Embedded Implementation of Visual Detection and Tracking Algorithm for Mobile Robot

Gunawan Lumban Gaol
Bandung Institute of Technology
School of Electrical Engineering and Informatics
Dept. of Electrical Engineering
40132, Bandung, Indonesia
gunawan.marbun@students.itb.ac.id

Kusprasapta Mutijarsa
Bandung Institute of Technology
School of Electrical Engineering and Informatics
Dept. of Electrical Engineering
40132, Bandung, Indonesia
sony@stei.itb.ac.id

Widyawardana Adiprawita
Bandung Institute of Technology
School of Electrical Engineering and Informatics
Dept. of Electrical Engineering
40132, Bandung, Indonesia
wadiprawita@gmail.com

***Abstract*— This article describes findings on implementation of person detection and tracking algorithms in Nvidia Jetson TK1 using OpenCV4Tegra and python language. It uses built-in OpenCV HOG (Histogram of Oriented Gradients) descriptor for detection and KCF (Kernelized Correlation Filters) for tracking. Parameter values are then tuned to the best trade off values between speed and accuracy. Both detection and tracking algorithms are evaluated on data samples taken while the robot is in function, providing realistic measurement on such real cases. Hardware implementation is done as stated in [1].**

***Keywords*— person detection and tracking, Jetson TK1, Opencv4Tegra.**

I. Introduction

This article addresses the embedded implementation problem of RGB based people detection and tracking from the perspective of a moving observer. We focus on indoor scenarios in which a robot is patrolling an area and taking tracking action on supposedly suspicious people. Patrolling used point to point movement in 2D world coordinate. During patrolling, robots can take observation between each point to determine suspicious people. Suspicious is defined as being in a relatively same location for an amount of duration, in which the robot will take tracking action until it is stopped or lose track of the person. Losing track is defined after a certain amount of image frame processed without detecting any person.

The first problem of embedded implementation is computational cost. Real time detection and tracking must be fast and accurate enough despite having several limitations on hardware. A widely accepted practice in software development is to use hardware-friendly languages such as C++ for writing software implementation code. However, developing in C++ demands substantial effort, which can lead to longer development times. Conversely, writing in high-level languages such as Python can slow down processes. To strike a balance between efficiency and ease of development, we adopted an approach that utilizes Cython to interface between C++ and Python.

The second problem addresses modularity. We have tried using ZED Camera but came up with Kinect since the ZED SDK supported on Jetson TK1 is only up to 1.2.0, disabling us from using python as our software implementation language. Although we use Kinect v.1 to acquire images [1], we use only RGB data to enable another commercial camera to be used as a substitute for Kinect as long as we use similar hardware as Nvidia Jetson TK1. Another advantage is that using RGB data acquired from the camera provides the most safe way as it doesn't require emitting a signal from the device.

Nvidia Jetson TK1 comes with Linux kernel version 3.10.40, CUDA 6.5, and OpenCV 2.4.8. Nvidia enables hardware acceleration of OpenCV in two ways, between 2x-5x speedup in Nvidia's Tegra CPU and between 5x – 20x speedup in GPU modules (including HOG module). List of accelerated functions can be found in [2].

II. Related Works

Hardware devices have their own limitations. For Kinect, suggested usage is 1-3 m distance to the sensor based on [3] that agrees on Microsoft documentation on Kinect specifications in [4]. Summary of specifications are shown in Table I.

TABLE I
Kinect Specifications

Specs	Values	Unit
Frame rate (depth and color stream	30	frames per second (FPS)
Viewing Angle	43 vertical, 57 horizontal	degree
Suggested Usage	1-3	meter
Color Resolution	640x480	pixel
Depth Resolution	640x480	pixel

We use OpenKinect libfreenect library [5] with python wrapper to use Kinect in our implementation. We discovered that we need to manually change the depth bit format in function sync_get_depth inside file freenect.pyx from

DEPTH_11BIT to DEPTH_REGISTERED in order to make depth values pixel mapped correctly to the colour values pixel. Directly modifying the format argument in main source code happened to invoke wrong results of depth values.

