

```

1  #include <opencv2/opencv.hpp>
2  #include <iostream>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string>
6  #include <fstream>
7  #include <vector>
8  #include <iomanip>
9  #include <algorithm>
10
11 using namespace std;
12 using namespace cv;
13
14 IplImage * ReadBytes(const char *);
15 IplImage * preprocessing (IplImage *);
16 IplImage * thresholding (IplImage *);
17 void ROIs(IplImage *, IplImage *, int, ofstream&);
18 bool check(Mat);
19
20 double performPCA(const vector<Point> &pts, Mat &img) {
21     //Construct matrix for data
22     int sz = static_cast<int>(pts.size());
23     Mat data_pts = Mat(sz, 2, CV_64F);
24
25     for (int i = 0; i < data_pts.rows; i++) {
26         data_pts.at<double>(i, 0) = pts[i].x;
27         data_pts.at<double>(i, 1) = pts[i].y;
28     }
29
30     //Perform PCA analysis
31     PCA pca_analysis(data_pts, Mat(), PCA::DATA_AS_ROW);
32
33     //Store the eigenvalues and eigenvectors
34     vector<Point2d> eigen_vecs(2);
35     vector<double> eigen_val(2);
36     for (int i = 0; i < 2; i++) {
37         eigen_vecs[i] = Point2d(pca_analysis.eigenvectors.at<double>(i, 0),
38             pca_analysis.eigenvectors.at<double>(i, 1));
39         eigen_val[i] = pca_analysis.eigenvalues.at<double>(i);
40     }
41
42     double angle = atan2(eigen_vecs[0].y, eigen_vecs[0].x); // orientation in radians
43     ofstream myfile;
44
45     myfile.open("data.txt", ios_base::app);
46     myfile << "Orientation " << angle * 180 / CV_PI << endl;
47     myfile << "Ratio " << eigen_val[0] / eigen_val[1] << endl;
48     myfile << "Eigenvector " << eigen_vecs[0] << endl;
49     myfile << "Eigenvalue " << eigen_val[0] << endl;
50     myfile << "Eigenvector " << eigen_vecs[1] << endl;
51     myfile << "Eigenvalue " << eigen_val[1] << endl;
52     myfile.close();
53
54     //return (eigen_val[0] / eigen_val[1]);
55     return eigen_val[0];
56 }
57
58 bool forsort(const vector<Point>& c1, const vector<Point>& c2) {

```

```

59     return (contourArea(c1, false) < contourArea(c2, false));
60 }
61
62 double analysis(Mat gray) {
63     Mat binary;
64     threshold(gray, binary, 0, 255, THRESH_BINARY | THRESH_OTSU);
65
66     //Contours in binary image
67     vector<vector<Point> > contours;
68     findContours(binary, contours, RETR_LIST, CHAIN_APPROX_NONE);
69
70     //Sort contour in ascending order
71     stable_sort(contours.begin(), contours.end(), forsort);
72
73     //PCA part
74     //If only 1 contour simple take the last (largest)
75     double RAT;
76     if(contours.size() < 2)
77         RAT = performPCA(contours[contours.size() - 1], gray);
78     //If 2 contours take 2 last ones
79     else
80         for (int i = 1; i < 3; i++)
81             RAT = performPCA(contours[contours.size() - i], gray);
82
83     binary.release();
84     return RAT;
85 }
86
87
88 int findBackground(int *, int);
89 void calcHistogram(IplImage *, int *);
90 void drawHistogram(const char *, int *);
91 IplImage * simpleThresholding(IplImage *, unsigned);
92 IplImage * getBinImage(IplImage *, unsigned);
93
94 int findContrast(IplImage *, int *);
95 bool multi(IplImage *, int *);
96
97
98 int main(int argc, char ** argv) {
99
100     // OPEN INPUT FILE
101
102     string line;
103     ifstream out(argv[1]);
104
105     // READ LINE BY LINE
106
107     int k = 1;        // Image number
108
109     ofstream fileout(argv[2]);
110
111     while(getline(out, line)) {
112         const char * imgFileName = line.c_str();        // .UNC file name
113
114         // READ IMAGE
115         IplImage * img = ReadBytes(imgFileName);
116

