Abstract—The goal of any biometric system consists in identifying individuals based on a certain characteristic possessed by the persons under examination. Among them, iris recognition is regarded as one of the most reliable and accurate biometric identification systems currently available. Most commercial iris recognition products use patented algorithms, which forces open source developers to design and use alternate algorithms. In this contribution, we propose two low-complexity methods for detecting and isolating the pupil and the iris using a greyscale image as the input data. The proposed algorithms can be easily implemented in any device, as they do not use complex operations or image transforms. In addition to that, we show a performance comparison which includes an implementation of our proposed algorithms and two other open source solutions.

Keywords: Biometric Identification, Iris, Pupil, Java, Security

1. Introduction

With the ever-growing need for reliable authentication mechanisms, biometric systems have experienced a great development in recent years. Due to their capability to perform the automatic and instant verification of an individual based on some specific physical characteristic, biometric security is now in a privileged position regarding other authentication solutions [1], [2].

There are several biometric technologies, based on different physical features: fingerprints, hand geometry, retina or iris scan, face recognition, voice analysis, etc. Among them, iris recognition has become one of the leading biometric technologies in our society, as the physical patterns of the iris are unique and can be obtained from some distance using the proper equipment [3].

John Daugman is credited as the developer of the first algorithms for iris recognition [4], and in 1994 he patented some of the algorithms that conform its foundations [5]. Even though Daugman’s patent expired in 2011 [6], most commercial biometric systems use other similarly patented algorithms, making necessary to develop alternative methods when implementing iris recognition applications.

In this contribution, we present a low-complexity method for locating the pupil and the iris in an image, so it can be processed afterwards for obtaining its associated code (also known as the iris template). We have implemented the proposed algorithms in Java and have compared its performance using two already existing open source solutions. The results from our tests allow us to state that our algorithms are faster and provide better identification rates than the other solutions considered.

This paper is organized as follows: Section 2 offers a brief introduction to iris identification. Section 3 presents two open source solutions available on the internet. Section 4 includes a complete description of our algorithms for pupil and iris location. In Section 5, we describe the Java application developed by us that implements the proposed algorithms. Section 6 contains the tests results of the three applications with the same iris database. Finally, Section 7 includes our conclusions about the new algorithms.

2. Foundations of iris identification

In the first layers of the eye there are different elements: cornea, iris, pupil, sclera, etc. The purpose of the iris is to constrict or enlarge the aperture of the pupil. By doing this, it determines the amount of light that enters the pupil [7]. Figure 1 shows the elements present in a front view of the eye.

![Fig. 1: View of the human eye (source: [8]).](image_url)
2) Pre-processing: detection of the iris prior to obtaining its normalized version, which is the result of transforming the image from polar to Cartesian coordinates.

3) Template generation: computation of the iris code based on the characteristics of the iris.

4) Feature comparison: calculation of a similarity score by comparing the user's iris code to other templates.

One of the most important prerequisites for performing authentications based on the iris consists in developing and using robust methods for the automatic detection and isolation of the pupil and the iris. Due to its regular size and uniform dark shade, the pupil is relatively easy to locate. However, locating the iris is not a trivial task due to its irregular pattern, the obstruction of the iris by the eyelids presented in some images, and the relative similarity of the iris and the sclera near the outer iris boundary [10].

3. Iris recognition applications

In this section we describe two iris recognition applications whose source code can be freely downloaded and inspected by developers.

3.1 Imperial College

The Iris Recognition Application from Project Iris is a free, open-source, cross-platform application developed using C++ and the Qt framework [11]. The application is the work of five Computing Science students at Imperial College London working under the supervision of Professor Duncan Gillies, and the source code is freely available under the GNU General Public Licence (GPL). Figure 2 shows a screenshot of the application after it has processed the image of and eye and located the pupil, the iris, and the eyelids.

![Fig. 2: Iris Recognition Application.](image1)

As it is stated in the project’s final report [10], its aim was to implement a working prototype of the techniques and methods used for iris recognition, and to test these methods on one of the databases of irides provided by the Chinese Academy of Sciences’ Institute of Automation (CASIA) [12].

The application automatically detects the pupil and the iris by first removing noise with a median filter, and then applying the Sobel operator and the Hough transform for edge detection. In addition to that, it allows users to manually locate the pupil and the iris using the mouse.

The authors state that in virtually any case where the iris boundary is correctly located by the program, the iris is subsequently identified, being a solid evidence the lack of false matches after 35,000 comparisons using the CASIA database [10].

As a disadvantage, their implementation provides an estimation of the location of the iris based on concentricity with the pupil which, among other factors, lowers the detection rate to around 70% [10].

