Integrated Computer Vision System for Semi-Autonomous Robots

Graham Romero
Linfield College
grromero@linfield.edu

December 11, 2014

Abstract

In this project I explored the creation of a computer vision system. I successfully implemented basic region-based tracking using both a small Raspberry Pi-powered robot and my computer, which wirelessly communicate. The robot sends its camera data stream to the computer, and the computer performs the tracking and sends movement instructions back to the robot. The primary goal of creating this type of computer vision system was successfully met, and future goals include adding GPS and digital-compass sensors to allow positional and orientation tracking of the robot, so the data may be plotted. Additionally, another goal is to implement (via post-processing) scene reconstruction based on the data gathered.

Contents

List of Figures

1 Introduction / Literature Review

The overarching concept of computer vision (CV) is to have a machine gain an understanding about the surrounding environment based on input from a variety of sensors, and then to possibly react based on that knowledge. Topics in computer vision primarily focus on data obtained in the visible spectrum of light, and thus typical cameras may be used to acquire this data, although it's worth noting that many custom cameras and different types of sensors may be used.

There are a few major categories in computer vision. These include: image/video processing, object recognition, and scene reconstruction. In the first, the input is manipulated in a manner such that more information may be extracted, such as detecting edges or motion. In the second, key points and features are detected in the image and then compared to source data, in order to identify the objects in the scene. The final category, scene reconstruction, uses a combination of techniques from the previous sections, with many others added in. This category often uses a large series of images or video as input, rather than a single image. Using a variety of transformations and algorithms, a 3D scene is reconstructed. The Microsoft Kinect is a popular input source for this category, and it is sometimes used as the 'eyes' for robots. While each of these may seem like something most people wouldn't need or want, the reality is quite the contrary. It is being used in the medical field to assist with identifying how to train and automate the screening of pathology tests for cancer, reducing the time of diagnosis. Large industries are using it to inspect a product for quality control or to count a particular object (such as if packaging together X number of objects into a package). And it's also being used, albeit in a rather basic way, in cell phones - the stitching together of images to make panoramas, for example, requires computer vision algorithms.

The application of computer vision in robotics is a bit less widespread. The primary goal for the integration of computer vision and robotics is to create more autonomous robotics. Google's self-driving car is a well-established example, being completely autonomous, although with only a handful of these, the average person will not likely be using these very soon. Other companies have found an application of computer vision in unmanned aerial vehicles, allowing them to be semi-autonomous while tracking targets. It is this subset of robotics that I take inspiration for my project - although to reduce cost and unrelated technical difficulties, I will stick to a small land-based robot.

For the context of my capstone project, I will utilize the OpenCV Computer Vision library to create an object tracking system which communicates with a remote robot. I will acquire OpenCV from www.opencv.org and set it up with Visual Studio. To learn the setup process and basics of working with this library, I will follow the official OpenCV tutorials. After completing the basics, I will work on building the robot using the instructions on the Dawn Robotics blog. I will then need to work on getting the robot to communicate with the computer. Once this works I will use code provided by Dawn Robotics to enable streaming the images from the camera to my computer (basic instructions on how to do this). At this point I will need to transition my C++ code to Python, which will enable my computer to perform the processing on the image data, and then send commands back to the robot.

Extended goals for the project will be to have the robot not only track where the region is (assuming the tracked region exists), but to react to the location of it. An initial form of this will be to follow the object in some way. Another extended goal The next task is to be able to track where the object is moving, which should consist of not just along the x and y axis of the screen, but also the z-axis (depth - closer and farther from the camera). The (x,y) axis motion may be detected by having a previous and current frame saved, and on the current frame, I will visually mark where it currently is, where it was, and a motion vector between them. There are tutorials for how to draw to the screen at <docs.opencv.org>, so I will use this as a basis for adding the mark-up to the screen. The depth should be detectable by keeping track of the size of the object, and performing some geometry to determine how far it's moved relative to the (x,y) axis.

2 Building the Robot

The robot is identical to Dawn Robotics' Raspberry Pi Camera Robot. They provide very good instructions on how to build it. The key difference between their instructions and what I ended up doing was with the order of a couple things, and with how I connected hardware to the robot base. Therefore, here is the order I attached things in.

1. I first attached the wheels (normal and castor) to the bottom base piece.
2. I then assembled the sensor pan/tilt kit, per the instructions provided with the kit.
    a. I also attached the camera at this time.
3. I then attached the pan/tilt kit to the top base piece, per fig. 1.
4. Afterwards, I attached the Mini Driver and Raspberry Pi, per fig. 1.
5. The top base piece was then attached to the bottom base piece via the attachment points described in fig. 1. After attaching the corner connections, place the power bank between the frames, and then attach the back-center connection.
6. Now that all of the hardware is affixed to the base, you can now attach the wires to connect everything. This is described in fig. 2.

    a. Note that during this step I also added an LED light next to the camera on the pan-tilt kit, and I plugged in the LED power connections to the Raspberry Pi's GPIO pins #7 and #9 (see this documentation on the GPIO pins).

