

ECEN 5623 Real Time Embedded Systems

Final Report

Object Contour Detection and Tracking system

Team Members

Vignesh Jayaram

vignesh.jayaram@colorado.edu

Vihanga Bare

vihanga.bare@colorado.edu

Vishal Vishnani

vishal.vishnani@colorado.edu

Date

May 3, 2018

Introduction	3
System Block Diagram	3
Functional (Capability) Requirements	4
Real-Time Requirements	5
Functional Design Overview and Diagrams	6
Real-Time Analysis and Design with Timing Diagrams	8
WCET and ACET simulation	8
Timing analysis with ACET and WCET	8
Timing analysis with Ci, Ti and Di	9
Scheduling Point / Completion tests	10
Safety margin analysis	10
Proof-of-Concept with Example Output and Tests Completed	12
Resources used	12
Usage of Openmp for achieving parallelism	12
Time-stamp tracing	12
Test cases	14
Test Case 1	14
Test Case 2	15
Test Case 3	16
Test Case 4	16
Test Case 5	17
Conclusion	17
Formal References	18
Acknowledgements	18
Appendices	18

Introduction

The aim of this project is to use techniques in image processing to accomplish detection of contours in an image around a certain object and then track the object in real time. The technique used is to separate certain areas of image based on a particular color threshold, saturation and hue selected, detect contours to recognize an object and once detected track the object in real time. We are using contour detection for separating image from its background. Once detected, we calculate center and area of the detected object, pass this as a feature to Kalman filter and then use kalman filter for tracking the object.

Our project aims at using these image processing techniques in real world scenario like processing a live video feed for object detection and tracking. These applications have an average frame rate ranging from 5 - 7 frames per second. Also our system can track objects walking/rolling with a average speed range of 1.3 to 1.6 meters per second.

System Block Diagram

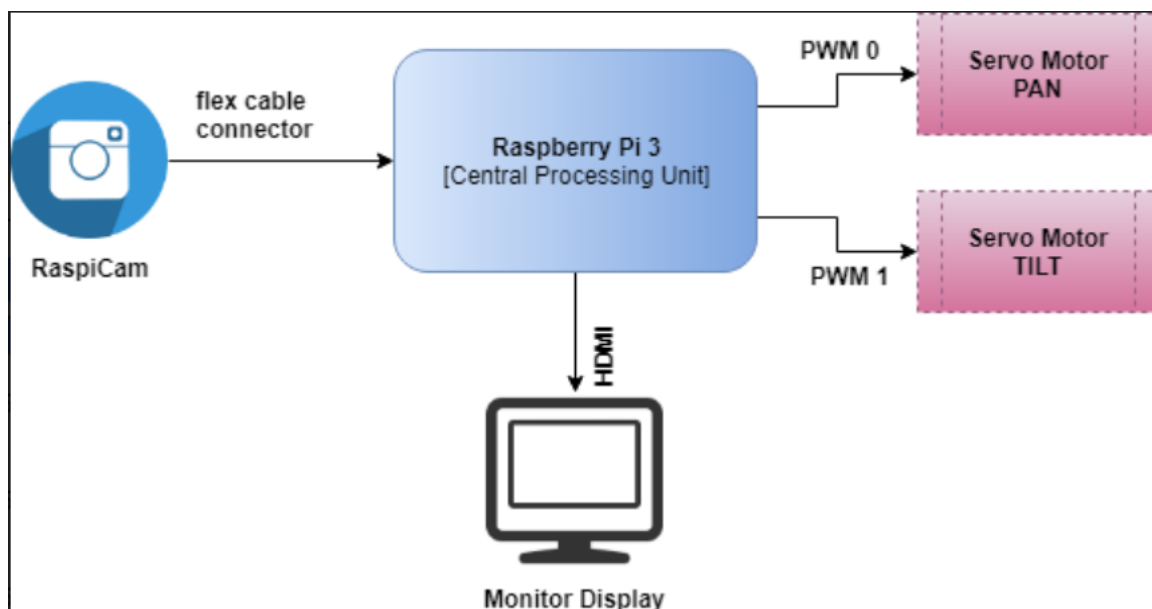


Fig 2.1 System Block Diagram

Functional (Capability) Requirements

Following are the 5 major functionality requirements of our system:

1. **Capture camera image:** The system should interface successfully with the camera, able to capture still as well as continuous video feed and be able to display the video output to the display with minimal jitter and lag in these operations. Also the image captured by this camera should be extracted successfully in OpenCV so that it can be used for further processing.
2. **Pre-processing of the captured image frame:** The captured image needs to be passed through certain filters in OpenCV so as to be able to separate the object from its background. Our system needs to remove extra noisy areas, patches, shadows so as to make the captured frame even, we use Gaussian Blur function for this to remove extremely high frequency areas by blurring the image. As we are detecting object based on its color, we are converting the captured image from RGB to HSV color space. Taking the HSV color space image and adjusting the hue, saturation levels, further applying erode and dilate functions to enhance the detected object in foreground and be able to separate the object from its background. After applying all the aforementioned pre processing filters, our final output should clearly depict the detected object separated from its background.
3. **Contour Detection:** Once obtained a successful output after thresholding, our system needs to find contours in the resulting black and white image, using these contours find area lying under each contours, thus with the calculation that the color which has lesser of the two areas is our object. Based on above calculation of area, it also calculates centroid of the detected object.
4. **Kalman Filter for object tracking:** Based on area, centroid and color value as three features of an object to track, our system uses Kalman filter algorithm, creates a current state matrix for Kth frame and based on updated matrix for K+1 th frame it perform object tracking. It draws a green rectangle to encompass the correct position of object and displays this to output console. Our system should be able to display both these outputs successfully and superimpose it on the live video.
5. **Motor actuation:** Starting with a predetermined field of vision, and based on the centroid values calculated in 4, we determine the position of object with respect to the frame boundaries. Depending on this position, we move the camera using two servo motors for pan and tilt movements so that we able to keep the object we are tracking within our field of vision.

