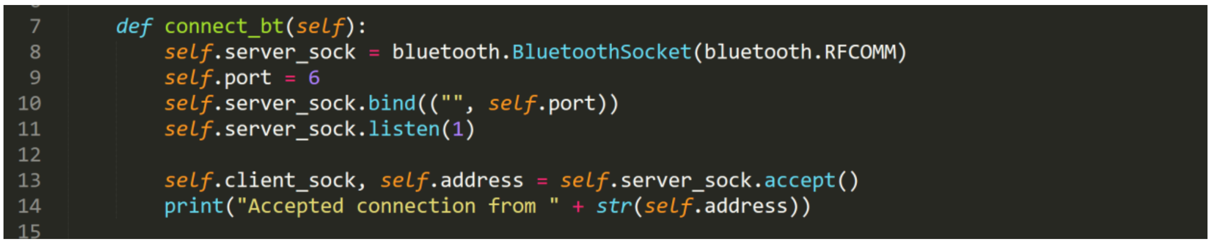
Bluetooth Communication

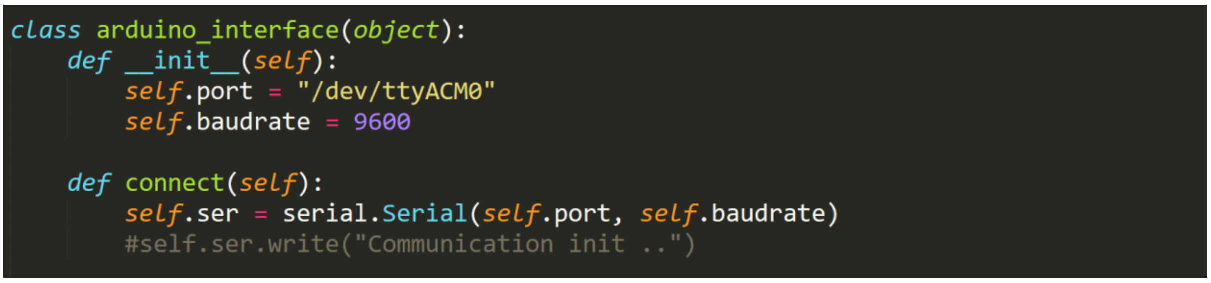
Communication between the Raspberry Pi and Android Tablet is achieved through Bluetooth using the RFCOMM protocol. Since Port 6 is the only serial port used on Raspberry Pi, we run the Raspberry Pi as the RFCOMM server and the Android Tablet as the RFCOMM client. In this way, the Bluetooth communication can always be established through Port 6 on Raspberry Pi.



```python
    def connect_bt(self):
        self.server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
        self.port = 6
        self.server_sock.bind(("", self.port))
        self.server_sock.listen(1)

        self.client_sock, self.address = self.server_sock.accept()
        print("Accepted connection from " + str(self.address))
```

Serial Communication

Communication between the Raspberry Pi and the Arduino board is achieved through serial communication. The Python library PySerial is used. When establishing connection, an ACM serial port will be open. However, the port number might change for each run. Therefore, the ACM port number in the /dev/ folder needs to be checked before each run.

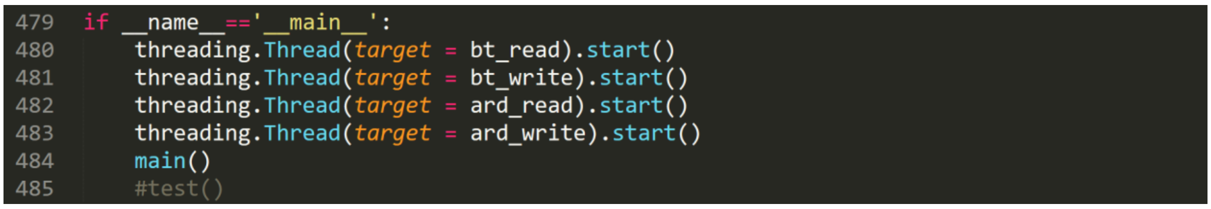


```python
class arduino_interface(object):
    def __init__(self):
        self.port = "/dev/ttyACM0"
        self.baudrate = 9600

    def connect(self):
        self.ser = serial.Serial(self.port, self.baudrate)
        #self.ser.write("Communication init ..")
```