3.2 Warsaw University of Technology

The Iris Recognition software, developed by Bernard Kobos and Piotr Zaborowski at the Warsaw University of Technology, is a freely available Java application [13] based on the work of Libor Masek presented in his Thesis [14].

The application is able to locate the iris and the pupil, excluding eyelids, eyelashes, and reflections. In order to do that, it uses the Hough transform before normalizing and filtering the iris region using a 1D Log-Gabor filter. The phase data associated to the iris patterns is then extracted and quantised with two bits per working point [13].

Regarding the comparison capabilities, this software uses the Minkowski distance. Figure 3 shows an screenshot of the application after detecting and isolating the iris and the pupil of one of CASIA’s test subjects.

![Fig. 3: Iris Recognition software.](image2)
4. Iris and pupil detection algorithms

After using the two applications commented in Section 3, we decided to create new algorithms for locating the pupil and the iris. As our detection procedures do not use image transforms or complex operations, they can be implemented in a broad range of devices with different computing capabilities.

Before presenting the algorithms, it is necessary to take into account the differences between the Cartesian system of coordinates and the system typically used in computer graphics, both represented in Figure 4. In the Cartesian system, coordinates increase to the right and up along the \( x \) and \( y \) axis, respectively. In comparison, computer graphics use by convention a coordinate system where the origin is in the upper-left corner, and the direction of the positive \( y \)-axis is downwards. Using this scheme, distances are measured in pixels, which are always integer values.

Another technical characteristic that must be taken into consideration is the format of the images. CASIA images are stored as 8-bit greyscale BMP (Windows bitmap) images. In that format, every pixel is encoded as a positive number ranging from 0 to 255, with 0 being pure black and 255 being pure white.

4.1 Pupil detection

Our pupil detection algorithm is a five-step procedure that basically computes a point acting as the centre of the circumference that represents the pupil and its associate radius, and then proceeds to adjust those values through a series of left-right and up-down movements.

Algorithm 1 provides the first approximation to the centre and radius of the pupil. The algorithm vertically scans the image and locates the longer sequence of at least 25 contiguous pixels whose value is below a certain threshold. The algorithm performs a loop until such a valid sequence is found, where each iteration uses a different limit value (starting at 5, and incremented in 5 units in each loop pass). Then, the algorithm identifies the middle point of the sequence in the scope of the image, which represents the first estimation for the pupil centre.

The following four steps of the procedure aim to improve the location of the pupil centre and the estimation of the radius. In order to do so, Algorithms 2 and 3 identify the values such that the four circumference points located at the far left, right, up, and down (from the point of view of the circumference centre) all have values smaller than the limit found in Algorithm 1, which means that the circumference is circumscribed in the pupil. In Algorithm 2, the one-directional movements proceed along the \( x \) axis (i.e., only left and right movements and reductions in the circumference radius are allowed). In comparison, the circumference can move along the \( y \) axis in Algorithm 3, so up and down movements and circumference reductions are the only operations allowed.

Algorithm 4 and 5 are quite similar to Algorithms 2 and 3. The difference consists in replacing the reduction operations with enlargement ones. The goal for those algorithms is to locate the circumference whose left, right, up, and down extreme points all have values bigger than the limit, which means that the circumference is acting as the external border of the pupil.

Once Algorithm 5 finishes its execution, the centre of the pupil circumference is represented as the point whose coordinates are pupilX and pupilY, while the diameter of the circumference is computed as pupilDiam.

![Fig. 4: Cartesian and computer graphics coordinate systems](source: [15])

```
Algorithm 1 Pupil detection (phase 1).
Require: image, height, width
1: count, pupilX, pupilY, pupilDiam ← 0
2: limit ← 5
3: repeat
4: for all i such that height < i < 3*height/4 do
5:     black ← false
6:     count, first, last, longFirst, longLast ← 0
7:     for all j such that width < j < 3*width/4 do
8:         val = image[i*width + j];
9:         if val < limit then
10:            count ← count + 1
11:            if black then
12:               last ← j
13:               else
14:               first ← j
15:               black ← true
16:         end if
17:         if count ≥ 25 then
18:             longFirst ← first
19:             longLast ← last
20:         end if
21:     end for
22:     if black then
23:         count ← 0
24:     end if
25:     end for
26:     if (longLast-longFirst) > 0 then
27:         dif ← (longLast-longFirst)
28:         if dif ≥ 0 then
29:             if pupilDiam ≤ dif then
30:                 pupilDiam ← dif
31:                 pupilX ← (longLast + longFirst)/2
32:                 pupilY ← i
33:             end if
34:         end for
35:         limit ← limit + 5
36:     until pupilDiam > 0
37: return pupilX, pupilY, pupilDiam, limit
```
4.2 Iris detection