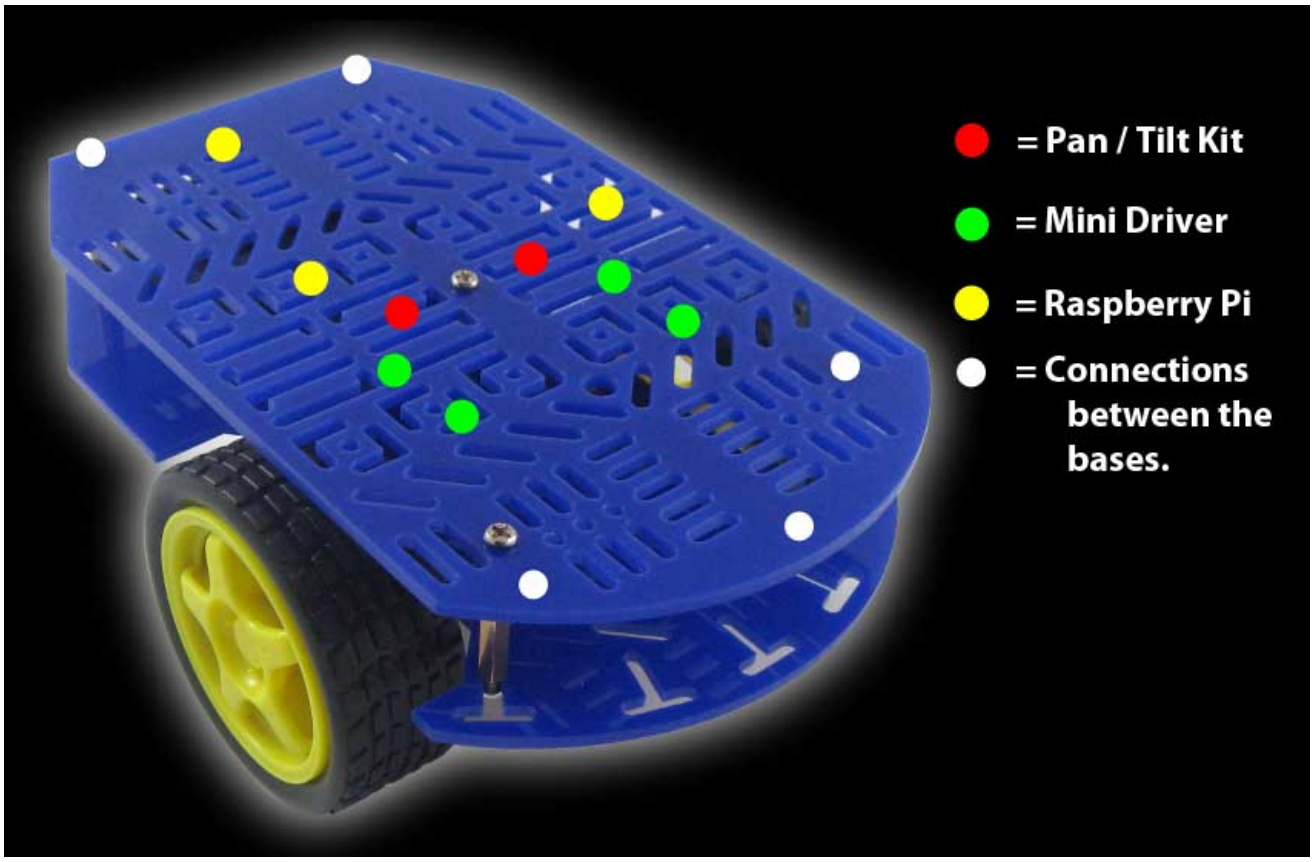

Figure 1: Diagram describing the attachment points for the hardware attaching to the base of the robot. Source: DawnRobotics. Edited by Graham Romero.

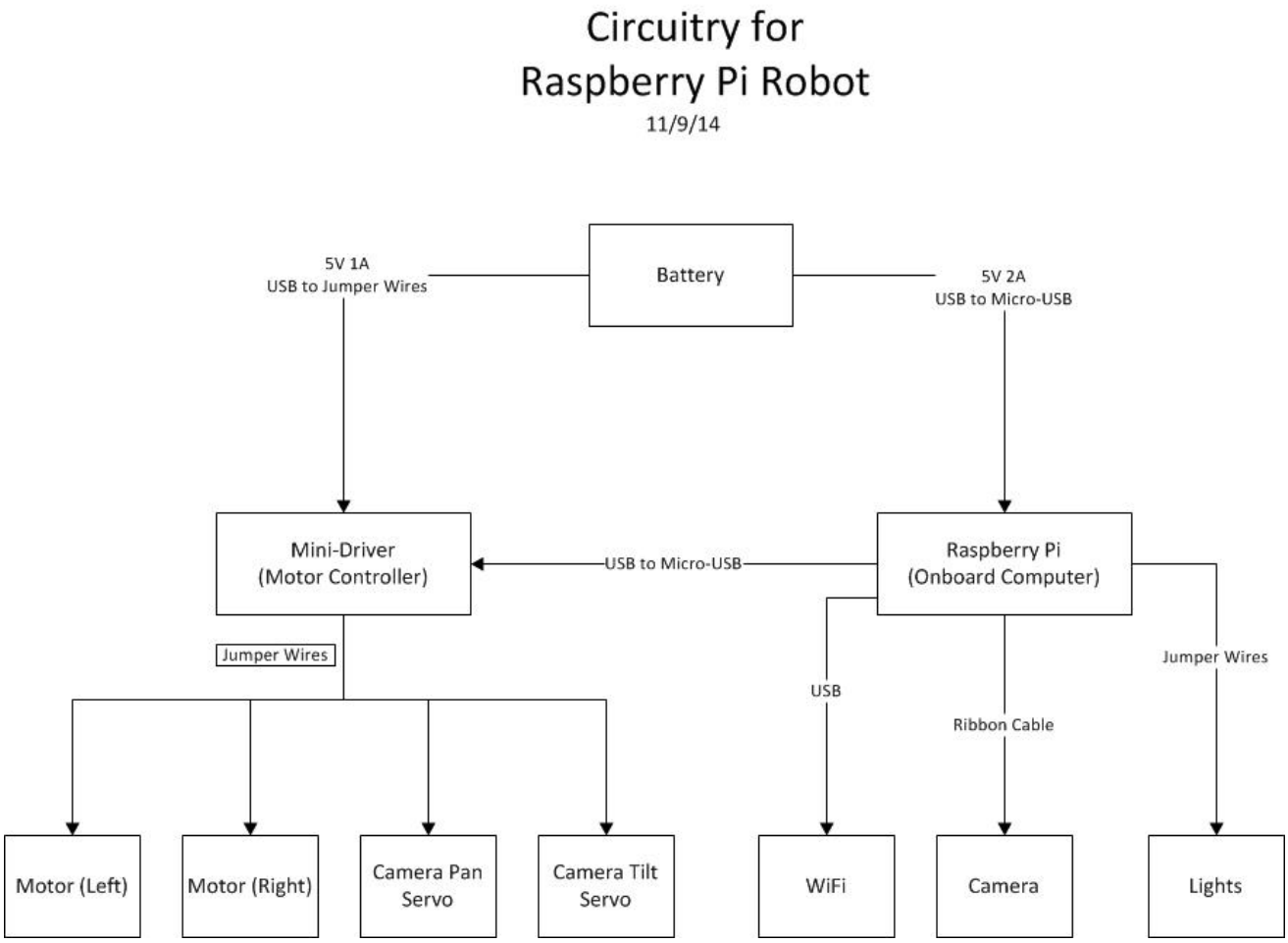

Figure 2: Diagram showing how the robot is wired together.