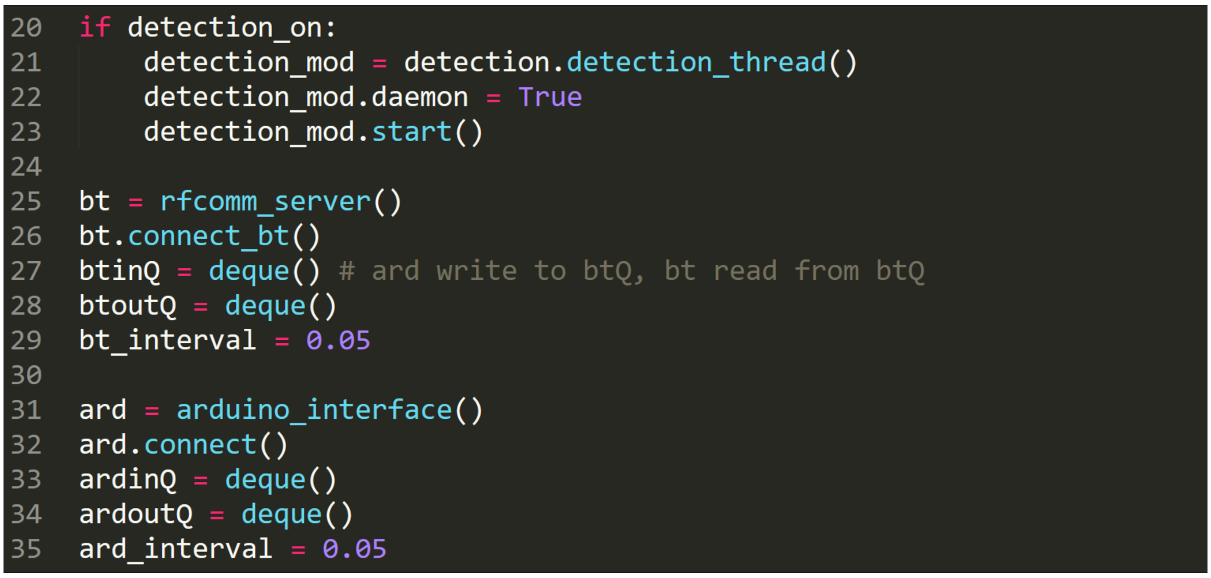
Multi-Threading

Since Raspberry Pi has a multi-core processor, running the system on multiple threads saves the overall processing time. The asynchronous nature of threading also requires extra efforts on synchronization of the processes.

On the main system we run four threads: Serial Communication Thread, Bluetooth Communication Thread, Image Detection Thread and the Main Thread. The image detection thread is run as a Daemon thread so that it constantly processes images in the background. When image detection result is needed, the main thread directly calls the detection thread for the result, saving time on Pi Camera initialization.