Depth values in cm are calculated from raw 11-bit disparity value by using formula given in [6]. The approximation is stated to be 10 cm off at 4 m away and less than 2 cm off within 2.5 m. Angle measurement values relative to the robot are calculated using a simple formula of dividing horizontal field of view to width of the image. Both formulas can be seen in the equation below.

$$Distance\,(cm) = 100/(-0.00307 * rawDisparity + 3.33)$$

$$Angle_{degree/pixel} = \frac{HFOV}{Width}$$

Some related works of object detection algorithm using Nvidia Jetson TK1 involving neural network claims to have around 5-6 frames per second using Darknet Yolo and 5 frames per second using Caffe SSD [7], too slow for application in real-time. Another reason for us to use a simpler approach using optimized HOG descriptors on Jetson TK1.

III. Object Detection Methods

We use built-in OpenCV methods of a pre-trained HOG + Linear SVM model to perform pedestrian detection. This method detects people (pedestrian) on structural level of having a head, two arms, a torso, and two legs. The first step in training an object detector using HOG is to sample *P* positive samples and *N* negative samples and extract HOG descriptor from these samples. Supposedly *N* has to be greater than *P*. Next step is to train a Linear Support Vector Machine on both of positive and negative samples. It basically split hog descriptor values of positive and negative sample by a linear line with a margin as illustrated by Fig. 1.

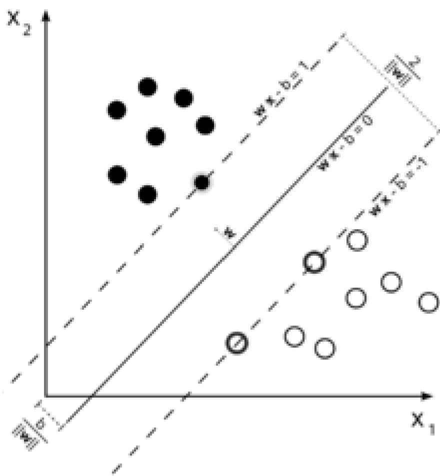

Fig. 1 Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called support vectors.

The next step is to retrain SVM further by applying hard-negative mining. This is done by computing HOG descriptors for each image and each possible scale of each image in the negative training set. If it incorrectly classifies a given window as an object, record the feature vector associated with false-positives patch along with the probability of classification.

After training the classifier, it is then used on a sliding window on each frame of each scale acquired from the camera to detect pedestrians in, returning the bounding box. We then used this bounding box information to count the assumed centre pixel of the detected people. We use simple calculations of dividing the width and height by two and adding it to the origin pixel of the bounding box located at the top left-most pixel.

For each frame processed, we then compare the centre pixel values found to the previous one. If the difference is lower than 20 pixels we assume the same location hence increasing the time measurement. If it doesn't detect people, same as no centre pixel values found, the timer is reset to 0. State change to tracking if the time measurement value exceeds timeout value.

Having mapped raw disparity values to each colour pixel, we then directly access raw disparity array values and apply distance(cm) calculation formula to get distance value relative to camera. This data of distance and angle measurement relative to the camera is then sent to the microcontroller to be used for navigation. Summary of algorithm is shown in Algorithm 1. Note that the bounding box drawing function is not shown.

Algorithm 1 Object Detection

```
 1: Initialize HOG descriptor and Linear SVM Classifier.
 2: Initialize prevpixel value.
 3: Acquire image and detect.
 4: if detected then
 5:     update centrepixel
 6:     if diffabs(centrepixel,prevpixel) < 20 then
 7:         count++.
 8:     else
 9:         count = 0.
10: else
11:     count = 0
12: if count>timeout then
13:     state = track.
14:     get distance and angle value.
15:     send data to microcontroller
16: go to step 3.
```

Object detection algorithm is tested based on this scenario. First scenario is to place a robot along with a camera on a location standing still. A person is then walking at a usual walking pace facing away from the camera near camera up to ~5 metres away and then walking back toward the camera facing to camera at usual walking pace. Second scenario is to make a person facing an angle rotated a whole circle from several distance locations from the camera. In both scenarios, we measure true positive rate and fps on various parameters.

We also measured the effect of different image sizes and colours.