Real-Time Requirements

Following are the three real time service requirements in our project:

- 1. Capture camera image and pre processing:** This service captures still as well as continuous video feed and displays the video output to the display with minimal jitter and lag in these operations. The captured image is passed through certain filters in OpenCV. To remove extra noisy areas, patches, shadows so as to make the captured frame even, we use Gaussian Blur function for this to remove extremely high frequency areas by blurring the image.
As we are detecting object based on its color, we are converting the captured image from RGB to HSV color space. Taking the HSV color space image and adjusting the hue, saturation levels, we apply erode and dilate functions to enhance the detected object in foreground and separate the object from its background. After applying all above preprocessing filters, our final threshold output image should clearly depict the detected object separated from its background. The service should finish within 185 msecs and is repeated every 215 msecs.
- 2. Contour Detection and Kalman filter for object tracking:** Once obtained a successful output after thresholding, our this service used findContours() function in OpenCV to detect contours. Using these contours it finds area underlying each contour. It calculates area for each contour and classifies the lesser of the two areas as our object. Using calculated area, it also derives centroid of the detected object.
Taking area, centroid and color value as three features of an object to track, this service applies Kalman filter algorithm to perform object tracking. It draws a green rectangle to encompass the correct position of object and displays this to output console. The deadline for this service is 10 msecs and period is 215 msecs.
- 3. Drive Motor actuation:** Starting with a predetermined field of vision, and based on the centroid values calculated in 4, this service determines the position of object with respect to the frame boundaries. Depending on this position, the service moves the camera using two servo motors for pan and tilt movements so that we are able to keep the object we are tracking within our field of vision. This service has deadline of 20 ms and period is 215 msecs.