Algorithm 6 contains all the logic needed to detect the outer border of the iris taking as input data the pupil centre and diameter provided by the previous phase, and the image. The core of the algorithm consists in working along a certain row (the one corresponding to the \( y \) value of the pupil’s centre, the one immediately before and the one immediately after), trying to locate the point in which a certain condition if fulfilled. When working with six consecutive groups of 8 pixels each, the condition if fulfilled when the average value of each of the three inner pixel groups (i.e., those closer to the pupil centre) is lower than the average value of each of the three outer pixel groups. The algorithm measures the distance form the pupil centre where that condition occurs when moving the groups of pixels first to the left and then to the right. The value of 8 pixels per group was the one that provided better results in our experiments.

Once the left and right distances have been taken in the three aforementioned rows, the bigger distance is taken, with the particularities described in Algorithm 6. If the left distance is not equal to the right distance, that means that the iris centre is not concentric to the pupil centre.

5. Iris Recognition Program

In order to test the practicability of the proposed algorithms and to compare the results to other solutions, we have developed a Java application using the software Java Development Kit (JDK) 1.6 update 27, with default parameters both for compiling and running the application. Figure 5 displays the main screen of our application, the Iris Recognition Program (IRP-CSIC).

This software allows the user to select a greyscale BMP image using the Select File button. Once the file is loaded, the application automatically tries to detect the pupil and iris boundaries, generating a code from the normalized image of the iris.

The application also allows to store the generated template in a database and to compare the current code to the ones stored in the database, though in the scope of this contribution that functionality is not used. With regards to that, the lines that intersect the circumferences in Figure 5 are used to mark the two lateral sectors which contain the iris data employed during the computation of the template.
6. Tests and results

The tests whose results are presented in this section were completed using a PC with Windows 7 Professional OS and an Intel Core i7 processor at 3.40 GHz.

6.1 Detection comparison

Table 1 includes the results of the tests performed with the three applications. The CASIA database used in those tests, called CASIA Iris Image Database Version 1.0 [12], contains 756 images from 108 different users. For each user, two iris recording sessions are provided (the first one producing 3756 images from 108 different users. For each user, two iris characteristics for the three cases:

1. For each element to be detected (pupil and iris), we have used the same batch of CASIA images belonging to 10 different users, so in each case a total of 70 images have been processed.
2. The time displayed for each element represents the average time of the 70 images processed.
3. For each individual test, the corresponding application has been started and closed, so every time a new test has been passed a fresh instance of the application has been executed.

In the case of IRP-CSIC, the tests include the following common characteristics for the three cases:

1. The Java function used for obtaining the timing is System.nanoTime().
2. In the pupil case, the start time has been taken exactly after the application retrieves the last candidate values provided by Algorithm 5.

In the pupil tests we have allowed a 5% error margin, whilst in the iris tests we have permitted an error margin of 10%. In the scope of these tests, the error is defined as the distance in pixels between the ideal circumference that represents the outer limit of the pupil (or, respectively, the iris), and the circumference drawn by the applications, divided by the diameter of the pupil (or the iris). That distance, which has been measured with the help of the Greenshot program [16], is taken in at most four points (left and right in all the cases, and up and down whenever those points were not covered by the eyelids).

<table>
<thead>
<tr>
<th>Application</th>
<th>Pupil</th>
<th>Iris</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRP-CSIC</td>
<td>754 (99.74 %)</td>
<td>683 (90.34 %)</td>
</tr>
<tr>
<td>Imperial College</td>
<td>745 (98.54 %)</td>
<td>538 (71.16 %)</td>
</tr>
<tr>
<td>Warsaw Tech. Univ.</td>
<td>411 (54.37 %)</td>
<td>381 (50.40 %)</td>
</tr>
</tbody>
</table>

Figures 6, 7, and 8 show three examples where our software outperforms the other open source solutions considered in this comparison in terms of iris and pupil detection. In those figures, the first image corresponds to IRP-CSIC, the second one to the Imperial College’s application, and the third one to the application developed at the Warsaw Technical University.

6.2 Running time

In order to measure the performance of our solution, we conducted a new test with the following common characteristics for the three cases:

1. For each element to be detected (pupil and iris), we have used the same batch of CASIA images belonging to 10 different users, so in each case a total of 70 images have been processed.
2. The time displayed for each element represents the average time of the 70 images processed.
3. For each individual test, the corresponding application has been started and closed, so every time a new test has been passed a fresh instance of the application has been executed.

