Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Questions 1 & 2

Submitted by

Parth T — Tirth

Submitted on May 5, 2024

1. **Real-Time Analysis: For your 2+ services Si with requirements Ci, Ti, Di provide Cheddar Simulation and Worst-Case analysis, but double check this with your own Scheduling Point and Completion Test analysis as well as hand-drawn timing (if feasible – i.e. if LCM is not ridiculously long). Provide an estimate of the safety margin your system should have compared to what it must have for feasibility and margin of error [variations seen in Ci to account for expected Ci and WCET].**

To meet the overall system deadline of 152 milliseconds, each service within the Hawkeye system must adhere to specific deadlines and execution times. The following real-time requirements define the timing constraints for each service:

- **RTR2.1**: The frame capture, face detection, and face processing service shall have a deadline (D1) of 52 milliseconds and a period (T1) of 50 milliseconds. This requirement ensures that the system can process frames at a rate of 20 FPS, providing smooth and responsive face tracking.
- **RTR2.2**: The execution time (C1) of the frame capture, face detection, and face processing service should be less than 50 milliseconds.
- **RTR2.3**: The servo actuation service shall have a deadline (D2) of 50 milliseconds and a period (T2) of 100 milliseconds. We have given this deadline to divide the overall response time to 152 ms.
- **RTR2.4**: The execution time (C2) of the servo actuation service shall be within 3-4 milliseconds. This requirement ensures that the servo actuation service can complete its tasks efficiently and leave enough time for other services to execute within the overall system deadline.
- **RTR2.5**: The servo shooting service shall have a deadline (D3) of 50 milliseconds and a period (T3) of 100 milliseconds. This requirement allows the servo shooting service to coordinate with the servo actuation service and trigger the laser pointer at the appropriate time.
- **RTR2.6**: The execution time (C3) of the servo shooting service shall be within 3-4 milliseconds. This requirement guarantees that the servo shooting service can perform its tasks quickly and efficiently, minimizing any additional latency in the system.

The deadlines and execution times for each service are carefully chosen based on the overall system deadline of 150 milliseconds and the interdependencies between the services. These requirements ensure that the system can perform all necessary tasks within the allocated time budget, providing a responsive and real-time user experience. Table summarizes the deadlines, periods, and execution times for each service in the system. These values are derived based on the real-time requirements and the system's overall timing constraints.

Table 1: Service Deadlines and Execution Times (Proposed)

Service	Deadline (ms)	Period (ms)	Execution Time (ms)
Frame Capture, Face Detection	52	50	35
Servo Actuation	50	50	3
Servo Shooting	50	50	3

To validate the feasibility of the Hawkeye system and ensure that it meets its real-time requirements, a thorough analysis of the system's timing properties is essential. The following real-time requirements pertain to system feasibility and analysis:

We have done cheddar analysis for the task with Rate monotonic scheduler, According to cheddar the system is feasible.