3 Process to Interface with the Robot

To get the software working on the robot, I directly purchased an SD card preloaded with the Dawn Robotics image, which automatically starts up a server that tries to stream images to any computers connected to it. Therefore, on the robot side, software was simple and required virtually no work.

On my computer, a bit more work was required, as follows. Note that I've only tested it on Windows computers, so I can't confirm whether or not it will work on other systems.

1. Download and Install Python 2.7.3 (32-bit).
2. Download pip (get-pip.py) and move it to the directory you installed Python to (I will be assuming C:/Python27/).
3. Now, open up the command prompt, and type the following commands:
   a. cd C:/Python27/
   b. python get-pip.py
   c. Tools/Scripts/win_add2path.py
4. Download and Install Numpy (numpy-1.6.2-win32-superpack-python2.7.exe).
5. Download OpenCV 2.3.1 Windows Superpack
   a. Unpack somewhere
   b. Open: opencv/build/python/2.7/
   c. Copy cv2.pyd to C:/Python27/lib/site-packages
6. Download and extract py_websockets_bot.
7. Now in the command prompt:
   a. cd <DIRECTORY THAT YOU EXTRACTED py_websockets_bot>
   b. C:/Python27/python setup.py
8. In the command prompt, use the following commands to install dependencies.
   a. pip install matplotlib

4 Process of Writing Code

Early on when working on this project, I realized that with the computer vision libraries I had available, tracking a simple color/shape combination would be much more difficult than tracking a complex shape with multiple colors, since the complex shape has more distinct regions. Therefore, rather than progressing my code by tracking a simple circle or square, and gradually coding it to track more complex objects, I instead went straight to having it track a selected region of the video stream, which would in theory contain the object you're trying to track (along with some extra space around it).

My code started separately from the robot, using c++, and just performing the tracking via a webcam. I first followed the Features2D tutorial on the OpenCV website. This tutorial takes a static source image and static target image and tries to find the target image in the source image. If found, it will try to draw a box around the new tracked region, transformed to match the new perspective.

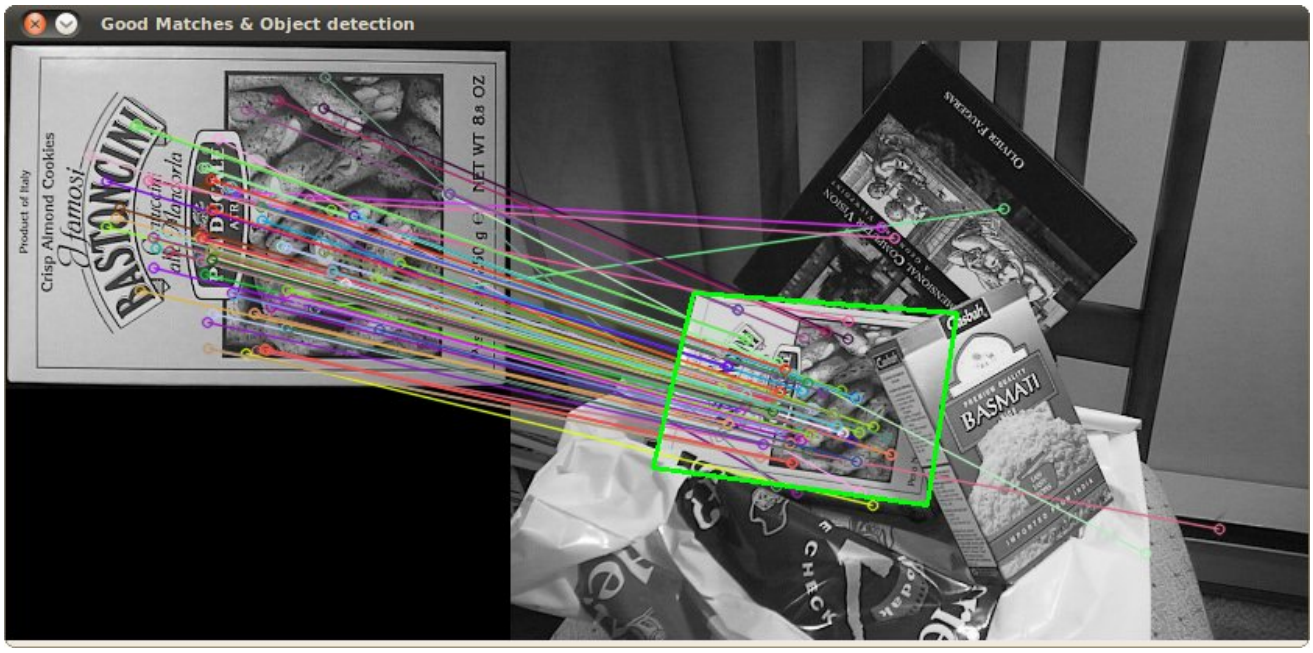

Figure 3: Example of basic tracking tutorial.
Source: OpenCV

I got this working fairly quickly, so I worked on editing the code to work with a video stream instead of a static source image, which was quickly successful. My next step was to be able to dynamically update the target region, so I needed a way to both select a Region-Of-Interest (ROI) given a set of coordinates and a method to get coordinates from mouse input.

At this time I received the robot hardware, which required using Python to interface with, so I started migrating my code to Python. To get it working quicker, I disabled some of what I'd worked on before, notably greying out part of the screen as you select a ROI. Migrating the code also required looking up the syntax for parts of the code in Python, specifically the ROI selection and adjusting the tracking algorithm.

Now that the tracking was working in Python, I wanted to be able to record it, so I needed a way of writing each frame to a video file. However, at this point my code was getting more challenging to manage, so I decided to start splitting parts of it into classes. I kept some code in the main file, but I moved much of the code to other classes, including the following: Mouse, Region, Tracker, and for the video files, Writer. Within the Writer class, I was able to create an empty video file fairly easily, actually writing each frame to that file proved to be more challenging, although the difficulties resulted from just the order of the video writing calls in my program.