```

```

117     // PREPROCESSING
118     IplImage * imgP = preprocessing(img);
119
120     // THRESHOLDING
121     IplImage * imgPBin = thresholding(imgP);
122
123     // FIND ROI
124     ROIs(img, imgPBin, k, fileout);
125
126     // RELEASE
127     cvReleaseImage(&img);
128     cvReleaseImage(&imgP);
129     cvReleaseImage(&imgPBin);
130
131     k++;
132 }
133
134
135 // CLOSE FILE
136 fileout.close();
137 out.close();
138
139 // CLEAN
140
141 return 0;
142 }
143
144
145
146 IplImage * ReadBytes(const char * filename) {
147     FILE * f = fopen(filename, "rb");
148
149     if(f == NULL)
150         throw "Argument Exception";
151
152     unsigned char info[34];
153     fread(info, sizeof(unsigned char), 34, f);    // read the 34-byte header
154
155     // extract image height and width from header
156     int width = *(short*)&info[30];
157     int height = *(short*)&info[28];
158     int IMOD = *(short*)&info[32];
159
160     uchar data[height * (IMOD + 1)][width];
161     unsigned char temp;
162
163     for(int i = 0; i < height * (IMOD + 1); i++) {
164         for(int j=0; j < width; j++) {
165             fread(&temp, sizeof(unsigned char), 1, f);
166             data[i][j] = temp;
167         }
168     }
169
170     fclose(f);
171
172     Mat imageMat = Mat(height * (IMOD + 1), width, CV_8U, data);
173
174     IplImage * img = cvCreateImage(cvSize(imageMat.cols - 10, imageMat.rows),

```

```

        IPL_DEPTH_8U, 1);
175     IplImage ipltemp = imageMat(Rect(9, 0, width - 10, height * (IMOD + 1)));
176     cvCopy(&ipltemp, img);
177
178     imageMat.release();
179
180     return img;
181 }
182
183
184
185 IplImage * preprocessing(IplImage * img) {
186     IplImage * I = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
187     cvSmooth(img, I, CV_BILATERAL, 0.05, 20);
188
189     int IT = 1;
190     cvDilate(I, I, NULL, IT);
191
192     return I;
193 }
194
195
196
197 typedef struct THR {
198     int B;
199     int x;
200 } THR;
201
202
203
204 IplImage * thresholding(IplImage * img) {
205     /// HISTOGRAM
206     int dep = 256;
207     int hist[dep];
208     calcHistogram(img, hist);
209
210
211     /// COMPUTE THE BACKGROUND = B
212     int B = findBackground(hist, img->height * img->width);
213
214     /// COMPUTE THE THRESHOLD = t
215     int t = 10;
216
217     /// FINAL THRESHOLD T = B + t
218     int T = t + B;
219
220     /// USE T TO THRESHOLD
221     IplImage * bin;
222     bin = simpleThresholding(img, T);
223
224     return bin;
225 }
226
227
228
229 void ROIs(IplImage * IPLimg, IplImage * IPLbin, int k, ofstream& fileout) {
230     Mat img, bin;
231     img = cvarrToMat(IPLimg); // Binary Image

```

```

232     bin = cvarrToMat(IPLbin);                                // Original Image
233
234     Mat bincpy = bin.clone();                                // Copy used by 'findContour'
235
236     blur(bincpy, bincpy, CvSize(3, 3));
237
238     double xc = IPLimg->width / 2 - 10;                      // Remember 10 pixels are cut out on the
239                                                                // left side.
240     double yc = IPLimg->height / 2;
241
242     /// //////////////////////////////////////// ↗
243     ///                                     FIND                ↗
244     ///          CONTOURS                                     ↗
245     /// //////////////////////////////////////// ↗
246     vector<vector<Point> > contours;                          // vector of vectors of Point --> Table ↗
247     // of x and y coordinates
248     findContours(bincpy, contours, CV_RETR_TREE, CV_CHAIN_APPROX_NONE, Point(0, 0));
249
250
251     /// //////////////////////////////////////// ↗
252     ///          COMPUTE MOMENTS FOR EACH CONTOURS AND EXTRACT ONLY THE NORMALIZED 1ST ORDER ↗
253     ///          MOMENTS ///                                  ↗
254     /// //////////////////////////////////////// ↗
255     int N = contours.size();                                  // Easier and less confusing to use N ↗
256     // than calling the method every time
257
258     /// //////////////////////////////////////// ↗
259     ///          COUNT CONTOURS THAT HAVE ENOUGH            ↗
260     ///          POINTS                                     /// ↗
261     /// //////////////////////////////////////// ↗
262     /// NOTE: Tests determined that 5 points is the minimum to be sure that moments can ↗
263     // be computed correctly
264
265     int K = 0;
266     for(int i = 0; i < N; i++)
267         if (contours[i].size() >= 5)
268             K++;
269
270     /// //////////////////////////////////////// ↗
271     ///          RECAP OF THE                                ↗
272     ///          COUNT                                     /// ↗
273     /// //////////////////////////////////////// ↗
274
275     //cout << endl << "Total number of objects: " << N << endl;
276     //cout << endl << "Number of useful objects: " << K << endl;