Functional Design Overview and Diagrams

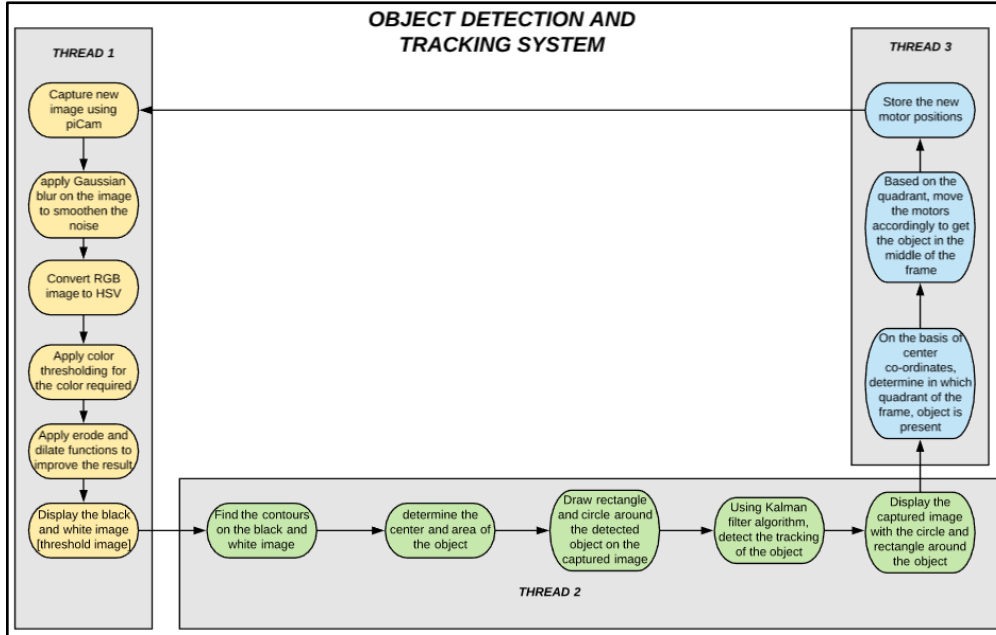


Fig 1 Software Flow diagram

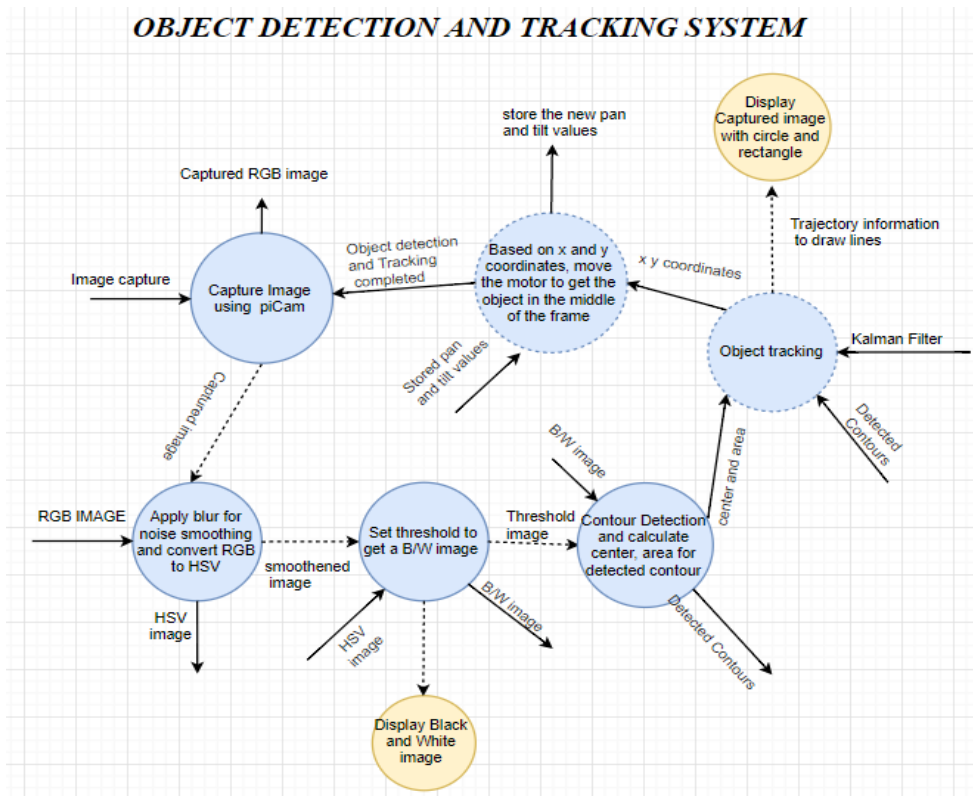


Fig 2 Control Flow diagram

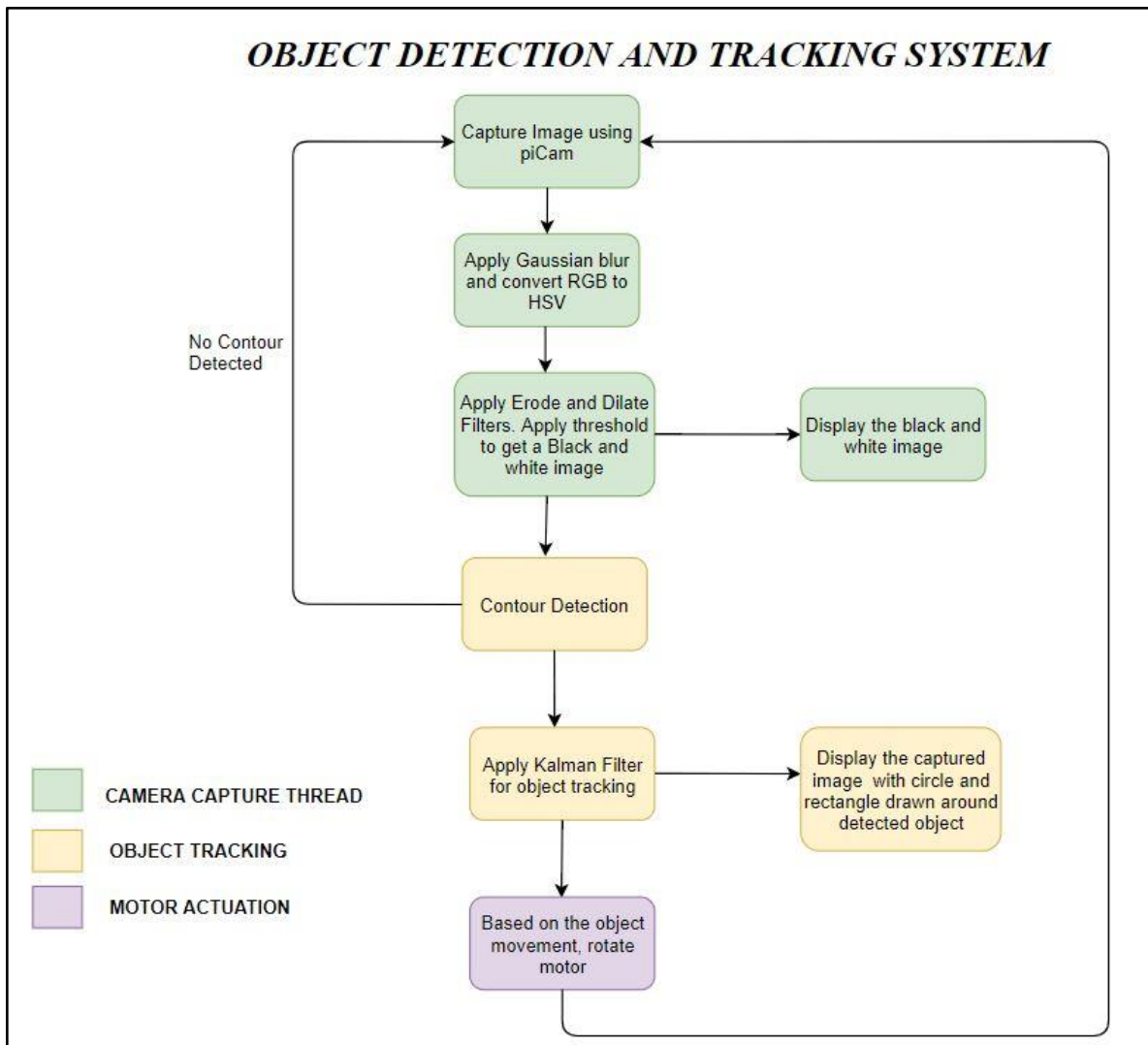


Fig 3 State Machine diagram

Real-Time Analysis and Design with Timing Diagrams

WCET and ACET simulation

```
pi@raspberrypi:~/RTES_FINAL_PROJECT/source$ make
g++ opencv_kalman.cpp -o asl -lopencv_calib3d -lopencv_core -lopencv_features2d -lopencv_flann -lopencv_highgui -lopencv_imgproc -lopencv_ml -lopencv_objdetect -lopencv_stitching -lopencv_superres -lopencv_video -lopencv_videoio -lopencv_videostab -lraspicam -lraspicam_cv -lrt -lpthread -l wiringPi -fopenmp
pi@raspberrypi:~/RTES_FINAL_PROJECT/source$ sudo ./asl
[4 procs]
thread 1 created successfully
thread 2 created successfully
thread 3 created successfully
Hit Ctrl-C to exit...
REAL TIME FINAL PROJECT ANALYSIS FOR [100] frames
EXEC PER FRAME = [201.398132 msecs]
(S1) ACET = [177.499059 msecs]
(S1) WCET = [451 msecs]
(S2) ACET = [6.980198 msecs]
(S2) WCET = [11 msecs]
(S3) ACET = [18.049999 msecs]
(S3) WCET = [362 msecs]
.....main end.....
pi@raspberrypi:~/RTES_FINAL_PROJECT/source$
```

Fig. 4 WCET and ACET simulation

Service 1 (capthread)

The execution time for this service involves the time required for capturing a frame, and pre-processing which involves filtering, blurring and converting from RGB to HSV. The ACET for this service is around 177 milliseconds and WCET is 451 milliseconds.

Service 2 (tracker thread)

The ACET for this service is 7 milliseconds and WCET is around 11 milliseconds which involves the time required for contour detection and applying Kalman Filter Algorithm.

Service 3 (motor thread)

This service performs the work of rotating servo motors depending on the current X and Y coordinate. Also, tracks the centroid of the object and draws a trajectory. The WCET for this service is 18 milliseconds and ACET is around 362 milliseconds which just involves the time required for applying PWM signal to the servo motors.

Timing analysis with ACET and WCET

Services	ACET (milliseconds)	WCET (milliseconds)
capthread	177	451
trackerthread	7	11
motorthread	18	362

Timing analysis with Ci, Ti and Di

Services	Execution Time (Ci) (milliseconds)	Deadline (Di) (milliseconds)	Period (Ti) (milliseconds)
capthread	177	185	215
trackerthread	7	10	215
motorthread	18	20	215

For our cyclic executive system we have taken same period for all the services which is the sum of their individual Average Case Execution Time (ACET) plus some margin. As the system we have designed is to be used soft real-time applications, we have taken ACET for the calculations.