Separately from the tracking code, but at the same time as writing the tracking code in Python, in section 3 I worked on using Python code to interface with the robot, and I wrote functions that allowed moving the robot via keyboard commands. Therefore, I felt like now was a good time to combine the two and have the robot perform tracking. Adjusting the tracking code to use the image matrix from the robot's video stream was a simple replacement of variables, and overall it required much fewer changes than I anticipated. As a result of this, the robot will now turn left or right, or tilt the camera up and down to keep the tracked region in the center of its view.

5 Structure of Code

I've taken an Object-Oriented-Design (OOD) approach to this project. I wanted each component of the code to function as independently of the rest as possible, allowing me to solely focus on creating or improving each component on its own. Therefore, I've created the following classes to use for this project:

- MainRobotCode - The 'main method' file that ties together the rest of the classes.
- Log - Writes data to log files (for debug mode).
- LogNull - Log class that does nothing (for runtime mode).
- Mouse - Manages mouse input.
- Region - Stores and manages selected regions of the video stream.
- RobotController - Manages the interface with the robot, sends commands for movement.
- Tracker - Tracks a given region in the video stream.
- Writer - Writes each frame of the video stream to an output file.

In the following sections below, I shall describe each in greater detail.

5.1 MainRobotCode

- **key_click** - Manages keyboard input, and calls functions or sets variables depending on which key is pressed.
- **draw_gui** - Draws the GUI onto the passed image.
- **output** - Calls draw_gui(), calls the video writer to write the displayed windows to the video file, and updates the currently displayed windows with the new image data.
- **shutdown** - Properly disconnects from the robot, closes the video files, and exits the displayed windows.
- **___main___** - This function creates the video writer streams, connects to the robot, and initializes the image streaming. It then loops to allow the program to run until the operator wishes for it to end. In this loop it makes calls to: retrieve the new image data, handle input, perform image processing, display output, and send movement commands to the robot.

5.2 Log

- **___init___** - This function creates the *logs* directory (if it doesn't already exist), sets the log filename, and sets the overwrite and debug mode variables.
- **write** - This function is called each time something is logged, and if it's currently in debug mode, it will either overwrite or append to the log file, depending on initialization settings. It writes the current time, the line number of the log message, and the message itself.

5.3 LogNull

This is structured identically to **Log**, except all of its functions do nothing.

5.4 Mouse

- **___init ___** - Sets variables to default values, creates a log file.
- **mouse_click** - When a mouse event occurs, this captures the location of the click both when the mouse is pressed down, and when it is released. Throughout this event, this method keeps track of what step we're currently on (such as pressing the mouse, moving it, but not yet releasing it). This data may be helpful in other parts of the code.
- **getMouseDown** - Returns the position the mouse was at when it was clicked down.
- **getMouseUp** - Returns the position of the mouse when the click was released.

5.5 Region

- **___init___** - Initializes the x, y, width, and height to zero, sets the minimum size of a tracking region, and creates a log file.
- **hasRegion** - Returns true if the width and height of the region are both greater than zero.
- **createNewRegion** - Sets the region coordinates (x, y, width, height) to the passed parameters.
- **drawRegion** - I disabled this code when I converted it to Python, but its primary original purpose was to draw a transparent box around the currently selected region.
- **getRegion** - Returns the image data for the selected region.
- **getShape** - Returns the shape of the selected region (width, height, and the number of color channels).
- **updateRegion** - Given two corner points, generate the proper x, y, width, and height values, and select that region from the current frame of the video stream.
- **drawRegionFilled** - Calls drawRegion with specific parameters set.
- **drawRegionDuringMatching** - Calls drawRegion with specific parameters set.
- **drawRegionNormal** - Calls drawRegion with specific parameters set.

5.6 RobotController

- **___init___** - Initializes constant values and positional values such as the speed, pan and tilt locations, and LED state. It also sets up the logging file and the timer, in addition to connecting to the robot via its API.
- **start_streaming** - Directly calls the start_streaming function in the robot's API.
- **get_latest_camera_image** - Directly calls the get_latest_camera_image function in the robot's API.
- **disconnect** - Directly calls the disconnect function in the robot's API.
- **update** - If the current action is completed, it stops the robot's motion. It also calls the update function in the robot's API.
- **set_led** - Given a boolean value, turns the LED on or off.

- **switch_led** - Turns the LED on if it's currently off, and off if it's currently on.
- **enable_fast_mode** - Sets the maximum movement speed to a higher-than-normal value (forward and backward motion should not affected, but turning and pivoting will be faster).
- **disable_fast_mode** - Resets the maximum movement speed to the normal value.
- **get_pan** - Returns the current pan value.
- **get_tilt** - Returns the current tilt value.
- **get_pan_center** - Returns 90.
- **get_tilt_center** - Returns 75.
- **set_movement_timer** - Given a number of ticks, sets the timer to the current time plus those ticks.
- **pantilt_up** - Note: The format for the following * functions are the same as this one, so for the sake of space I will not describe them again. It writes its action to the log file, then sets the tilt amount (relative to the current position), and then calls the robot's API to actually change the position.
- **pantilt_down*** - Tilts the pan-tilt sensor down.
- **pantilt_left*** - Pans the pan-tilt sensor left.
- **pantilt_right*** - Pans the pan-tilt sensor right.
- **stop_moving** - Via the robot's API, sets the motor speeds to zero.
- **move_forward*** - Note: The remaining * functions have the same format as this function. Given a number of ticks to perform the action, it writes its action to the log file, then calls set_movement_timer based on the given ticks, then sets the motor speeds. It moves the robot forward.
- **move_backward*** - Moves the robot backward.
- **turn_left*** - Turns the robot left (both wheels move at 1/6 normal speed, in opposite directions).
- **turn_right*** - Turns the robot right (both wheels move at 1/6 normal speed, in opposite directions).
- **pivot_left*** - Pivots the robot left (left motor speed: 0, right motor speed: 1/3 normal)
- **pivot_right*** - Pivots the robot right (left motor speed: 1/3 normal, right motor speed: 0)
- **pivot_back_left*** - Pivots the robot left (left motor speed: -1/3, right motor speed: 0)
- **pivot_back_right*** - Pivots the robot right (left motor speed: 0, right motor speed: -1/3)

5.7 Tracker

- **___init___** - Resets coordinate variables to [0,0], initializes the tracking region, SIFT tracker, and FLANN based matcher.
- **updateObject** - Given a set of coordinates, update the region to track.
- **hasObject** - Returns true if a tracking region is currently stored.
- **getCenter** - Returns the center of the keypoints in the tracked region.
- **getVariance** - Returns the variance of the keypoints in the tracked region.
- **getStdDev** - Returns the standard deviation of the keypoints in the tracked region.
- **isTracking** - Returns true if it's currently tracking the selected region in the video stream.
- **cancelTracking** - Resets the tracking region to a 1-pixel area, effectively turning off tracking.