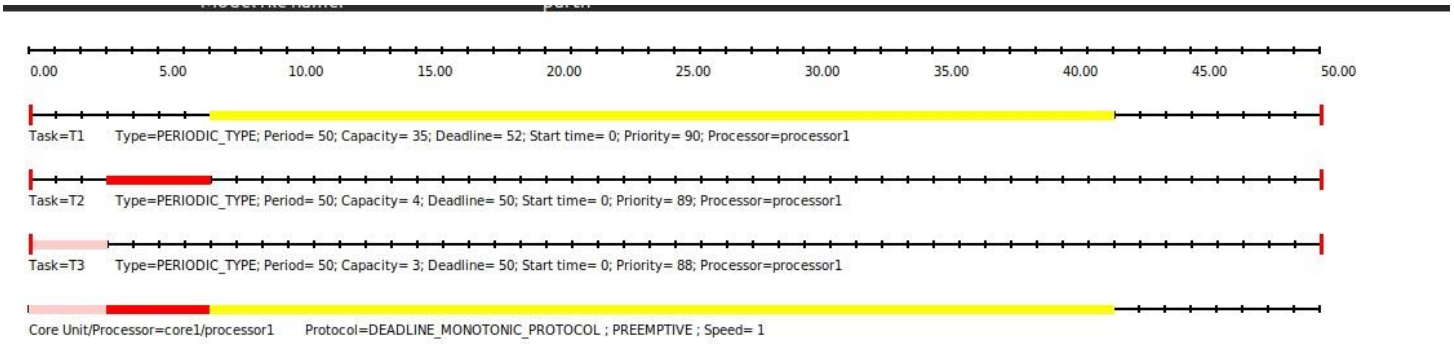

Figure: Scheduling output

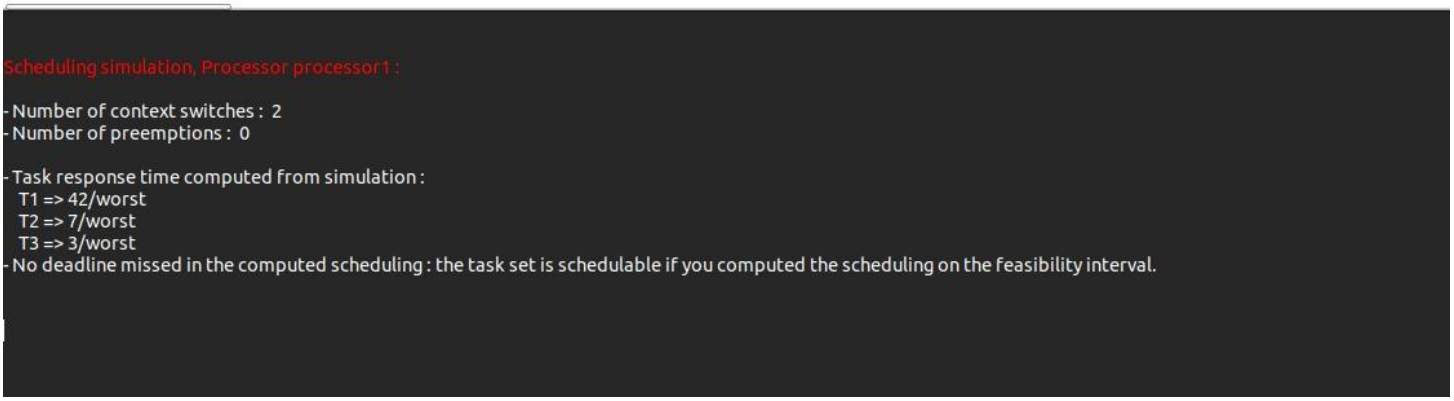

Figure: Cheddar Scheduling

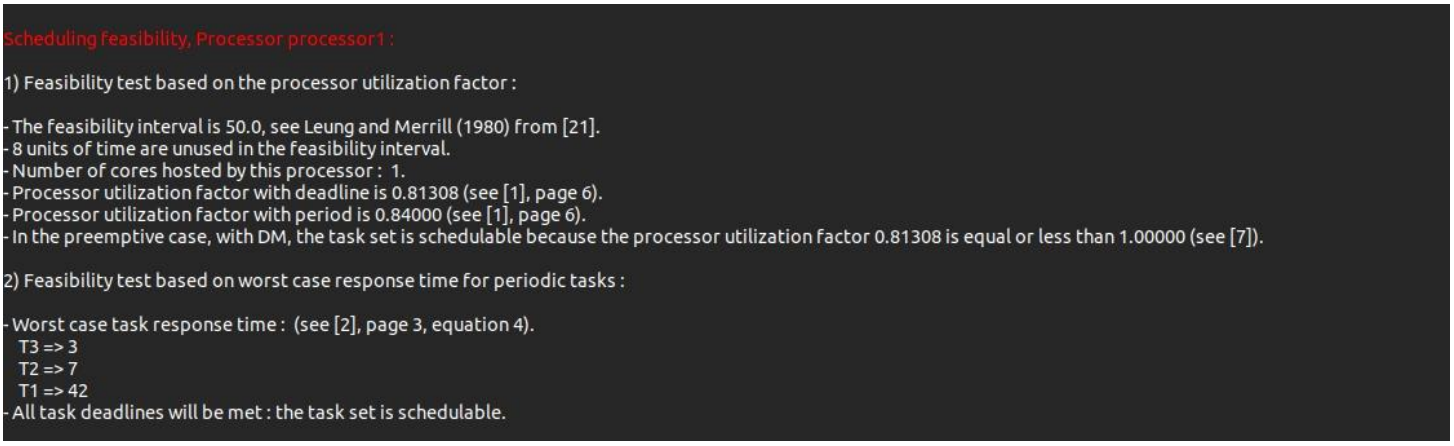

Figure: Feasibility Test cheddar

- We have done Scheduling Point analysis and Completion time feasibility with help of Excercise 2 code. According to necessary and sufficient test analysis we can say that the system is feasible.

```
C: 35 4 3
T: 50 50 50
D: 52 50 50

Task 0, WCET=35, Period=50, Utility Sum = 0.700000
Task 1, WCET=4, Period=50, Utility Sum = 0.780000
Task 2, WCET=3, Period=50, Utility Sum = 0.840000

Total Utility Sum = 0.840000
LUB = 0.779763
RM LUB: Infeasible
Completion time feasibility: Feasible
Scheduling point feasibility: Feasible
Deadline monotonic: Feasible

(Period)
Total utility in EDF: 0.840000 Which is less than 1.0
EDF on Period: Feasible
Total utility in LLF: 0.840000 Which is less than 1.0
LLF on Period: Feasible

(Deadline)
Total utility in EDF: 0.813077 Which is less than 1.0
EDF on Deadline: Feasible
Total utility in LLF: 0.813077 Which is less than 1.0
LLF on Deadline: Feasible
```

Table 2: Service Deadlines and Execution Times (Obtained)

SERVICE	SERVICE NAME	WCET	Proposed Ci	Obtained Ci	Di	Ti
S1	Face Recognition Service	26.75 ms	35 ms	25 ms	52ms	50ms