Cheddar Simulation

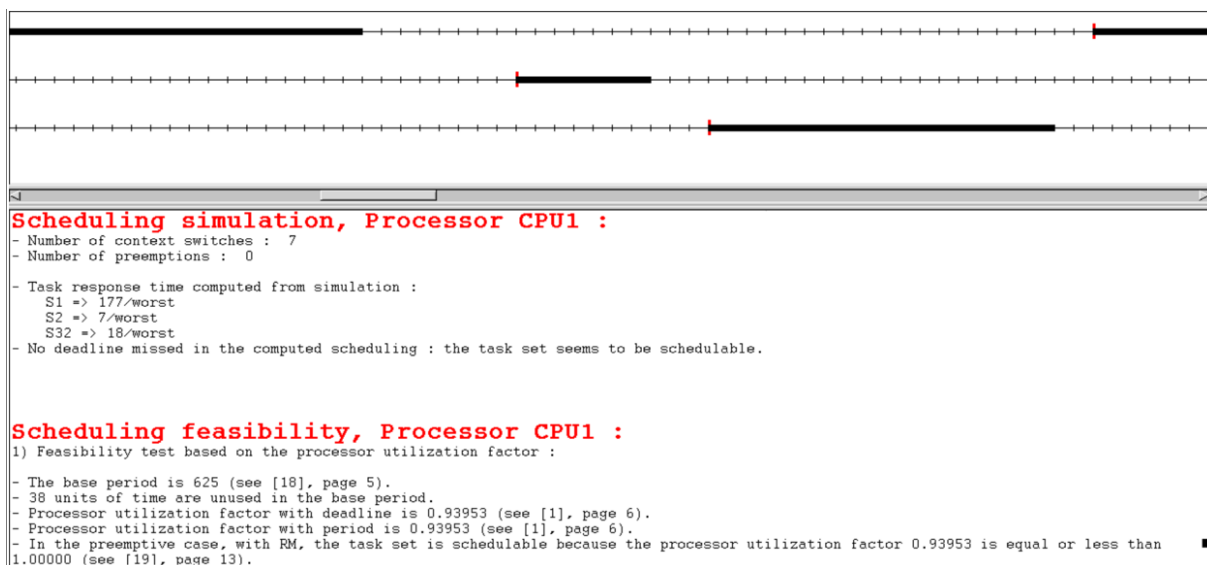


Fig. 5 Cheddar Simulation

Hand Drawn Timing Diagram

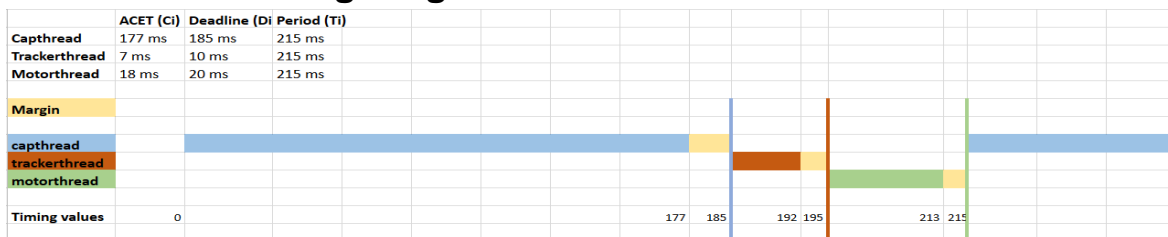


Fig. 6 Hand Drawn Timing Diagram

We used RM policy for testing the scheduling feasibility of services in our system. Per Cheddar analysis, our task set was found to be schedulable, with a processor utilisation factor of 93.953%. We also verified the plotted Cheddar with our hand drawn timing diagram and found the deadlines and the margin to be the same.

Scheduling Point / Completion tests

Scheduling point test is an iterative test based on the Lehoczky, Sha, Ding theorem, if a set of services can be shown to meet all deadlines from the critical instant up to the longest period of all tasks in the set, then the set is feasible.

Completion test is another necessary and sufficient (N & S) feasibility test where the cumulative demand from (n) tasks up to time (t) is less than or equal to the deadline for a service S_n , which proves that S_n is feasible. Proving this same property for all S from S_1 to S_n proves that the service set is feasible.

The code taken from Professor Sam Siewert's textbook is an algorithm for Scheduling Point and Completion time tests. The code runs with an assumption that, arrays are sorted according to the RM policy where period [0] is the highest priority and shortest period.

```
pi@raspberrypi: ~/Feasibility
pi@raspberrypi:~/Feasibility $ make
gcc -O0 -g -c feasibility_tests.c
gcc -O0 -g -o feasibility_tests feasibility_tests.o -lm
pi@raspberrypi:~/Feasibility $ sudo ./feasibility_tests
-----RTES FINAL PROJECT Completion Test Feasibility Example-----
U=0.94 (C1=177 ms,C2=7 ms,C3=18 ms; T1=215 ms, T2=215 ms, T3=215 ms; D1= 185 ms, D2= 10 ms, D3 = 20 ms): FEASIBLE

-----RTES FINAL PROJECT Scheduling Point Feasibility Example-----
U=0.94 (C1=177 ms, C2=7 ms, C3=18 ms; T1=215 ms, T2=215 ms, T3=215 ms; D1 = 185 ms, D2=10 ms,D3 = 20 ms): FEASIBLE
pi@raspberrypi:~/Feasibility $
```

Fig. 7 Scheduling point and completion time feasibility tests

From above figure we can see that our system's service set was found to be feasible by both scheduling point and completion time tests.