- **match** - Given a frame of the video stream, it calculates keypoints in that image, then tries to match them to keypoints in the selected tracking region. If enough good matches are made, then tracking is successful, and data about the tracking is updated (center of points, variance, standard deviation). Additionally, an X is drawn at the center of the tracked region.
- **drawMatches** - This function was modified from the one posted at StackOverflow. It draws small circles on the keypoints, and it draws lines between the associated keypoints in both images.

5.8 Writer

- **___init___** - Sets up the video writer object from OpenCV (in this case it uses the XVID codec). The output folder is created if it doesn't already exist, and the video file's name is based on the current date and time, and a name parameter is concatenated.
- **write** - writes the current frame to the output file.
- **release** - closes the video writing stream and logs this action.

6 Conclusions

Overall, this project was way more successful than I initially thought it would be. That being said, if I were to redo the project, I would have started coding in Python so I grew into familiarity with the language. With the way I did this project, I didn't really use Python until after I had the tracking code complete, at which point I literally copied and pasted my C++ code in, and edited it until it worked.

Additionally, if I had more time to work on this project, there are many other features and optimizations I would have liked to add in. The most notable is to add GPS and digital compass sensors to the robot so I can stream its location and orientation in space to the operator. This will allow for graphically mapping the history of the robot's path, and through some testing, hopefully an approximate relative position of the tracked object. Another extension would be to add a second camera to get stereoscopic (3D) data. It may be more feasible to purchase a two-camera system instead. A very far-reaching goal I have that would be dependent on the first and possibly second goals would be to implement post-processing that uses the spacial information with the video stream to reconstruct a rough 3D scene of where the robot was. I would like to implement this where it may be exported as a .OBJ or related filetype so I could, for example, import the file into Unity and allow a person to virtually "walk through " the region.

Some future applications that work such as this could progress to include remote 3D modelling of structures after disasters such as earthquakes, allowing operators to estimate where it's safe to send emergency response workers in. If the current goal of reconstructing the 3D scene can be done in real-time, perhaps by improving hardware and the algorithms, an operator could use a head-mounted-display such as the Oculus Rift, allowing the robot to essentially be a surrogate for the operator. I see this application going hand-in-hand with a robot such as Atlas from Boston Dynamics.