S2	Servo Actuation Service	0.24 ms	4 ms	0.2 ms	50ms	50ms
S3	Servo Aiming Service	0.2 ms	3 ms	0.1 ms	50ms	50ms

Safety Margin Analysis:

Task 1 (T1): C1=35ms, D1=52ms, T1=50ms
Task 2 (T2): C2=4ms, D2=50ms, T2=50ms
Task 3 (T3): C3=3ms, D3=50ms, T3=50ms

Obtained Values:
Average Computation Times (Avg Ci):
C1= 25ms
C2= 0.2ms
C3= 0.1ms

Worst Case Execution Time (WCET):
C1= 26.75ms
C2= 0.24ms
C3= 0.2ms

Total Deadline:
The total deadline for all tasks is given as 152ms, which is the sum of D1, D2, and D3 (52ms + 50ms + 50ms).

Safety Margin Estimation:
Estimated vs Obtained Computation Time (Ci):

Task 1:
Proposed C1: 35ms
Obtained Avg C1: 25ms
Obtained WCET C1: 26.75ms
Safety Margin: The proposed computation time has a margin since the obtained WCET (26.75ms) is lower than the proposed (35ms). The system performs better than expected by 8.25ms.

Task 2:
Proposed C2: 4ms
Obtained Avg C2: 0.2ms
Obtained WCET C2: 0.24ms
Safety Margin: Significant margin as the WCET is substantially lower than the proposed time, suggesting that the system has a lot of leeway (3.76ms).

Task 3:
Proposed C3: 3ms
Obtained Avg C3: 0.1ms
Obtained WCET C3: 0.2ms
Safety Margin: Similar to Task 2, there's a significant margin of 2.8ms, indicating much better performance than anticipated.