Safety margin analysis

We estimated following margins for our system-

Services	C _i + Margin (ms)	Deadline (D _i) (ms)	Period (T _i) (ms)
capthread	177 (+ 8 ms)	185	215
trackerthread	7 (+ 3 ms)	10	215
motorthread	18 (+ 2 ms)	20	215

We estimated margins for our system as denoted in table above, which is 13 msec over a total period of (177+7+18 = 202) msec which amounts to approximately 6 %. We verified this from the value of utilisation factor calculated in scheduling point and completion time tests. Rate Monotonic Least Upper Bound is a feasibility test by Liu and Layland which is defined as-

$$U = \sum_{i=1}^m (C_i/T_i) \leq m(2^{1/m} - 1)$$

U: Utility of the CPU resource achievable
C_i: Execution time of Service i
m: Total number of services in the system sharing common CPU resources
T_i: Release period of Service i

Ref. Real time embedded systems textbook by Prof. Sam Siewert

However, RM LUB is not a necessary feasibility test, but sufficient. The RM LUB is a pessimistic feasibility test that will fail some proposed service sets that work, but it will never pass a set that doesn't work. For our set of services, we calculated CPU Utilization (U_i) = 94 %, and RM LUB is 78 %. But our analysis using the necessary and sufficient scheduling point and completion time tests we found our system to be FEASIBLE.

Generally for hard real time systems, according to RM LUB requirements the required margins are approximately 25% for a safe system. Since our system is Soft Real time, which needs to give a predictable response before deadline (not deterministic), we have taken a margin of 6 %. Also, per our tests using Cheddar analysis as well as jitter statistics noted over a range of 100 frames, we were able to produce a predictable response with minimal frames missing the deadline thus meeting the soft real time criterion.

Proof-of-Concept with Example Output and Tests Completed

Resources used

- Raspberry Pi Model B for development board with following features:
- Quad Core 64 bit ARM Cortex A5
- 5 MP Raspberry Pi Camera module with Camera Serial Interface
- Broadcom GPIO pinout for interfacing with servo motors
- OpenCV C++ features of BGR to HSV, Threshold, Contour detection, Kalman Filter
- Usage of OpenMP libraries for parallel execution of certain functions
- Two 9g Micro servo motors (0 to 180 deg) for movements across X and Y axes
- C++ API for interfacing with RPi Cam, GPIO access C library for RPi
- Camera mount to facilitate Pan and Tilt movements

Usage of Openmp for achieving parallelism

We have used OpenMP, which is a multiprocessing library in C for performing certain functions in the code in parallel manner. Our development board had a quad core CPU, so for parallelism, we used OpenMP library to execute certain functions on these four cores in a parallel manner. The code excerpt is as follows –

```
//function to draw the trajectory lines. openMP used for parallel execution of for loop
#ifdef TRAJECTORY
    if(myframecount > 2)
    {
#pragma omp parallel shared(center_array_x,center_array_y,result)
    {
        int i=0;
#pragma omp for private(i)
        for(i = 1; i<myframecount; i++)
        {
            line(result, Point(center_array_x[i],center_array_y[i]), Point(center_array_x[i-1],center_array_y[i-1]), Scalar(0,255,0),5);
        }
    }
}
#endif
```

Calculation for center co-ordinates was performed using OpenMP where the for loop is calculated on each of the four cores in a parallel manner for private variables not shared by any other thread.

Time-stamp tracing

For every thread, immediately after semaphore is received, **start timestamp** is taken and before releasing the semaphore for next thread, **stop timestamp** is taken. Difference between the two timestamps is taken using **Professor Sam Siewert's delta function**. The difference calculated is the **execution time**. Once, the execution time is calculated, the **jitter** for that thread is calculated with respect to the deadline. **Jitter is calculated by taking the difference of execution time and the deadline. Positive jitter** means execution time **exceeds**

deadline. Negative jitter is when the execution time is completed **before the given deadline**. And after capturing 100 frames, the total jitter i.e. sum of positive and negative jitter is calculated. We saw that, our jitter varied depending on the camera view, light conditions and the nearby environment.

```

pi@raspberrypi:~/RTES_FINAL_PROJECT/source $ make
g++ opencv-kalman.cpp -o asl -lopencv_calib3d -lopencv_core -lopencv_features2d -lopencv_flann -lopencv_highgui -lopencv_imgproc -lopencv_ml -lopencv_obj
o -lopencv_stitching -lopencv_superres -lopencv_video -lopencv_videoio -lopencv_videostab -lraspicam -lraspicam_cv -lrt -lpthread -l wiringPi -fopenmp
pi@raspberrypi:~/RTES_FINAL_PROJECT/source $ sudo ./asl
Cap thread created
Tracker thread created
Motor thread created

Hit Ctrl-C to exit...

-----MOTOR THREAD STATISTICS-----
Frames Missed Deadline: [2]
Total positive jitter: 352.000000
Total Negative Jitter: -454.000000
Total Jitter: -102.000000

-----CAPTURE THREAD STATISTICS-----
Frames Missed Deadline: [5]
Total positive jitter: 281.000000
Total Negative Jitter: -1049.000000
Total Jitter: -768.000000

-----TRACKER THREAD STATISTICS-----
Frames Missed Deadline: [0]
Total positive jitter: 0.000000
Total Negative Jitter: -120.000000
Total Jitter: -120.000000

-----FINAL ANALYSIS-----
Exec Per Frame: [199.685013 msecs]
Frames Missed Deadline: [2]
Total Loop positive jitter: 614.000000
Total Loop Negative Jitter: -1910.000000
Total Loop Jitter: -1296.000000

(Capthread) ACET: [177.396042 msecs]
(Capthread) WCET: [452 msecs]
(Trackerthread) ACET: [6.445545 msecs]
(Trackerthread) WCET: [8 msecs]
(Motorthread) ACET: [16.980000 msecs]
(Motorthread) WCET: [361 msecs]

-----MAIN END-----

```

Fig. 8 Jitter Analysis

Services	Positive Jitter (milliseconds)	Negative Jitter (milliseconds)	Total Jitter (milliseconds)
capthread	281	-1049	-768
trackerthread	0	-120	-120
motorthread	352	-454	-102
All services	614	-1910	-1296