7 Code

7.1 !runRobotCode.cmd

```
C:\Python27\python.exe MainRobotCode.py
Pause
```

7.2 MainRobotCode.py

```
import cv2
import time
import argparse
import math
# My Classes
import RobotController
import Mouse
import Writer
import Log
import Tracker

import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

global TRACKING_WINDOW_NAME
global ORIGINAL_WINDOW_NAME
global target
global myMouse
global myTracker
global show_tracking
global fast_on
global turning_delay
TRACKING_WINDOW_NAME = 'Image'
ORIGINAL_WINDOW_NAME = 'Original'
myMouse = Mouse.Mouse()
myTracker = Tracker.Tracker()
myLog = Log.Log("Main.txt",False)
show_tracking = True
fast_on = False
turning_delay = 0

global image
global original

def key_click(key):
   if (key == -1):
      return ""
   #print key
   if (key == 27 or key == 8): # Esc or Backspace
      return "exit"

   # PAN TILT KEYS
   if (key == 2490368): # UP
      rc.pantilt_up()
   if (key == 2621440): # DOWN
      rc.pantilt_down()
   if (key == 2424832): # LEFT
```

```
      rc.pantilt_left()
   if (key == 2555904): # RIGHT
      rc.pantilt_right()

   move_time = 200
   # MOTOR KEYS
   if (key == 119 or key == 87): # W or w
      rc.move_forward(move_time)
   if (key == 115 or key == 83): # S or s
      rc.move_backward(move_time)
   if (key == 97 or key == 65): # A or a
      rc.turn_left(move_time/2)
   if (key == 100 or key == 68): # D or d
      rc.turn_right(move_time/2)
   if (key == 113 or key == 81): # Q or q
      rc.pivot_left(move_time)
   if (key == 101 or key == 69): # E or e
      rc.pivot_right(move_time)
   if (key == 122 or key == 90): # Z or z
      rc.pivot_back_left(move_time)
   if (key == 99 or key == 67): # C or c
      rc.pivot_back_right(move_time)
   if (key == 32): # Space
      rc.stop_moving()
   if (key == 49): # 1
      global show_tracking
      show_tracking = not show_tracking
   if (key == 50): # 2
      global fast_on
      fast_on = not fast_on
      if (fast_on):
         rc.enable_fast_mode()
      else:
         rc.disable_fast_mode()
   if (key == 51): # 3
      rc.switch_led()
   if (key == 52): # 4
      myTracker.updateObject(original, 0, 0, 1, 1)
   return ""

def draw_gui(original):
   HEIGHT, WIDTH, CHANNELS = original.shape
   # TEXT
   cv2.putText(original, "Tilt: " + str(rc.get_tilt()), (WIDTH - 150, 50), cv2.FONT_HERSHEY_PLAIN,
       2, (0,0,255), 1)
   cv2.putText(original, "Pan: " + str(rc.get_pan()), (WIDTH - 150, 100), cv2.FONT_HERSHEY_PLAIN,
       2, (0,0,255), 1)
   if (myTracker.isTracking() == True):
      cv2.putText(original, "Tracking!", (WIDTH - 150, 150), cv2.FONT_HERSHEY_PLAIN, 1, (0,0,255),
          1)
      #TARGET
      center = myTracker.getCenter()
      cv2.line(original,(center[1]-10,center[0]-10),(center[1]+10,center[0]+10),(0,0,255),5)
      cv2.line(original,(center[1]-10,center[0]+10),(center[1]+10,center[0]-10),(0,0,255),5)
      #rc.switch_led()
   else:
      cv2.putText(original, "Not Tracking!", (WIDTH - 150, 150), cv2.FONT_HERSHEY_PLAIN, 1,
          (0,0,255), 1)
   return original
```

```
def output():
   global original
   # Draw onto the image
   draw_gui(original)
   # Write frame to video file.
   origVideoWriter.write(original)
   # Display the image
   cv2.imshow( ORIGINAL_WINDOW_NAME, original )
   global show_tracking
   if (show_tracking):
      if (myTracker.isTracking()):
         image = myTracker.drawMatches()
         image = cv2.resize(image, (840,480))
         editVideoWriter.write(image)
         cv2.imshow(TRACKING_WINDOW_NAME, image)
   else:
      cv2.destroyWindow(TRACKING_WINDOW_NAME)

def shutdown():
   # Disconnect from the robot
   myLog.write(lineno(), "Disconnecting...")
   rc.disconnect()
   origVideoWriter.release()
   editVideoWriter.release()
   cv2.destroyAllWindows()

#-----------------------------------------------------------------------------------------------
if __name__ == "__main__":
   global image
   global turning_delay
   origVideoWriter = Writer.VideoWriter('original', 10, (640, 480))
   editVideoWriter = Writer.VideoWriter('edited', 10, (840, 480))
   # Set up a parser for command line arguments
   #parser = argparse.ArgumentParser( "Gets an image from the robot" )
   #parser.add_argument( "hostname", default="localhost", nargs='?', help="The ip address of the
       robot" )
   #args = parser.parse_args()
   # Connect to the robot
   #rc = RobotController.RobotController(args.hostname)
   rc = RobotController.RobotController("192.168.42.1")
   # Start streaming images from the camera
   rc.start_streaming()
   image, image_time = rc.get_latest_camera_image()
   original = image.copy()
   image_rows = image.shape[0]
   image_cols = image.shape[1]
   CAMERA_WIDTH = image_cols
   CAMERA_HEIGHT = image_rows
   cv2.imshow(TRACKING_WINDOW_NAME, image )
   cv2.imshow(ORIGINAL_WINDOW_NAME, original )
   cv2.setMouseCallback(ORIGINAL_WINDOW_NAME , myMouse.mouse_click)
   is_done = False

   movementTimeThreshold = 10
   currentMovementTime = 0

   while (is_done == False):
      rc.update()
```

```
    # Get an image from the robot
    image, image_time = rc.get_latest_camera_image()
    #print "Image Dimensions: ", image.shape
    image_rows = image.shape[0]
    image_cols = image.shape[1]
    image_channels = image.shape[2]

    # Flip if needed so image is always right-side-up
    if (rc.get_tilt() > rc.get_tilt_center()):
       M = cv2.getRotationMatrix2D((image_cols/2, image_rows/2), 180, 1)
       image = cv2.warpAffine(image, M, (image_cols, image_rows))
    # Copy to show original.
    original = image.copy()

    if (myMouse.mouse_event_occured):
       md = myMouse.getMouseDown()
       mu = myMouse.getMouseUp()
       myTracker.updateObject(original, md[1], md[0], mu[1], mu[0])
       myMouse.mouse_event_occured = False
    success = False
    if (myMouse.capture_in_progress == False):
       if (myTracker.hasObject()):
          myTracker.match(image)
    output()

    # MOVE ROBOT
    turning_delay += 1
    center = myTracker.getCenter()
    if (center[1] > 0 and turning_delay > 5):
       if (center[1] < original.shape[1] * 2 / 5):
          rc.turn_left(100)
          turning_delay = 0
       if (center[1] > original.shape[1] * 3 / 5):
          rc.turn_right(100)
          turning_delay = 0
       center[1] = 0
    if (center[0] > 0):
       if (center[0] < original.shape[0] * 2 / 5):
          rc.pantilt_up()
       if (center[0] > original.shape[0] * 3 / 5):
          rc.pantilt_down()
       center[0] = 0

    # Wait for the user to press a key
    isMovementKey = False
    key_result = key_click(cv2.waitKey( 15 ))
    if (key_result == "exit"):
       is_done = True
 shutdown()
```

7.3 Log.py

```
import os
import time

class LogNull:
   def __init__(self,filename,overwrite):
      abc = 0
      # Do Nothing
   def write(self, message):
      abc = 0
      # Do Nothing
   def write(self, line, message):
      abc = 0
      # Do Nothing

class Log:
   def __init__(self,filename,overwrite):
      self.filename = filename
      dirpath = os.path.join(os.path.dirname(__file__),'logs')
      if not os.path.exists(dirpath):
         os.makedirs(dirpath)
      self.filepath = os.path.join(dirpath, filename)
      self.overwrite = overwrite
      self.DEBUG_MODE = True

   """def write(self, message):
      if (self.DEBUG_MODE):
         cur_time = time.strftime("%Y-%m-%d_%H.%M.%S")
         with open(self.filepath, "a") as myfile:
            myfile.write("Time: "+cur_time+", Message: "+message+"\n")"""

   def write(self, line, message):
      if (self.DEBUG_MODE):
         cur_time = time.strftime("%Y-%m-%d_%H.%M.%S")
         with open(self.filepath, "a") as myfile:
            myfile.write("Time: "+cur_time+", Line: "+str(line)+", Message: "+message+"\n")
```