In the case of IRP-CSIC, the tests include the following features:

1. The Java function used for obtaining the timing is System.nanoTime().
2. In the pupil case, the start time has been taken exactly before calling the application method that implements Algorithm 1, while the finish time has been taken after the application retrieves the last candidate values provided by Algorithm 5.

Algorithm 4 Pupil detection (phase 4).

Require: image, height, width, pupilX, pupilY, pupilDiam, limit
1: finished ← false
2: left ← false
3: right ← false
4: while not finished do
5:   radius ← diam/2
6:   west = image[pupilY+width+posX-radius];
7:   east = image[pupilY+width+posX+radius];
8:   if (east ≤ limit) or (west ≤ limit) then
9:     if ((east ≤ limit) and (west ≤ limit)) or (left and right) then
10:    pupilDiam ← pupilDiam+2
11:   else
12:     left ← false
13:     right ← false
14:   else
15:     if (west ≤ limit) and (east > limit) then
16:       pupilX ← pupilX -1
17:     else
18:       if (east ≤ limit) and (west > limit) then
19:         pupilX ← pupilX +1
20:     end if
21:     end if
22:   end if
23: end if
24: end if
25: finished ← true
26: end if
27: end while
28: return pupilX, pupilY, pupilDiam

Algorithm 5 Pupil detection (phase 5).

Require: image, height, width, pupilX, pupilY, pupilDiam limit
1: finished ← false
2: up ← false
3: down ← false
4: while not finished do
5:   radius ← pupilDiam/2
6:   north = image[pupilY+width+posX-radius];
7:   south = image[pupilY+width+posX+radius];
8:   if (south ≤ limit) or (north ≤ limit) then
9:     if ((south ≤ limit) and (north ≤ limit)) or (up and down) then
10:    pupilDiam ← pupilDiam+2
11:   else
12:     up ← false
13:     down ← false
14:   else
15:     if (north ≤ limit) and (south > limit) then
16:       pupilY ← pupilY -1
17:     else
18:       if (south ≤ limit) and (north > limit) then
19:         pupilY ← pupilY +1
20:     end if
21:     end if
22: end if
23: end if
24: end if
25: finished ← true
26: end if
27: end while
28: return pupilX, pupilY, pupilDiam
3) In the iris case, the start time and the stop time has been taken exactly before and after calling the method which implements Algorithm 6, respectively.

With regards to the Imperial College application, the most important characteristics of the tests are the following:

1) The C++ method used for obtaining the timing is `clock()`. The difference between the finish and start values given by that function has been divided by the value `CLOCKS_PER_SEC`, provided by the system, in order to obtain the time elapsed.

2) In the pupil case, the start time has been taken exactly before calling the method `findPupil()` that implements the pupil location algorithm, while the finish time has been taken immediately after that method returns.

3) In the iris case, the start time has been taken exactly before calling the method `findCircle()` that implements the iris location algorithm, while the finish time has been taken immediately after that method returns its estimation.

Finally, regarding the application from the Warsaw Technical University, the most relevant features are the following:

1) The Java function used for obtaining the timing is `System.nanoTime()`.

2) We have measured the time used by the Sobel and Hough methods and added half that time to both the pupil and the iris running time, as the location of both elements take advantage of the Sobel and Hough computations.

3) The Sobel and Hough start time has been taken before calling the method `sobelObject.init()`, while the stop time has been taken after the methods `sobelObject.process()`, `sobel()`, and `hough()` are executed.

4) In the pupil case, the initial running time comprises the execution of the `pupil()` method.

5) In the iris case, the initial running time comprises the execution of the `iris()` method.

6) In all the portions of code involved in the calculations, we have removed the printing methods which outputs information on the console in order to remove the delays added by the printing of information.

Table 2 includes the results obtained after completing the performance tests with the three applications. As it can be observed, our proposal is faster than the other solutions.