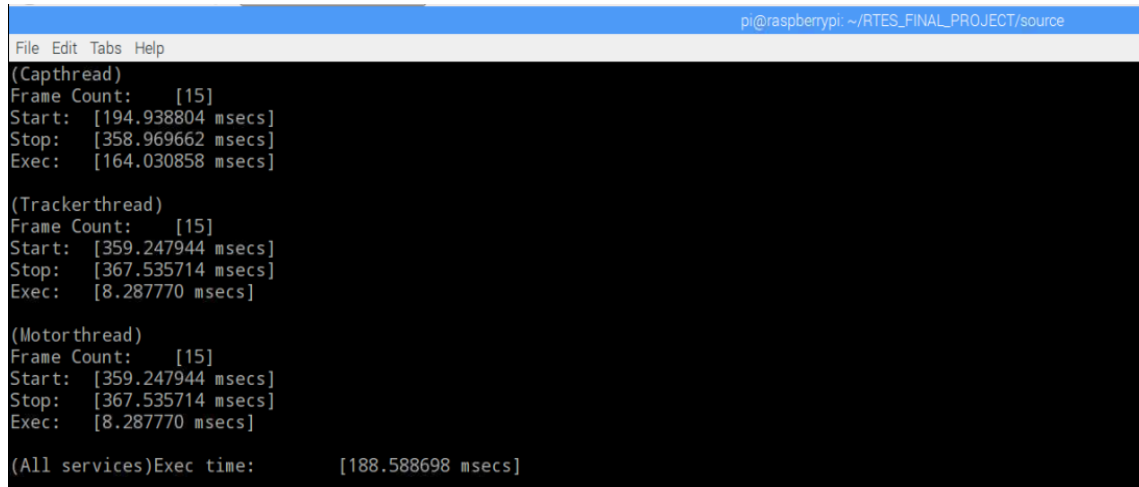
From above values we can see that, negative jitter is much higher as compared to positive jitter for each of the individual services and overall entire frame as well. Also from our output of jitter analysis we could see that overall for entire system only **2 frames out of 100** missed their deadline. Per this verification, we found our system to be acceptable for a soft real time application.

Test cases

To test our system we chose following test cases -

Test Case 1

For verification that a sample group of frames followed our WCET and jitter analysis, we printed timestamps to capture service request time, service response time and service execution time.



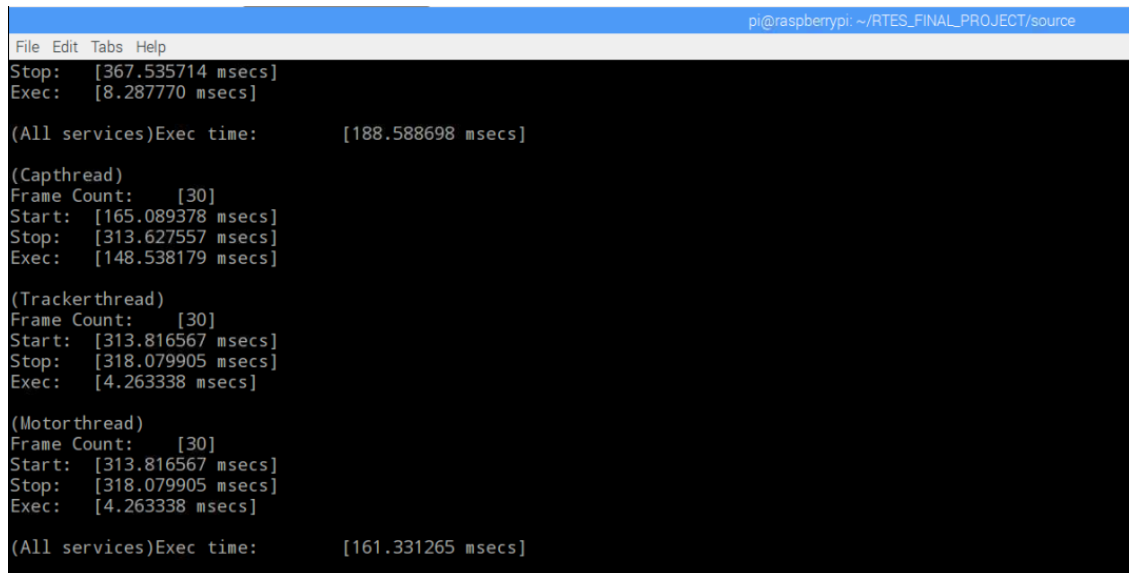
```
pi@raspberrypi: ~/RTES_FINAL_PROJECT/source
File Edit Tabs Help
(Capthread)
Frame Count: [15]
Start: [194.938804 msecs]
Stop: [358.969662 msecs]
Exec: [164.030858 msecs]

(Trackerthread)
Frame Count: [15]
Start: [359.247944 msecs]
Stop: [367.535714 msecs]
Exec: [8.287770 msecs]

(Motor thread)
Frame Count: [15]
Start: [359.247944 msecs]
Stop: [367.535714 msecs]
Exec: [8.287770 msecs]

(All services)Exec time: [188.588698 msecs]
```

Fig. 9 Time stamp tracing for frame #15



```
pi@raspberrypi: ~/RTES_FINAL_PROJECT/source
File Edit Tabs Help
Stop: [367.535714 msecs]
Exec: [8.287770 msecs]

(All services)Exec time: [188.588698 msecs]

(Capthread)
Frame Count: [30]
Start: [165.089378 msecs]
Stop: [313.627557 msecs]
Exec: [148.538179 msecs]

(Trackerthread)
Frame Count: [30]
Start: [313.816567 msecs]
Stop: [318.079905 msecs]
Exec: [4.263338 msecs]

(Motor thread)
Frame Count: [30]
Start: [313.816567 msecs]
Stop: [318.079905 msecs]
Exec: [4.263338 msecs]

(All services)Exec time: [161.331265 msecs]
```

Fig. 10 Time stamp tracing for frame #30

```

pi@raspberrypi: ~/RTES_FINAL_PROJECT/source
File Edit Tabs Help
(All services)Exec time:      [200.377118 msecs]

(Capthread)
Frame Count:      [90]
Start: [461.359528 msecs]
Stop:  [639.254535 msecs]
Exec:  [177.895007 msecs]

(Trackerthread)
Frame Count:      [90]
Start: [639.491097 msecs]
Stop:  [647.485063 msecs]
Exec:  [7.993966 msecs]

(Motorthread)
Frame Count:      [90]
Start: [639.491097 msecs]
Stop:  [647.485063 msecs]
Exec:  [7.993966 msecs]

(All services)Exec time:      [201.578674 msecs]

```

Fig. 11 Time stamp tracing for frame #90

Test Case 2

These test cases (2,3,4,5) were to test the accuracy of object detection and tracking in test environments with different values of luminosity and for different hue values. We chose an environment with poor lighting conditions and another environment with good lighting conditions. The test performed also included two different color hues (red and blue) for the object to be tracked.