7.4 Mouse.py

```
import cv2
import Log

import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

class Mouse:
   def __init__(self):
      self.capture_in_progress = False
      self.mouse_event_occured = False
      self.draw_capture = False
      self.mouse_down = (0,0)
      self.mouse_up = (0,0)
      self.myLog = Log.Log('Mouse.txt',False)

   def mouse_click(self,event,x,y,flags,param):
      #print "Mouse Event Occurred!"
      if (event == cv2.EVENT_MOUSEMOVE):
         if (self.capture_in_progress):
            self.mouse_up = (y, x)
            #mouse_event_occured = True
            #print "--Mouse Moved: (", mouse_up[0], ",", mouse_up[1], ")"
         return
      if (event == cv2.EVENT_LBUTTONDOWN):
         if (self.capture_in_progress == False):
            self.mouse_down = (y, x)
            self.capture_in_progress = True
            self.draw_capture = False
            self.myLog.write(lineno(),"Mouse Button Down!")
         return
      if (event == cv2.EVENT_LBUTTONUP):
         if (self.capture_in_progress):
            self.mouse_up = (y, x)
            self.capture_in_progress = False
            self.mouse_event_occured = True
            self.draw_capture = True
            #self.myLog.write("Captured: ("+str(self.mouse_down[0])+", "+str(self.mouse_down[1])+")
                -> ("+str(self.mouse_up[0])+","+str(self.mouse_up[1])+")")
         return
      return

   def getMouseDown(self):
      return self.mouse_down
   def getMouseUp(self):
      return self.mouse_up
```

7.5 Region.py

```
import Log

import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

class Region:
   def __init__(self):
      self.x = 0
      self.y = 0
      self.w = 0
      self.h = 0
      #self.regionImage
      self.MIN_CAPTURE_WIDTH = 100
      self.MIN_CAPTURE_HEIGHT = 100;
      self.BLUE_CHANNEL = 0
      self.GREEN_CHANNEL = 1
      self.RED_CHANNEL = 2;
      self.myLog = Log.Log('Region.txt',False)
   def hasRegion(self):
      return (self.w > 0 and self.h > 0)
   def createNewRegion(self, nx, ny, nw, nh):
      self.x = nx
      self.y = ny
      self.w = nw
      self.h = nh
   def drawRegion(self, src, FILL_CAPTURE, DURING_MATCHING):
      """result = src[0:src.shape[1], 0:src.shape[0]]
      OFFSET = 0
      if (DURING_MATCHING):
         OFFSET = self.w
      alpha = 0.6 # Between 1.0 - 3.0
      beta = 5.0 # Between 0 - 100
      #CV_Assert(src.depth() == cv2.CV_8U) # Accept only uchar images.
      nChannels = src.shape[2]
      print self.x, self.y, self.w, self.h
      result = src.copy()
      for cy in range(0, src.shape[0]-1):
         for cx in range(0, src.shape[1]-1):
            for c in range(0, nChannels-1):
               if (cy >= self.y and cy < self.y + self.h and cx >= self.x and cx < self.x + self.w):
                  if (FILL_CAPTURE):
                     #result[cx + OFFSET, cy, c] = regionImage[cx - self.x, cy - self.y, c]
                     result[cy, cx + OFFSET, c] = regionImage[cy - self.y, cx - self.x, c]
                  else:
                     if (self.w > self.MIN_CAPTURE_WIDTH and self.h > self.MIN_CAPTURE_HEIGHT and c
                         == self.GREEN_CHANNEL):
                        result[cy, cx + OFFSET, c] = alpha*(src[cy, cx + OFFSET, c]) + beta + 50
                     else:
                        result[cy, cx + OFFSET, c] = alpha*(src[cy, cx + OFFSET, c]) + beta
               #else:
                  #result[cy, cx] = src[cy, cx]
                  #result[0:cx, 0:cy] = src[0:cx, 0:cy]"""
   def getRegion(self):
      return self.regionImage
   def getShape(self):
      return self.regionImage.shape
```

```
def updateRegion(self, src, dx, dy, ux, uy):
   #self.myLog.write("("+str(dx)+", "+str(dy)+") ("+str(ux)+", "+str(uy)+")")
   #self.myLog.write("("+str(src.shape[1])+", "+str(src.shape[0])+")")
   # Limit mouse_down region to bounds of screen.
   if (dx < 0):
      dx = 0
   if (dy < 0):
      dy = 0
   if (dx >= src.shape[1]):
      dx = src.shape[1] - 1
   if (dy >= src.shape[0]):
      dy = src.shape[0] - 1
   # Limit mouse_up region to bounds of screen.
   if (ux < 0):
      ux = 0
   if (uy < 0):
      uy = 0
   if (ux >= src.shape[1]):
      ux = src.shape[1] - 1
   if (uy >= src.shape[0]):
      uy = src.shape[0] - 1

   # Make (x,y) top left and (x+w,y+h) bottom right.
   self.x = min(dx,ux)
   self.y = min(dy,uy)
   self.w = abs(ux-dx)
   self.h = abs(uy-dy)

   #roi = src[self.x: self.x+self.w, self.y: self.y+self.h].copy()
   roi = src[self.y: self.y+self.h, self.x: self.x+self.w].copy()
   self.regionImage = roi
   #self.myLog.write("roi: " + str(self.regionImage.shape))
   #self.myLog.write(str(self.x)+' '+str(self.y)+' '+str(self.w)+' '+str(self.h))
def drawRegionFilled(self, src):
   self.drawRegion(src, True, False)
def drawRegionDuringMatching(self, src):
   self.drawRegion(src, False, True)
def drawRegionNormal(self, src):
   self.drawRegion(src, False, False)
```