System Deadline vs Task Execution:

The system's total deadline is 152ms, and the sum of the worst-case execution times for all tasks is 27.19ms (26.75ms + 0.24ms + 0.2ms). This indicates that the tasks are well within the deadline, providing a substantial safety margin in terms of time, which ensures that the system can handle additional load or delays in other processes.

The obtained computation times being significantly lower than the proposed values indicate efficient task execution and robust system design. The system is well within safety limits, suggesting potential capacity for more tasks or higher complexity without breaching the total deadline. This analysis supports both system reliability and the potential for scalability.

2. Real-Time Prototype Build: Implement at least 2 services from the overall service set design and execute them on a TI Tiva TM4C or De1-SoC using FreeRTOS or Cyclic Executive; or on a De1-SoC, Raspberry Pi or Jetson board with Linux using SCHED_FIFO; trace request time and completion time with a continuous periodic request for each service using fixed priority preemptive scheduling (and processor core affinity if applicable). Present the results along with a description of how predictable the responses are relative to the request times as well as how constant the request frequency is for your system design. Use of other boards or RTOS require written approval from the instructor.

To ensure the system meets its real-time requirements and achieves optimal performance, careful task allocation and scheduling are necessary. The following real-time requirements address task allocation and scheduling:
The sequencer is running at 50 Hz with the highest priority, giving a semaphore to each service every 50 ms. The frame capture has the highest priority, while servo actuation has lower priority, and the servo shoot service has even lower priority.

- **RTR3.1**: The frame capture, face detection, and face processing tasks shall be allocated to Core 0 of the processor. This requirement ensures that these computationally intensive tasks have dedicated resources and can execute without interference from other tasks.
- **RTR3.2**: The servo actuation and servo shooting tasks shall be allocated to Core 1 of the processor.
- **RTR3.3**: The frame capture, face detection, and face processing tasks shall have the highest priority among all tasks.

- **RTR3.4**: The servo actuation task shall have a lower priority than the frame capture, face detection, and face processing tasks, but higher than the servo shooting task. This requirement ensures that the servo actuation task executes after the face detection tasks and provides the necessary data for the servo shooting task.
- **RTR3.5**: The servo shooting task shall have the lowest priority among all tasks. As it depends on servo actuation service to be executed first

Figure illustrates the task allocation and scheduling in the Hawkeye system. The frame capture, face detection, and face processing tasks are allocated to Core 0, while the servo actuation and servo shooting tasks are allocated to Core 1. The priorities of the tasks are assigned based on their criticality and dependencies.