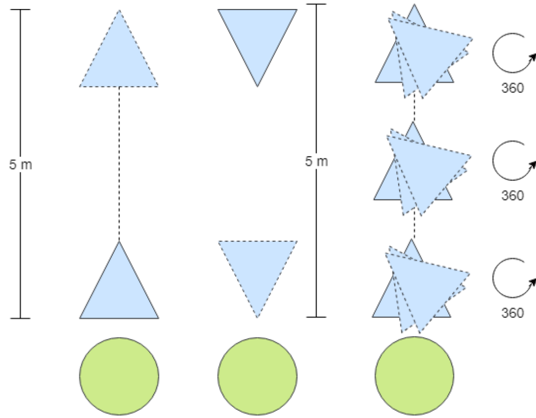

Fig. 2 Illustration of object detection testing scenario. Green circle is a robot. Blue Triangle is person

IV. Object Tracking Methods

For tracking, we use Kernelized Correlation Filter with the initialized bounding box as the last frame of detection before the timeout. The method is implemented from [8] with wrappers in python from [9]. KCF is used to have high speed and moderate accuracy. We noticed some extended works in C++ in [10] and [11] but haven't yet created the python wrappers. Some works suggest the use of OpenCV contrib module, but due to shortage of device storage, we did not manage to use that module.

Our chosen method is then from [9] with slight modification on the KCF.pyx file to include peak value as to determine tracking failure by simple thresholding. This thresholding value suggests failure when an object is moving too fast or is in occlusion with another object. When this happens, the robot will return to detect the state while patrolling.

First, the tracker bounding box is initialized from the last bounding box frame with modification. We discovered that a person's head has more distinctive features compared to the whole body, so we modify the bounding box to frame only the head by assuming simple approximations to manage computational cost. This modified bounding box is then used as initialization of the tracker. It then updates the tracker continuously if the peak value is still lower than the threshold value ranging from 0.1 up to 1. Summary of algorithm is shown in Algorithm 2. Note that the interface part of showing the bounding box is not shown here.

Algorithm 2 Object Tracking

1: Acquire last bounding box
2: Draw new bounding box based on last bounding box
3: Initialize tracker based on new bounding box
4: getpeakvalue()
5: **if** peakvalue > threshold **then**
6: acquire a new image.
7: update bounding box.
8: get distance and angle value.
9: send data to microcontroller
8: go to step 4.
9: **else**
10: state = detect

The object tracking algorithm scenario is tested on both scenarios used in testing object detection algorithms with true positive rates values exchanged with peak value. Another scenario is also added when the robot truly performs its task in following a real person in real scenario, which in this case is an indoor environment one.

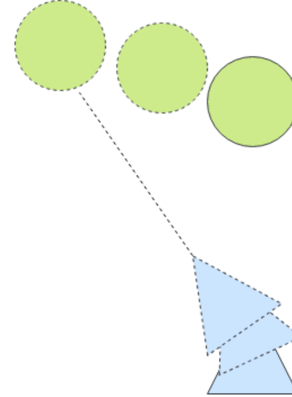

Fig. 3 Illustration of object tracking testing scenario. Green Circle is Robot. Blue Triangle is a person.

V. Video Streaming Methods

As described in [1], our system uses Flask micro framework to stream video for browsers via Motion JPEG. This method has low latency although not the best in quality since JPEG compression is not very efficient for motion video.

The generator function takes a Camera class that returns processed frames with bounding box and information drawn in jpeg format. File writing done using function cv2.imwrite() which benefits from CPU acceleration provided by Nvidia. We measure the effect of setting up streaming video on the fps of the process. We use a mobile hotspot from our phone to connect the Nvidia Jetson TK1 to the internet to stream to a local IP address. This to minimize the effect of loss packet due to heavy latency.

VI. Experimental Results

The robot is small, with a height of approximately 48 cm and diameter of 12 cm. We place the Kinect as high as we can and get around 33 cm in height. We tilt Kinect around 5^0 upward to maximize vertical field of view in order to increase accuracy.

A. Range and Distance Measurement Results

We test the maximum and minimum range of detection of correct distance reading value using HOG descriptor. We get a range from around 2.25 metre to 4.5 metre. However, the

correct distance value can only be measured up to a maximum of around 3 metres, the same result as in [3] and [4].

We provide distance measurement results at 2.4 m to 4.5 m with 30 cm interval measurement. The average error is around 2.1 cm. Some frames read are shown in Fig. 4.

TABLE II

DISTANCE ACCURACY MEASUREMENT

Distance(cm)	Measured(cm)	Error(cm)
240	241.84	1.84
270	269.89	0.11
300	294.27	5.73
330	330.05	0.05
360	363.17	3.17
390	389.21	0.79
420	419.27	0.73
450	454.36	4.36
	Average	2.0975

Fig. 4 Measurement frame at 390 cm (top) and 360 cm (bottom). Images are cropped.