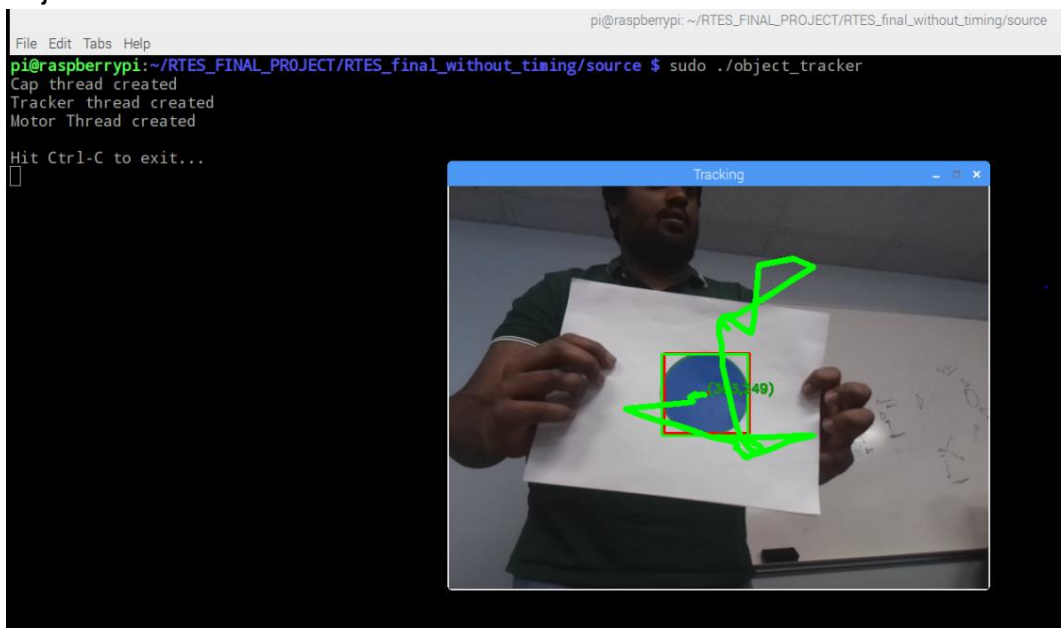


Fig. 12 Low light conditions, detected color is blue, detected object is circular in shape

Test Case 3

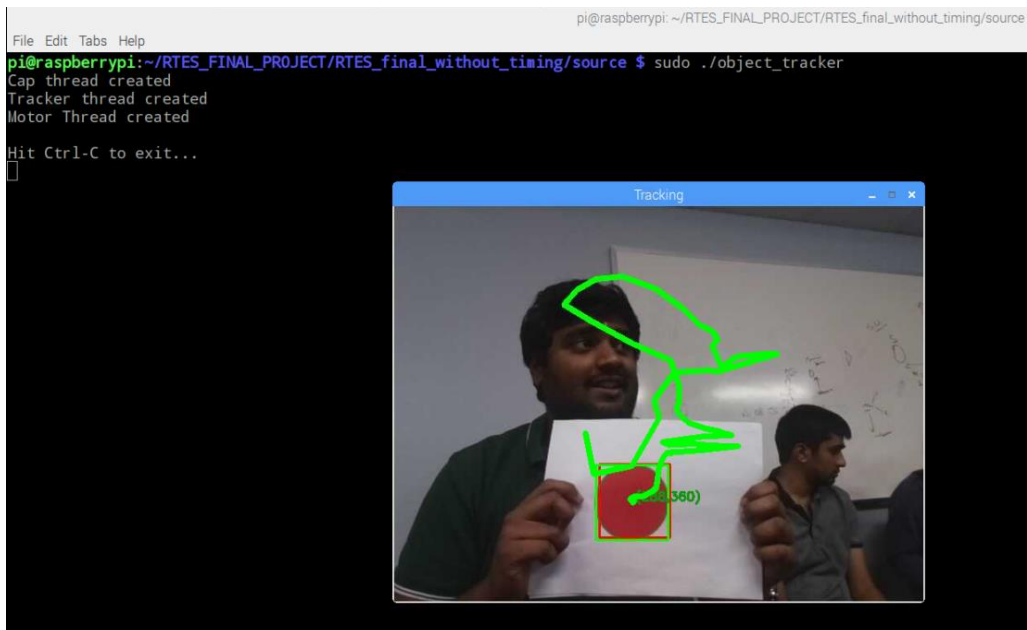


Fig. 13 Low light conditions, detected color is red, detected object is circular in shape

Test Case 4

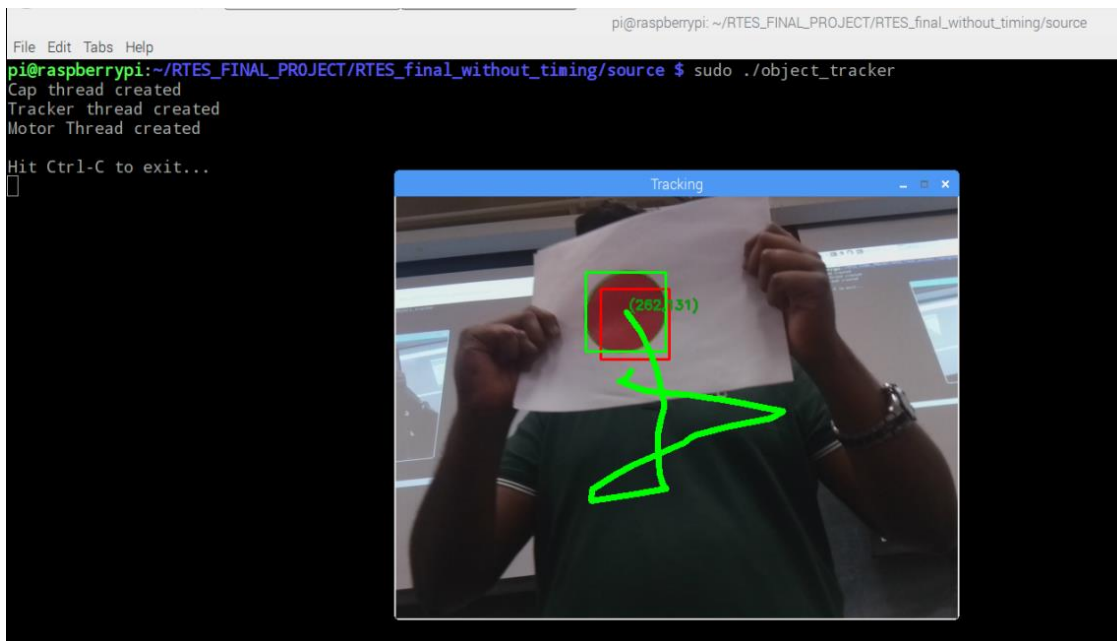


Fig. 14 Classroom light conditions, detected color is red, detected object is circular in shape