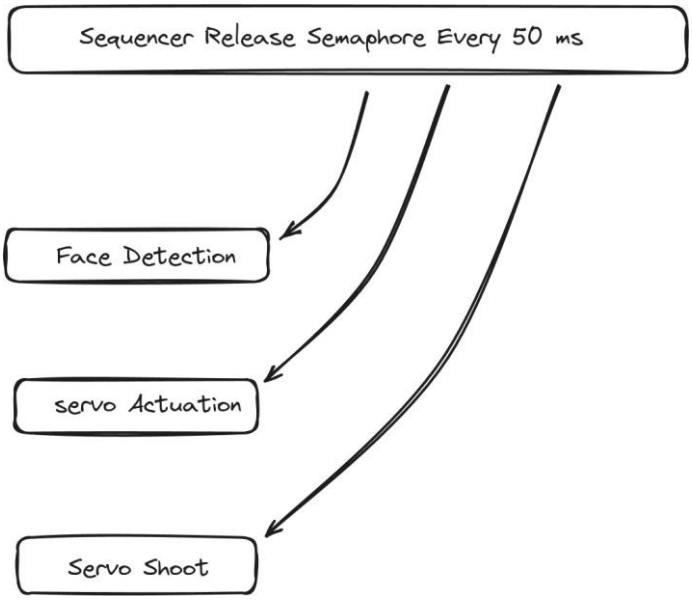

Figure: Task Allocation via sequencer

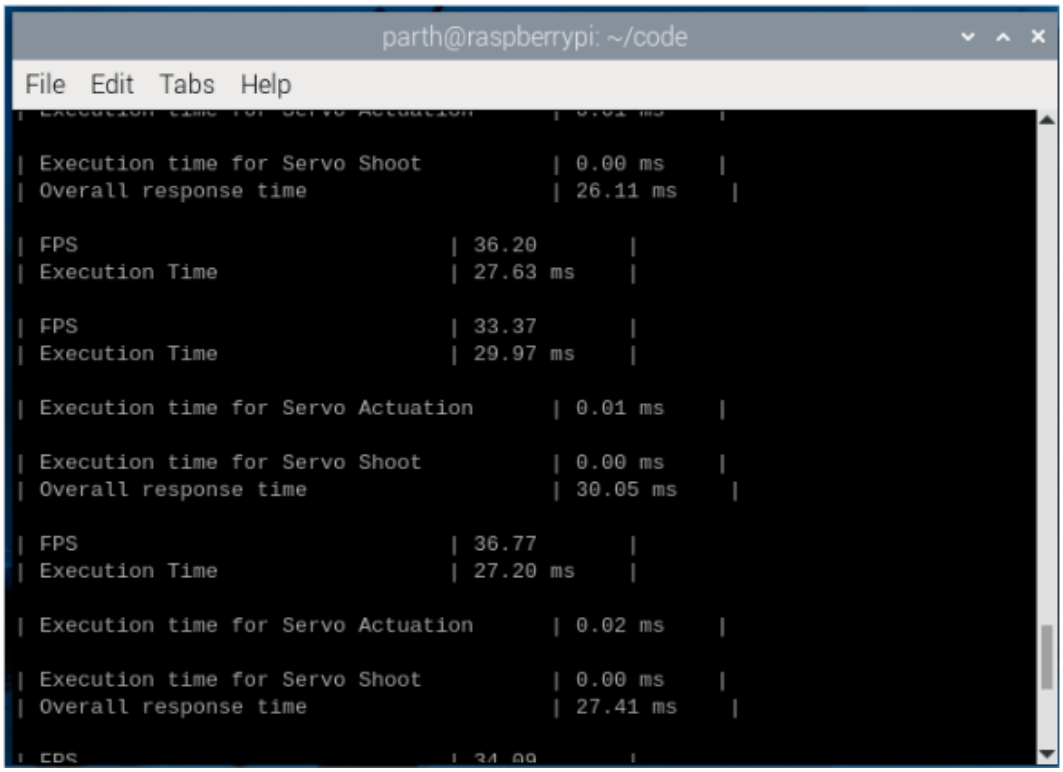

Figure: Log statements of execution time and FPS after running each set of services

