
    string check1 = "Check";
    string check2 = "Check_Result";
    string checkName1 = check1 + numIm + sufix;
    string checkName2 = check2 + numIm + sufix

    cvSaveImage(checkName1.c_str(), I1);
    cvReleaseImage(&imgcpy1);
    cvReleaseImage(&I1);

    cvSaveImage(checkName2.c_str(), I2);
    cvReleaseImage(&imgcpy2);
    cvReleaseImage(&I2);*/
    /// //////////////////////

    bin.release();
    img.release();
    bincpy.release();

}



bool check(Mat img) {
    int hist[256];

    // FAINTNESS
    IplImage * I = cvCreateImage(cvSize(img.cols, img.rows), IPL_DEPTH_8U, 1);
    IplImage Itemp = img;
    cvCopy(&Itemp, I);
    int c = findContrast(I, hist);

    if(c < 9){
        cvReleaseImage(&I);
        return false;
    }

    // MULTIPLICITY

    IplImage * cpy = cvCloneImage(I);

    // Opening: 12 iterations
    cvMorphologyEx(cpy, cpy, NULL, NULL, CV_MOP_OPEN, 12);
    // Erosion: 10 iterations
```

```cpp
    cvErode(cpy, cpy, NULL, 10);
    // Dilation: 3 iterations
    cvDilate(cpy, cpy, NULL, 3);

    calcHistogram(cpy, hist);
    int B = findBackground(hist, cpy->height * cpy->width);

    int t = 10;
    int T;
    IplImage * bin = NULL;

    while(t <= 20) {
        T = t + B;
        bin = simpleThresholding(cpy, T);
        if(multi(bin, hist)) {
            cvReleaseImage(&I);
            cvReleaseImage(&cpy);
            cvReleaseImage(&bin);
            return false;
        }
        t += 4;
    }
    return true;
}



bool multi(IplImage * img, int * hist) {
    vector<vector<Point> > segm;
    Mat imgM = cvarrToMat(img);
    findContours(imgM, segm, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
    imgM.release();

    if(segm.size() > 1)
        return true;
    else
        return false;
}



int findContrast(IplImage * img, int hist []) {

    calcHistogram(img, hist);

    int cnt = 0, TOT = (img->height * img->width);
    int p5, p95;
    bool f1 = true, f2 = true;

    for(int i = 0; i < 256; i++) {
        cnt += hist[i];
        if(f1 && ((cnt * 100) / TOT) >= 5) {
            p5 = i;
            f1 = false;
        }

        if(f2 && ((cnt * 100) / TOT) >= 95) {
            p95 = i;
```

```
            f2 = false;
        }
    }

    return (p95 - p5);
}


// Support function used by calcHistogram
void func(uchar c, int * h) {
    unsigned x = c;
    (*(h + x))++;
}


void calcHistogram(IplImage * gray, int hist []) {
        /// Initialize histograms
    for (int i = 0; i < 256; i++)
        hist[i] = 0;

        /// Calculate histogram
    for(int i = 0; i < gray->height; i++) {
        char * ptr = gray->imageData + i*gray->widthStep;
        for(int j = 0; j < gray->width; j++) {
                func((uchar)(*ptr), hist);
                ptr++;
        }
    }
}



void drawHistogram(const char * histName, int * hist) {
    int st = 4;             // To separate lines in the histograms, for better look ;)
    int dep = 256;

    /// Create image for the histogram
    IplImage * his = cvCreateImage(cvSize(st * dep, 600), IPL_DEPTH_8U, 3);
    cvSet(his, cvScalar(0,0,0));            // Initialize image (all black)
    his->origin = IPL_ORIGIN_BL;            // Set the origin in the bottom left corner

    for (int i = 0; i < 8 ; i++) {
        int x = dep / 8;
        cvLine(his, cvPoint(i * x * st, 0), cvPoint(i * x * st, 600), cvScalar(122,122,122), 1, 4);
    }

        /// Draw the histogram
    for (int i = 0; i < dep; i++)
        if (hist[i] != 0)
            cvLine(his, cvPoint(i * st, 0), cvPoint(i * st, hist[i] / 10), cvScalar(255, 0, 0), 1, 4);

    cvSaveImage(histName, his);

    cvReleaseImage(&his);
}

```

```


int findBackground(int hist[], int tot) {
    bool fmin = true, fmax = true;
    int m, M;

    for (int i = 0; fmin || fmax; i++) {
        if (fmin)
            if(hist[i]){
                m = i;
                fmin = false;
            }
        if (fmax)
            if(hist[255 - i]){
                M = 255 - i;
                fmax = false;
            }
    }

    THR res_t;
    res_t.B = m + (M - m) / 16;

    int frac = 45, cnt = 0;

    for(int i = m; i <= res_t.B; i++)
        cnt += hist[i];

    if( (res_t.x = ((cnt * 100) / tot)) == frac)
        return res_t.B;
    else if (res_t.x < frac) {
        while(res_t.x < frac) {
            res_t.B++;
            cnt += hist[res_t.B];
            res_t.x = (cnt * 100) / tot;
        }
        THR t1, t2;
        t1.B = res_t.B;
        t1.x = res_t.x;
        t2.B = res_t.B - 1;
        t2.x = ((cnt - hist[t2.B])* 100) / tot;
        if (abs(t2.x - frac) < abs(t1.x - frac) )
            return t2.B;

        else
            return t1.B;

    }
    else { // x > frac
        while(res_t.x > frac) {
            res_t.B--;
            cnt -= hist[res_t.B];
            res_t.x = (cnt * 100) / tot;
        }
        THR t1, t2;
        t1.B = res_t.B;
        t1.x = res_t.x;
        t2.B = res_t.B + 1;
        t2.x = ((cnt + hist[t2.B])* 100) / tot;
```

```
        if (abs(t2.x - frac) < abs(t1.x - frac) )
            return t2.B;
        else
            return t1.B;
    }
}



IplImage * simpleThresholding(IplImage * img, unsigned th) {
    IplImage * bin = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    for(int i = 0; i < img->height; i++) {
        char * ptr = img->imageData + i*img->widthStep;
        char * p = bin->imageData + i*bin->widthStep;
        for(int j = 0; j < img->width; j++) {
                if ((uchar)(*ptr) >= (uchar)th)
                    *p = 255;
                else
                    *p = 0;
            ptr++;
            p++;
        }
    }
    return bin;
}
```