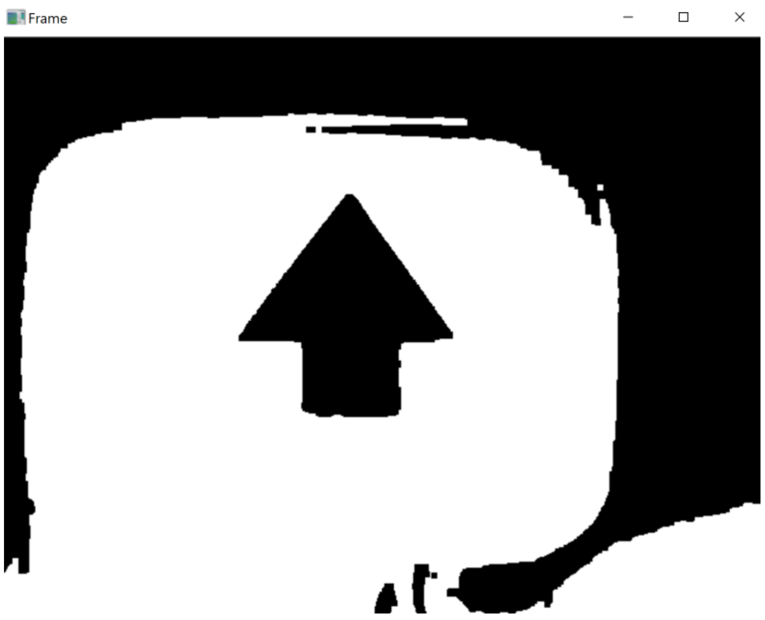
```python
if __name__=='__main__':
    threading.Thread(target = bt_read).start()
    threading.Thread(target = bt_write).start()
    threading.Thread(target = ard_read).start()
    threading.Thread(target = ard_write).start()
    main()
    #test()
```



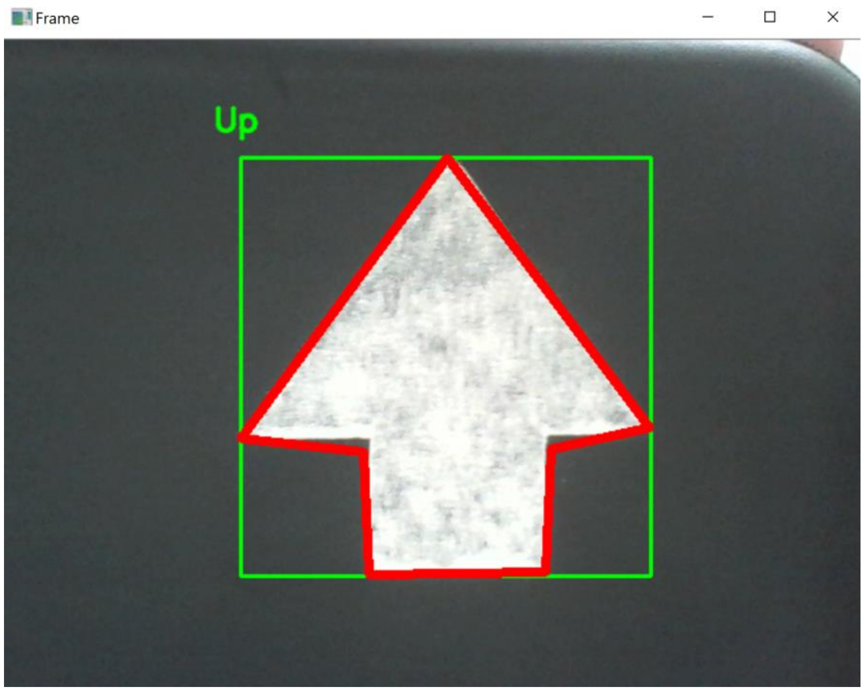
```python
if detection_on:
    detection_mod = detection.detection_thread()
    detection_mod.daemon = True
    detection_mod.start()

bt = rfcomm_server()
bt.connect_bt()
btinQ = deque() # ard write to btQ, bt read from btQ
btoutQ = deque()
bt_interval = 0.05

ard = arduino_interface()
ard.connect()
ardinQ = deque()
ardoutQ = deque()
ard_interval = 0.05
```