Test Case 5

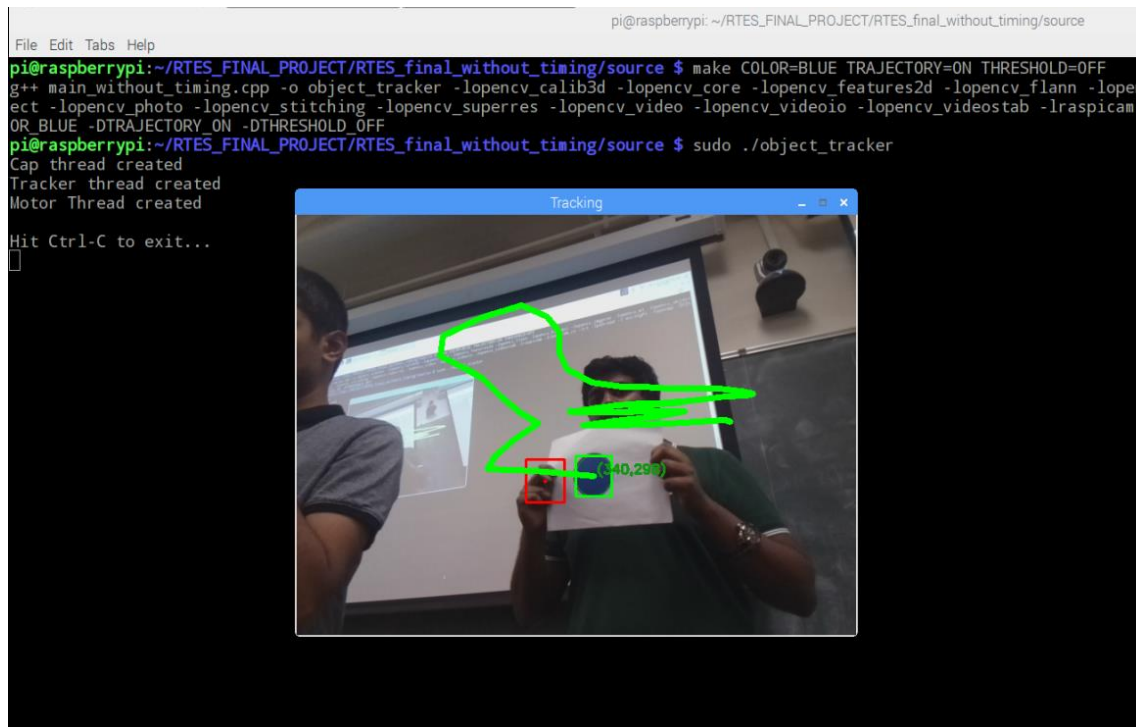


Fig. 14 Classroom light conditions, detected color is blue, detected object is circular in shape

Conclusion

The prototype of Object Detection and Tracking System was implemented through the knowledge of Real-time and OpenCV concepts learnt along the duration of coursework. We faced some difficulties during the prototype implementation of our system. The OpenCV installation took us a lot of time than we had expected. Also, we are detecting objects based on the values of hue and saturation. But these values were found to change with different light conditions. Therefore, instead of a single value we were taking a range of values for hue and saturation for better detection of object.

Also, through timestamp tracing we observed that the drawing of the trajectory which represents the path traversed by the object took a lot of time due to which more frames were missing deadlines and generated less fps. Using OpenMP with 4 threads on the quad-core processor for its parallel execution helped to reduce its time considerably.

Overall we were able to receive frame rate of 5 which is acceptable for surveillance applications. Also, our system can detect objects in the speed range of 1.3 to 1.6 metres/second. The limitation of this speed is due to Raspberry Pi Processor and RaspiCam. Using a better processor and a camera with better resolution and fps, we could have detected objects with greater speeds.

Formal References

1. Usage of HSV saturation to detect colors and contours in captured images - <http://www.cs.utah.edu/~ssingla/CV/Project/Index.html>
2. A multiple object tracking method using Kalman filter - <https://ieeexplore.ieee.org/document/5512258/>
3. Video object tracking using adaptive Kalman filter - <https://www.sciencedirect.com/science/article/pii/S1047320306000113>
4. Erode and Dilate - https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html
5. Real time analysis, RM LUB analysis - Real time embedded systems by Professor Sam Siewert
6. Kalman Filter Algorithm - <http://web.mit.edu/kirtley/kirtley/binlustuff/literature/control/Kalman%20filter.pdf>
7. Object tracking algorithms in OpenCV C++ - <https://www.myzhar.com/blog>

Acknowledgements

We would like to thank Professor Timothy Scherr for his valuable guidance throughout the duration of the project. We would like to thank all three TAs for all their valuable inputs helped us out of the tricky parts of this project.

We would also like to thank the ITL staff who helped us use their workshop for constructing the camera Pan and Tilt mount arrangement with its supporting base.

Appendices

1. The code associated with our system is included in the folder named - *ECEN5623_RTES_Final_Project_Code_With_Timing_Analysis*
2. Feasibility test code references from Professor Sam Siewert's code included in folder - *ECEN5623_RTES_Final_Project_Feasibility_Code*
3. Final presentation file attached in file named - *ECEN5623_RTES_Final_Presentation_Vignesh_Vihanga_Vishal*
4. Project demo video attached - *ECEN5623_RTES_Final_Project_Demo_Vignesh_Vihanga_Vishal*