```

```

275
276
277  /// //////////////////////////////////////  ↗
    ///
278  ///          FOR EACH USEFUL CONTOUR DETERMINES ALL  ↗
    MOMENTS          ///
279  /// //////////////////////////////////////  ↗
    ///
280
281  vector<Moments> mmt(K);          // Vector of objects of 'Moments'
282
283  /// Keep the following 4 instructions in mind and read the note below...
284  int j = 0;
285  for(int i = 0; i < N; i++)
286      if (contours[i].size() >= 5)
287          mmt[j++] = moments(contours[i], false);
288
289
290  /// //////////////////////////////////////  ↗
    ///
291  ///          NORMALIZED FIRST  ↗
    MOMENTS          ///
292  /// //////////////////////////////////////  ↗
    ///
293
294  /// NOTE: Necessary because the class 'Moments' doesn't have normalized first  ↗
    moments...
295  vector<Point2f> norm1stMoments(K);
296
297  for (int i = 0; i < K; i++)
298      norm1stMoments[i] = Point2f((mmt[i].m10 / mmt[i].m00), (mmt[i].m01 / mmt
    [i].m00));
299
300
301  /// NOTE: exactly the same for cycle is repeated twice: first, to compute K, and a  ↗
    second time to
302  /// calculate the moments. In the present implementation this is necessary  ↗
    because the value
303  /// K is used to declare 'mmt'.
304  /// The most elegant (and probably correct) method is allocating dynamically  ↗
    'mmnts'.
305
306
307  /// //////////////////////////////////////  ↗
    ///
308  ///          SCAN THROUGH COORDINATES OF EACH  ↗
    CONTOUR          ///
309  /// //////////////////////////////////////  ↗
    ///
310
311  // These are used to form the name of the ROI images.
312  string name = "ROI_";
313  //string suffix = ".png"
314  char numIm[21];
315  sprintf(numIm, "%d", k);
316  char numROI[21];
317  int n = 0;
318

```

```

319     int X = bin.cols, Y = bin.rows;           // Image sizes
320     int S = 10;                               // Half side of the ROI
321
322
323     /* Show which objects have been "seen" from the original image
324     IplImage * imgcpy1 = cvCloneImage(IPLimg);
325     IplImage * I1 = cvCreateImage(cvGetSize(imgcpy1), IPL_DEPTH_8U, 3);
326     cvCvtColor(imgcpy1, I1, COLOR_GRAY2RGB);
327     IplImage * imgcpy2 = cvCloneImage(IPLimg);
328     IplImage * I2 = cvCreateImage(cvGetSize(imgcpy2), IPL_DEPTH_8U, 3);
329     cvCvtColor(imgcpy2, I2, COLOR_GRAY2RGB);
330     unsigned short A = 8;*/
331     /// //////////////////////////////////
332
333     string Rname;
334
335     for (int i = 0; i < K; i++) {
336         // Discard ROIs that fall out of the image
337         if(norm1stMoments[i].x - S >= 0 && norm1stMoments[i].y - S >= 0 &&           ↗
            norm1stMoments[i].x + S < X && norm1stMoments[i].y + S < Y) {
338
339
340             //cvCircle(I1, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y), A,           ↗
                cvScalar(255,0,255), 1, 4);
341             /// //////////////////////////////////
342
343             // Definition of ROI
344             Rect rect(norm1stMoments[i].x - S, norm1stMoments[i].y - S, 2 * S, 2 * S);
345
346             // Creation of ROI image with reasonable size
347             Mat Roi, RoiCpy;
348             resize(img(rect), Roi, Size(400, 400), 1, 1, INTER_LANCZOS4);
349
350             //bilateralFilter(RoiCpy, Roi, 3, 1, 20);
351
352             if (!check(Roi)) {
353                 n++;
354                 Roi.release();
355                 continue;
356             }
357
358             //cvCircle(I2, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y), A,           ↗
                cvScalar(0,0,255), 1, 4);
359
360             // Ensure a coherent numeration (i.e. Use 'n' instead of 'i' as part of the ↗
                name)
361             sprintf(numROI, "%d", n);
362
363             double EV = analysis(Roi);
364
365             double R = sqrt(pow((norm1stMoments[i].x - xc), 2) + pow((norm1stMoments ↗
                [i].y - yc), 2));
366
367             Rname = name + numIm + "_" + numROI;
368
369             fileout << setfill(' ') << setw(16) << left << Rname;
370             fileout << " \t" << setw(7) << std::internal << R << "\t\t \t";
371             fileout << EV << "\t\t";