<table>
<thead>
<tr>
<th>Application</th>
<th>Pupil Time</th>
<th>Iris Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRP-CSIC</td>
<td>526.6 µs</td>
<td>454.0 µs</td>
</tr>
<tr>
<td>Imperial College</td>
<td>51.0 ms</td>
<td>29.3 ms</td>
</tr>
<tr>
<td>Warsaw Tech. Univ.</td>
<td>388.0 ms</td>
<td>359.9 ms</td>
</tr>
</tbody>
</table>

7. Conclusions

Iris identification is one of the most interesting applications of biometric techniques. As the algorithms used in most commercial applications are patented, developers must either pay for the corresponding licences or create new algorithms.
Require: image, height, width, pupilX, pupilY, pupilDiam

1: irisL1X, irisL1Y, irisL1Diam, irisL2X, irisL2Y, irisL2Diam ← 0
2: for all k such that −1 ≤ k ≤ 1 do
3:    good ← true, finished ← false
4:    if irisY < pupilY + k, x ← pupilX - pupilDiam/2
5:    irisDiam ← 0, diam ← 0
6:    repeat
7:        for all i such that 0 ≤ i < 8 do
8:            val1 ← val + image[irisY + width * x:0]
9:            val2 ← val + image[irisY + width * x:8]
10:           val3 ← val + image[irisY + width * x:16]
11:          val4 ← val + image[irisY + width * x:24]
12:         val5 ← val + image[irisY + width * x:32]
13:        val6 ← val + image[irisY + width * x:40]
14:    end for
15:   val1 ← val1/2, val2 ← val2/8, val3 ← val3/8
16:   val4 ← val4/8, val5 ← val5/8, val6 ← val6/8
17:   if (val1 < val4) and (val1 < val5) and (val1 < val6) and (val2 < val4)
18:      and (val2 < val6) and (val3 < val4) and (val3 < val5) and (val3 < val6)) then
19:      finished ← true
20:   end if
21: x ← x + 1
22: until finished or (x=48) < 0
23: finished ← false
24: x ← pupilX + diam/2
25: repeat
26:    for all i such that 0 ≤ i < length do
27:       val1 ← val1 + image[(irisY + width * x):+1]
28:      val2 ← val2 + image[(irisY + width * x):+8]
29:     val3 ← val3 + image[(irisY + width * x):+16]
30:    val4 ← val4 + image[(irisY + width * x):+24]
31:   val5 ← val5 + image[(irisY + width * x):+32]
32:  val6 ← val6 + image[(irisY + width * x):+40]
33: end for
34: val1 ← val1/length, val2 ← val2/length, val3 ← val3/length
35: val4 ← val4/length, val5 ← val5/length, val6 ← val6/length
36: if (val1 < val4) and (val1 < val5) and (val1 < val6) and (val2 < val4)
37: and (val2 < val6) and (val3 < val4) and (val3 < val5) and (val3 < val6)) then
38:    distRight= (x-pupilX) + 3*length) + length/2;
39:   finished ← true
40: end if
41: x ← x + 1
42: until finished or (x=48) ≥ width
43: if (distLeft = 0) and (distRight > 0) then
44:    distLeft ← distRight, good ← false
45: end if
46: if (distRight = 0) and (distLeft > 0) then
47:    distRight ← distLeft, good ← false
48: end if
49: if (distLeft > distRight) and ((distLeft-distRight)>18) then
50:    distRight ← distLeft, good ← false
51: end if
52: if (distRight > distLeft) and ((distRight-distLeft)>18) then
53:    distLeft ← distRight, good ← false
54: end if
55: if (distLeft > distRight) then
56:    irisX = pupilX -(distLeft-distRight)/2;
57: else
58:    irisX = pupilX + (distRight-distLeft)/2;
59: end if
60: if good and (irisDiam > bestL1Diam) then
61:   irisL1X ← irisX, irisL1Y ← irisY, irisL1Diam ← diam
62: end if
63: if (not good) and (irisDiam > irisL2Diam) then
64:   irisL2X ← irisX, irisL2Y ← irisY, irisL2Diam ← diam
65: end if
66: end if
67: if irisL1Diam > 0 then
68:   return irisL1X, irisL1Y, irisL1Diam
69: else if irisL2Diam > 0 then
70:   return irisL2X, irisL2Y, irisL2Diam
71: else
72:   return null
73: end if

In this contribution we have described in detail two low-complexity methods for locating the pupil and the iris which are patent-free and that, due to their simplicity, can be implemented in a broad range of devices with a good performance. After comparing our proposal to two other open source solutions, we can state that the application that implements our algorithms is faster and more accurate than the other solutions considered in the comparison.

Even though the iris detection is a complex issue, we consider that the lower average running time (when compared to the average running time for pupil detection) in our application is due to the following facts:

- The pupil identification procedure generates five calls to other Java methods, whilst the iris identification procedure generates three calls. Calling a Java method increments the overall running time.
- The iris identification procedure takes advantage of the work done by the pupil identification procedure, as it starts its execution with the knowledge of the point which represents the center of the circle which simulates the pupil.

Acknowledgment

This work has been partially supported by Ministerio de Ciencia e Innovación (Spain) under the grant TIN2011-22668.

References