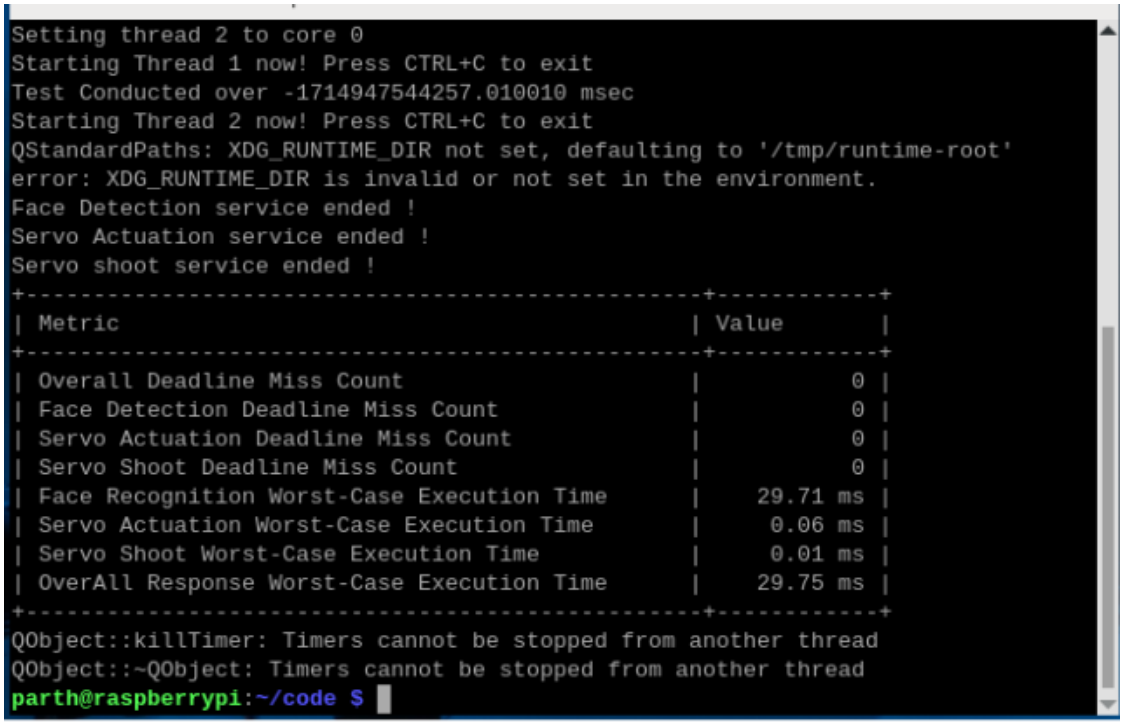

Figure: Log statements showing the WCET of each service and an overall response Worst-Case Execution Time

System Response and Execution Times:

Execution Times:
Servo Actuation: Executes in approximately 0.01 to 0.02 ms.
Servo Shoot: Executes in 0.00 ms.
Overall Response Time: Varies from 26.11 ms to 30.05 ms.
Frames Per Second (FPS) and Execution Time: FPS ranges from 33.37 to 36.77.
Corresponding Execution Times range from 27.20 ms to 29.97 ms.

Metrics and Results from System Testing:
Overall Deadline Miss Count: 0 — This indicates that all tasks were completed within their respective deadlines.

Worst-Case Execution Time (WCET) for Key Components:
Face Recognition: 29.71 ms
Servo Actuation: 0.06 ms
Servo Shoot: 0.01 ms
Overall Response: 29.75 ms

Analysis of Predictability and Request Frequency:
Predictability:
The predictability of a real-time system is primarily assessed by its ability to consistently meet deadlines and maintain a stable worst-case execution time (WCET). In your results, the overall deadline miss count is zero, which strongly indicates that the system is highly predictable.
The WCET for each component remains consistent across different test runs, which supports the reliability and predictability of the system under varying conditions.

Constancy of Request Frequency:
The FPS values show minor variations (33.37 to 36.77), suggesting a mostly stable but slightly variable request frequency. The variation in FPS might be due to the processing load or other system activities impacting the frame generation rate.
The execution time related to these FPS values indicates that while there might be variability in how frames are processed, the system handles these variations without missing deadlines.
Technical Detail on System Design:
Hawkeye efficiently manages the complex tasks of face detection, servo actuation, and servo shooting without

compromising on performance. The use of efficient algorithms is optimized for the Raspberry Pi in use, and effective scheduling strategies (evidenced by no missed deadlines) all contribute to this high level of predictability and performance consistency. The real-time operating constraints are well-maintained, with all tasks completing well within the overall system deadline, which sums up to 152 ms.