```

```

372         fileout << norm1stMoments[i] << "\n";
373
374         //string result;
375         //result = Rname + suffix;
376         //imwrite(result, Roi);
377         Roi.release();
378
379         n++;
380     }
381     //else
382         //cout << "Image # " << k << " - Discarded: " << norm1stMoments[i] << endl;
383 }
384
385 /*
386 string check1 = "Check";
387 string check2 = "Check_Result";
388 string checkName1 = check1 + numIm + suffix;
389 string checkName2 = check2 + numIm + suffix
390
391 cvSaveImage(checkName1.c_str(), I1);
392 cvReleaseImage(&imgcpy1);
393 cvReleaseImage(&I1);
394
395 cvSaveImage(checkName2.c_str(), I2);
396 cvReleaseImage(&imgcpy2);
397 cvReleaseImage(&I2);*/
398 /// //////////////////////////////////////
399
400 bin.release();
401 img.release();
402 bincpy.release();
403
404 }
405
406
407
408 bool check(Mat img) {
409     int hist[256];
410
411     // FAINTNESS
412     IplImage * I = cvCreateImage(cvSize(img.cols, img.rows), IPL_DEPTH_8U, 1);
413     IplImage Itemp = img;
414     cvCopy(&Itemp, I);
415     int c = findContrast(I, hist);
416
417     if(c < 9){
418         cvReleaseImage(&I);
419         return false;
420     }
421
422     // MULTIPLICITY
423
424     IplImage * cpy = cvCloneImage(I);
425
426     // Opening: 12 iterations
427     cvMorphologyEx(cpy, cpy, NULL, NULL, CV_MOP_OPEN, 12);
428     // Erosion: 10 iterations