B. *Object Detection Measurement Results*

Default parameters of HOG descriptor win stride size of (8,8), padding size of (16,16), scale of 1.06. Results shown in several tables below. On Table III, parameters are set to default. Table IV shows the effect of changing window size. Table V shows the effect of changing the scaling value, evaluated on gray400 mode. Table VI shows the effect of changing scaling value on win 16, evaluated on gray400 mode.

TABLE III

EFFECT OF SIZE AND COLOUR

Mode	Pos	Neg	FPS
rgb640	0.651163	0.348837	3.251988
rgb400	0.762712	0.237288	7.941857
gray640	0.606061	0.393939	3.769687
gray400	0.75	0.25	9.085793

TABLE IV

EFFECT OF WINSTRIDE VALUE

Mode	Pos	Neg	FPS
win4-gray400	0.548387	0.451613	3.64511
win8-gray400	0.75	0.25	9.085793
win16-gray400	0.818182	0.181818	9.411325

TABLE V

EFFECT OF SCALE VALUE SMALL WINSTRIDE

Mode	Pos	Neg	FPS
win4-scale1.06	0.548387	0.451613	3.64511
win4-scale1.12	0.65	0.35	5.453001
win4-scale1.18	0.612245	0.387755	6.922598
win4-scale1.30	0.727273	0.272727	6.293497

TABLE VI

EFFECT OF SCALE VALUE BIG WINSTRIDE

Mode	Pos	Neg	FPS	Rate
win16-scale1.01	0.666667	0.333333	3.63821	0.209302
win16-scale1.06	0.818182	0.181818	9.411325	0.05314
win16-scale1.18	0	1	23.7641	0

The positive count increases if the frame return detected box with the correct corresponding distance value (not showing negative value), it counted negative otherwise. Mode win16-gray400 looks promising, but if we look into how many frames it lost totalling the not detected bounding box, it only detects 5% of total frames, which is 1/6 times smaller than the average results of 30%, therefore unreliable.

The best tuning parameters therefore are presented with the default parameter values, at 7.94 fps of 76% positive rate detection.

Using the default parameter values, we then examined the facing angle in which the algorithm fails to detect a person. The data on 2nd position further from camera are not presented because of false distance reading values while data on 3rd position further from camera for not showing any detection. Some figures from the 1st position further from the camera where facing angle causes failure in detection are shown below.

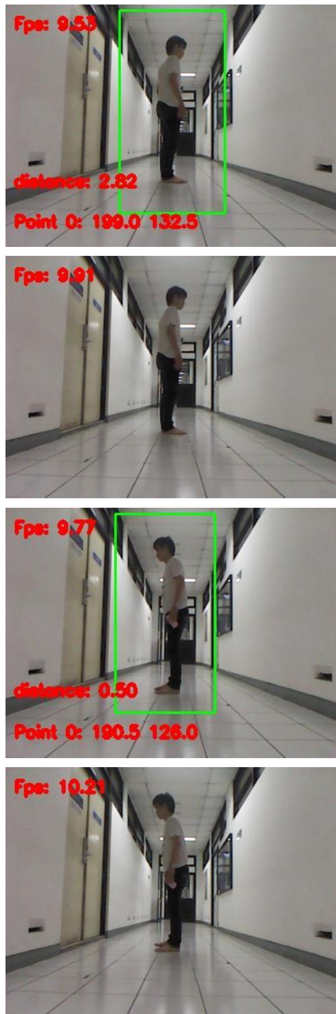

Fig. 5 Positive and negative detection. The difference between the image on the left and on the right is 1 frame. Images on the left precede images on the right. Person rotated clockwise.

From Fig. 5 we can infer that the detection algorithm fails to detect a person at an angle in which the facing angle is perpendicular to the camera angle. Possible cause is missing one leg and one hand features from the pretrain SVM detector that detect two legs and two hands.

C. Object Tracking Measurement Results

Positive rates are determined as correct distance measurement. Measurement values are shown in Table VII.