By allocating tasks to specific processor cores and assigning appropriate priorities, the system can ensure that each task receives the necessary resources and execution time to meet its real-time requirements. This allocation and scheduling scheme helps to optimize the system's performance, minimize interference between tasks, and guarantee the timely execution of critical tasks.

facedetect.cpp

```
// CPP program to detects face in a video 2
// Include required header files from OpenCV directory
#include "opencv2/objdetect.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>

using namespace std;
using namespace cv; 11
// Function for Face Detection
void detectAndDraw( Mat& img, CascadeClassifier& cascade,
                    CascadeClassifier& nestedCascade, double scale );
string cascadeName, nestedCascadeName; 16
int main( int argc, const char** argv )
{
    // VideoCapture class for playing video for which faces to be detected
    VideoCapture capture;
    Mat frame, image; 22
    // PreDefined trained XML classifiers with facial features
    CascadeClassifier cascade, nestedCascade;
    double scale=1;

    // Load classifiers from "opencv/data/haarcascades" directory
    nestedCascade.load( "./haarcascade_eye_tree_eyeglasses.xml" ) ; 29
    // Change path before execution
    cascade.load( "./haarcascade_frontalcatface.xml" ) ; 32
    // Start Video..1) 0 for WebCam 2) "Path to Video" for a Local Video
    capture.open(0);
    if( capture.isOpened() )
    {
        // Capture frames from video and detect faces
        cout << "Face Detection Started...." << endl;
        while(1)
        {
            capture >> frame;
            if( frame.empty() )
                break;
            Mat frame1 = frame.clone();
            detectAndDraw( frame1, cascade, nestedCascade, scale );
            char c =   (char)waitKey(10);

            // Press   q to exit from window
            if( c ==   27 || c == 'q' || c == 'Q' )
                break;
        }
    }
    else
```

```
            cout<<"Could not Open Camera";
        return 0;
}
```

```
void detectAndDraw( Mat& img, CascadeClassifier& cascade,
                    CascadeClassifier& nestedCascade,
                    double scale, bool tryflip )
{
    vector<Rect> faces, faces2; Mat gray, smallImg;

    cvtColor( img, gray, COLOR_BGR2GRAY ); // Convert to Gray Scale
    double fx = 1 / scale;

    // Resize the Grayscale Image
    resize( gray, smallImg, Size(), fx, fx, INTER_LINEAR ); equalizeHist( smallImg, smallImg );

    // Detect faces of different sizes using cascade classifier cascade.detectMultiScale( smallImg, faces, 1.1, 2, 0|CASCADE_SCALE_IMAGE, Size(30, 30) );

    // Draw circles around the faces
    for ( size_t i = 0; i < faces.size(); i++ )
    {
        Rect r = faces[i]; Mat smallImgROI;
        vector<Rect> nestedObjects;
        Point center;
        Scalar color = Scalar(255, 0, 0); // Color for Drawing tool
        int radius;

        double aspect_ratio = (double)r.width/r.height;
        if( 0.75 < aspect_ratio && aspect_ratio < 1.3 )
        {
            center.x = cvRound((r.x + r.width*0.5)*scale); center.y = cvRound((r.y + r.height*0.5)*scale); radius = cvRound((r.width + r.height)*0.25*scale); circle( img, center, radius, color, 3, 8, 0 );
        }
        else
            // rectangle( img, cvPoint(cvRound(r.x*scale), cvRound(r.y*scale)), cvPoint(cvRound((r.x + r.width-1)*scale), cvRound((r.y + r.height-1)*scale)), color, 3, 8, 0);
        if( nestedCascade.empty() )
            continue;
        smallImgROI = smallImg( r );

        // Detection of eyes in the input image nestedCascade.detectMultiScale( smallImgROI, nestedObjects, 1.1, 2, 0|CASCADE_SCALE_IMAGE, Size(30, 30) );

        // Draw circles around eyes
        for ( size_t j = 0; j < nestedObjects.size(); j++ )
        {
            Rect nr = nestedObjects[j];
            center.x = cvRound((r.x + nr.x + nr.width*0.5)*scale);
```

```
107                 center.y = cvRound((r.y + nr.y + nr.height*0.5)*scale); radius =
108                 cvRound((nr.width + nr.height)*0.25*scale); circle( img, center,
109                 radius, color, 3, 8, 0 );
110             }
111         }
112 
113         // Show Processed Image with detected faces imshow( "Face
114         Detection", img );
115 }
116 
```