```



```
429     cvErode(cpy, cpy, NULL, 10);
430     // Dilation: 3 iterations
431     cvDilate(cpy, cpy, NULL, 3);
432
433     calcHistogram(cpy, hist);
434     int B = findBackground(hist, cpy->height * cpy->width);
435
436     int t = 10;
437     int T;
438     IplImage * bin = NULL;
439
440     while(t <= 20) {
441         T = t + B;
442         bin = simpleThresholding(cpy, T);
443         if(multi(bin, hist)) {
444             cvReleaseImage(&I);
445             cvReleaseImage(&cpy);
446             cvReleaseImage(&bin);
447             return false;
448         }
449         t += 4;
450     }
451     return true;
452 }
453
454
455
456 bool multi(IplImage * img, int * hist) {
457     vector<vector<Point> > segm;
458     Mat imgM = cvarrToMat(img);
459     findContours(imgM, segm, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
460     imgM.release();
461
462     if(segm.size() > 1)
463         return true;
464     else
465         return false;
466 }
467
468
469
470 int findContrast(IplImage * img, int hist []) {
471     calcHistogram(img, hist);
472
473     int cnt = 0, TOT = (img->height * img->width);
474     int p5, p95;
475     bool f1 = true, f2 = true;
476
477     for(int i = 0; i < 256; i++) {
478         cnt += hist[i];
479         if(f1 && ((cnt * 100) / TOT) >= 5) {
480             p5 = i;
481             f1 = false;
482         }
483     }
484
485     if(f2 && ((cnt * 100) / TOT) >= 95) {
486         p95 = i;
```

```

487         f2 = false;
488     }
489 }
490
491 return (p95 - p5);
492 }
493
494
495 // Support function used by calcHistogram
496 void func(uchar c, int * h) {
497     unsigned x = c;
498     (*(h + x))++;
499 }
500
501
502 void calcHistogram(IplImage * gray, int hist []) {
503     /// Initialize histograms
504     for (int i = 0; i < 256; i++)
505         hist[i] = 0;
506
507     /// Calculate histogram
508     for(int i = 0; i < gray->height; i++) {
509         char * ptr = gray->imageData + i*gray->widthStep;
510         for(int j = 0; j < gray->width; j++) {
511             func((uchar)(*ptr), hist);
512             ptr++;
513         }
514     }
515 }
516
517
518
519 void drawHistogram(const char * histName, int * hist) {
520     int st = 4;           // To separate lines in the histograms, for better look ;)
521     int dep = 256;
522
523     /// Create image for the histogram
524     IplImage * his = cvCreateImage(cvSize(st * dep, 600), IPL_DEPTH_8U, 3);
525     cvSet(his, cvScalar(0,0,0));           // Initialize image (all black)
526     his->origin = IPL_ORIGIN_BL;           // Set the origin in the bottom left corner
527
528     for (int i = 0; i < 8 ; i++) {
529         int x = dep / 8;
530         cvLine(his, cvPoint(i * x * st, 0), cvPoint(i * x * st, 600), cvScalar
531             (122,122,122), 1, 4);
532     }
533
534     /// Draw the histogram
535     for (int i = 0; i < dep; i++)
536         if (hist[i] != 0)
537             cvLine(his, cvPoint(i * st, 0), cvPoint(i * st, hist[i] / 10), cvScalar
538                 (255, 0, 0), 1, 4);
539
540     cvSaveImage(histName, his);
541     cvReleaseImage(&his);
542 }

```

```
543
544
545 int findBackground(int hist[], int tot) {
546     bool fmin = true, fmax = true;
547     int m, M;
548
549     for (int i = 0; fmin || fmax; i++) {
550         if (fmin)
551             if(hist[i]){
552                 m = i;
553                 fmin = false;
554             }
555         if (fmax)
556             if(hist[255 - i]){
557                 M = 255 - i;
558                 fmax = false;
559             }
560     }
561
562     THR res_t;
563     res_t.B = m + (M - m) / 16;
564
565     int frac = 45, cnt = 0;
566
567     for(int i = m; i <= res_t.B; i++)
568         cnt += hist[i];
569
570     if( (res_t.x = ((cnt * 100) / tot)) == frac)
571         return res_t.B;
572     else if (res_t.x < frac) {
573         while(res_t.x < frac) {
574             res_t.B++;
575             cnt += hist[res_t.B];
576             res_t.x = (cnt * 100) / tot;
577         }
578         THR t1, t2;
579         t1.B = res_t.B;
580         t1.x = res_t.x;
581         t2.B = res_t.B - 1;
582         t2.x = ((cnt - hist[t2.B])* 100) / tot;
583         if (abs(t2.x - frac) < abs(t1.x - frac) )
584             return t2.B;
585
586         else
587             return t1.B;
588     }
589
590     else { // x > frac
591         while(res_t.x > frac) {
592             res_t.B--;
593             cnt -= hist[res_t.B];
594             res_t.x = (cnt * 100) / tot;
595         }
596         THR t1, t2;
597         t1.B = res_t.B;
598         t1.x = res_t.x;
599         t2.B = res_t.B + 1;
600         t2.x = ((cnt + hist[t2.B])* 100) / tot;
```

```
601         if (abs(t2.x - frac) < abs(t1.x - frac) )
602             return t2.B;
603         else
604             return t1.B;
605     }
606 }
607
608
609
610 IplImage * simpleThresholding(IplImage * img, unsigned th) {
611     IplImage * bin = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
612     for(int i = 0; i < img->height; i++) {
613         char * ptr = img->imageData + i*img->widthStep;
614         char * p = bin->imageData + i*bin->widthStep;
615         for(int j = 0; j < img->width; j++) {
616             if ((uchar)(*ptr) >= (uchar)th)
617                 *p = 255;
618             else
619                 *p = 0;
620             ptr++;
621             p++;
622         }
623     }
624     return bin;
625 }
626
```