TABLE VII

MEASUREMENT VALUES OF OBJECT TRACKING USING KCF

Mode	Pos	Neg	FPS
rgb400(default)	0.980545	0.019455	14.58001

We observe a fluctuating fps value from 5.7 fps to 20.45 fps. The cause is shown to be a serial communication problem between the microcontroller and camera in the USB hub as we write files simultaneously both from the microcontroller and from the Nvidia Jetson TK1. Nevertheless, we achieve a high positive rate of 98% at around 14.58 fps average, making this algorithm suitable enough for real time tracking.

D. Video Streaming Measurement Results

Streaming results give around 1 to 2.5 fps drop from offline use. Intermittent delay sometimes happens due to mobile hotspot connectivity problems. Fig. 6 show some frame sample of online streaming.

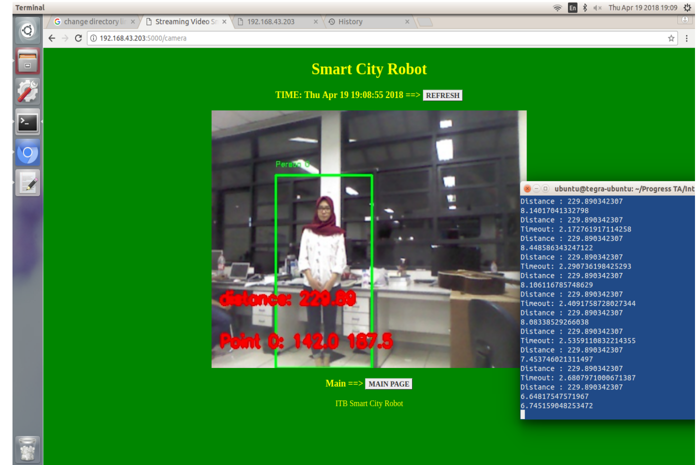

Fig. 6 Motion JPEG streaming implementation on detection algorithm using mobile hotspot. FPS values range between 6 to 8 fps.

VII. CONCLUSIONS

We present findings in Nvidia Jetson TK1 OpenCV4Tegra implementation of person detection using HOG descriptor + Linear SVM and person tracking using Kernelized Correlation Filter using python language. The best tuning parameter value for HOG detect multiscale is win stride value of (8,8), scale value of 1.06, on 400 pixels width RGB frames. Fps for this tuning parameter is 7.94 with 76% positive rate of detection.

Future works would be to implement the ensemble tracker algorithm to address problems with occlusion. The use of more advanced Nvidia Jetson version e.g. TX1 is recommended when using neural network framework approach.

REFERENCES

[1] G. Gunawan Lumban, S. Rahmadiana, K. Ricky, "Development Platform Robot Smart City",
[2] (2018) The Elinux website. [Online]. Available: http://elinux.org/
[3] Khoshelham K., Elberink S.O. Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications. Sensors. 2012;12:1437-1454. doi: 10.3390/s120201437.
[4] (2018) The Microsoft Developer Network (MSDN) website. [Online]. Available: https://msdn.microsoft.com/en-us/library/jj131033.aspx
[5] (2018) OpenKinect github. [Online]. Available: https://github.com/OpenKinect/libfreenect/
[6] (2018) OpenKinect Documentation Website. [Online]. Available: https://openkinect.org/wiki/Imaging_Information#Depth_camera_accuracc
[7] (2018) Nvidia Devtalk Website. [Online]. Available: https://devtalk.nvidia.com/
[8] J. F. Henriques, R. Caseiro, P. Martins, J. Batista, "High-Speed Tracking with Kernelized Correlation Filters", TPAMI 2015.
[9] (2018) KCFpy github. [Online]. Available: https://github.com/uoip/KCFcpp-py-wrapper

[10] Senna, Pedro and Dummond, Isablea Neves and Bastos, Guilherme Sousa. “Real-time ensemble-based tracker with Kalman filter” in SIBGRAPI’17.

[11] Huynh, Phung & Choi, In-Ho & Kim, Yong-Guk. (2015). “Tracking a Human Fast and Reliably Against Occlusion and Human-Crossing”. 10.1007/978-3-319-29451-3_37.

[12] (2018) Flask Documentation Website. [Online]. Available: http://flask.pocoo.org/docs/0.12/

[13] Liu, Wei and Anguelov, Dragomir and Erhan, Dumitru and Szegedy, Christian and Reed, Scott and Fu, Cheng-Yang and Berg, Alexander C. “SSD: Single Shot Multibox Detector”. ECCV. 2016