Arrow Detection

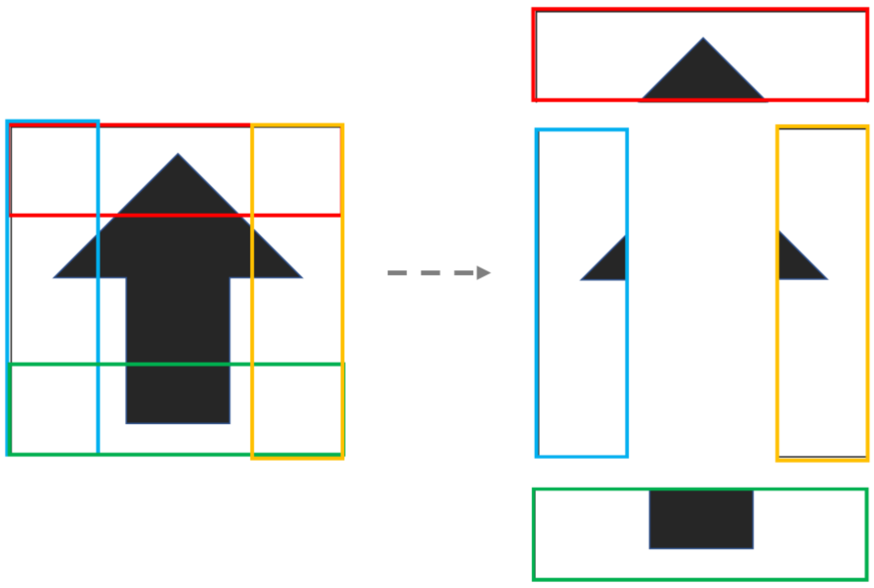
The detection target is a white arrow on a black background. An acceptable black color is first selected from the HSV color space. Based on the color range selected, a mask is created and applied to the camera frame to filter out non-black colors. In the resulting binary image, all black objects would appear white(targets) while object with other colors would appear black.



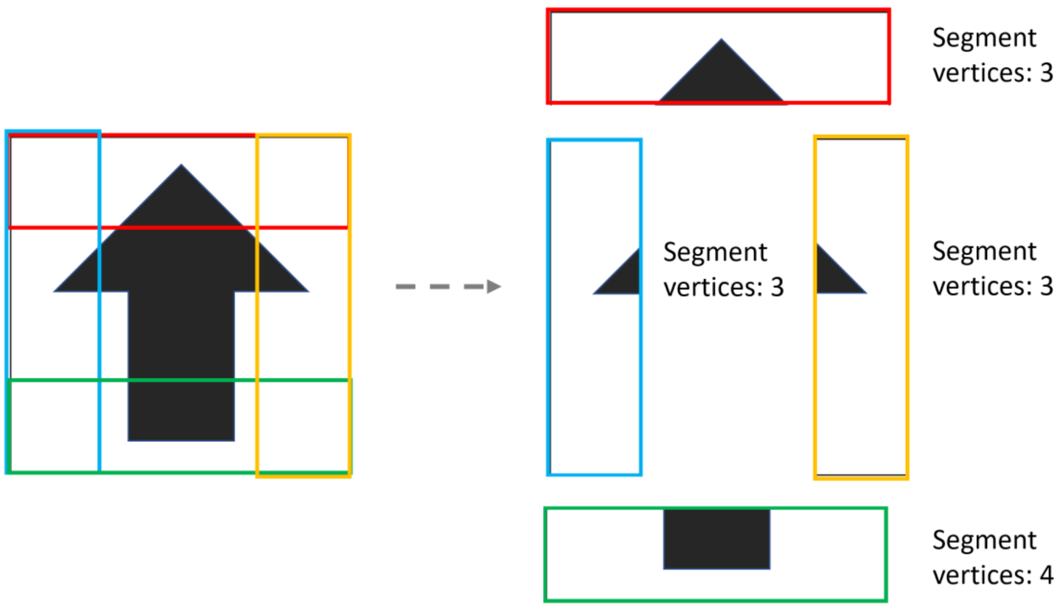
The distinct shape of an arrow can be observed from the binary images above. The OpenCV built-in method findContours() is then used to compute contours of all the segments in the binary image. To find the arrow, we search for the segment with 7 vertices.



If the ratio of the detected segment with respect to the entire image is within the predefined ratio range, we move on to the second check on arrow features. During the second check, 4 sub-segments are cropped from the 7-vertices segment.



Again we compute the contours and number of vertices in the sub-segments. Since the arrow is always pointing upwards during the actual run, we can expect the number of vertices to be computed as follows:



If the detected segment passes the second check, it would be considered a detected arrow and saved to the current directory.