7.6 RobotController.py

```
import py_websockets_bot
import time
import Log

import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

# Use: import RobotController
# Use: rc = RobotController.RobotController(hostname)
class RobotController:
   def __init__(self, hostname): # 'Constructor'
      self.bot = py_websockets_bot.WebsocketsBot( hostname )
      # CONSTANTS
      self.MIN_MOVEMENT_SPEED = -100 # -100 # Debug: -15
      self.MAX_MOVEMENT_SPEED = 100 # 100 # Debug: 15
      self.TILT_MIN = 0
      self.TILT_MAX = 165
      self.PAN_MIN = 0
      self.PAN_MAX = 180
      # CURRENT VALUES
      self.pan = 90
      self.tilt = 150
      self.tilt_mod = 2
      self.bot.set_neck_angles( self.pan, self.tilt ) # (pan, tilt)
      # TIMERS
      self.timer_movement = 0
      #self.myLog = Log.Log('RobotController.txt',False)
      self.myLog = Log.Log('RobotController.txt',False)
      self.led_onoff = 0

   # PASS-THROUGH METHODS
   def start_streaming(self):
      self.bot.start_streaming_camera_images()
   def get_latest_camera_image(self):
      return self.bot.get_latest_camera_image()
   def disconnect(self):
      self.bot.disconnect()
   def update(self):
      if (self.timer_movement - time.clock() < 0):
         self.stop_moving()
      self.bot.update()
   def set_led(self, on_off):
      self.led_onoff = on_off
      self.bot.set_led(self.led_onoff)
   def switch_led(self):
      if (self.led_onoff == 0):
         self.set_led(1)
      else:
         self.set_led(0)

   # OTHERS
   def enable_fast_mode(self):
      self.MAX_MOVEMENT_SPEED = 200
   def disable_fast_mode(self):
      self.MAX_MOVEMENT_SPEED = 100
```

```
# GETTERS
def get_pan(self):
   return self.pan
def get_tilt(self):
   return self.tilt
def get_pan_center(self):
   return 90
def get_tilt_center(self):
   return 75

# TIMER METHODS
def set_movement_timer(self, ticks):
   self.timer_movement = time.clock() + float(ticks)/1000

# PAN TILT MOTION
def pantilt_up(self):
   self.myLog.write(lineno(), "PANTILT_Up")
   self.tilt = max(self.tilt - self.tilt_mod, self.TILT_MIN)
   self.bot.set_neck_angles(self.pan, self.tilt)
def pantilt_down(self):
   self.myLog.write(lineno(), "PANTILT_Down")
   self.tilt = min(self.tilt + self.tilt_mod, self.TILT_MAX)
   self.bot.set_neck_angles(self.pan, self.tilt)
def pantilt_left(self):
   self.myLog.write(lineno(), "PANTILT_Left")
   self.pan = min(self.pan + 30, self.PAN_MAX)
   self.bot.set_neck_angles(self.pan, self.tilt)
def pantilt_right(self):
   self.myLog.write(lineno(), "PANTILT_Right")
   self.pan = max(self.pan - 30, self.PAN_MIN)
   self.bot.set_neck_angles(self.pan, self.tilt)

# WHEELS MOTION
def stop_moving(self):
   #print "MOVE_Stop"
   self.bot.set_motor_speeds( 0, 0 )
def move_forward(self, ms):
   self.myLog.write(lineno(), "MOVE_Move_Forward")
   self.set_movement_timer(ms)
   self.bot.set_motor_speeds( -self.MAX_MOVEMENT_SPEED, -self.MAX_MOVEMENT_SPEED )
def move_backward(self, ms):
   self.myLog.write(lineno(), "MOVE_Move_Backward")
   self.set_movement_timer(ms)
   self.bot.set_motor_speeds( self.MAX_MOVEMENT_SPEED, self.MAX_MOVEMENT_SPEED )
def turn_left(self, ms):
   self.myLog.write(lineno(), "MOVE_Turn_Left")
   self.set_movement_timer(ms)
   self.bot.set_motor_speeds( -self.MAX_MOVEMENT_SPEED/6, self.MAX_MOVEMENT_SPEED/6 )
def turn_right(self, ms):
   self.myLog.write(lineno(), "MOVE_Turn_Right")
   self.set_movement_timer(ms)
   self.bot.set_motor_speeds( self.MAX_MOVEMENT_SPEED/6, -self.MAX_MOVEMENT_SPEED/6 )
def pivot_left(self, ms):
   self.myLog.write(lineno(), "MOVE_Pivot_Left")
   self.set_movement_timer(ms)
   self.bot.set_motor_speeds( 0, self.MAX_MOVEMENT_SPEED/3 )
def pivot_right(self, ms):
   self.myLog.write(lineno(), "MOVE_Pivot_Right")
   self.set_movement_timer(ms)
```

```
    self.bot.set_motor_speeds( self.MAX_MOVEMENT_SPEED/3, 0 )
  def pivot_back_left(self, ms):
    self.myLog.write(lineno(), "MOVE_Pivot_Left")
    self.set_movement_timer(ms)
    self.bot.set_motor_speeds( -self.MAX_MOVEMENT_SPEED/3, 0 )
  def pivot_back_right(self, ms):
    self.myLog.write(lineno(), "MOVE_Pivot_Right")
    self.set_movement_timer(ms)
    self.bot.set_motor_speeds( 0, -self.MAX_MOVEMENT_SPEED/3 )
```

7.7 Tracker.py

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
import math

import Region
import Log
import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

class Tracker:
   def __init__(self):
      self.center = [0,0]
      self.variance = [0,0]
      self.std_dev = [0,0]
      self.is_tracking = False
      self.myRegion = Region.Region()
      self.sift = cv2.SIFT(0, 3, 0.04, 10, 1.6)
      self.FLANN_INDEX_KDTREE = 0
      self.index_params = dict(algorithm = self.FLANN_INDEX_KDTREE, trees = 5)
      self.search_params = dict(checks = 50)
      self.flann = cv2.FlannBasedMatcher(self.index_params, self.search_params)

   def updateObject(self, src, x1, y1, x2, y2):
      self.myRegion.updateRegion(src, x1, y1, x2, y2)
      self.object = cv2.cvtColor(self.myRegion.getRegion(), cv2.COLOR_BGR2GRAY)
      # Initiate SIFT detector
      #sift = cv2.SIFT()
      if (abs(x1-x2) < 50 or abs(y1-y2) < 50):
         is_tracking = False
         return
      self.kp1, self.des1 = self.sift.detectAndCompute(self.object,None)
   def hasObject(self):
      return self.myRegion.hasRegion()
   def getCenter(self):
      return self.center
   def getVariance(self):
      return self.variance
   def getStdDev(self):
      return self.std_dev
   def isTracking(self):
      return self.is_tracking
   def cancelTracking(src):
      updateObject(src, 0, 0, 1, 1)
   def match(self, img2):
      rs = self.myRegion.getShape()
      if (rs[0] < 50 or rs[1] < 50):
         return

      MIN_MATCH_COUNT = 10
      self.is_tracking = False
      self.center = [0,0]
      self.variance = [0,0]
      self.std_dev = [0,0]
      self.img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
```

```
# find the keypoints and descriptors with SIFT
self.kp2, self.des2 = self.sift.detectAndCompute(self.img2,None)
if (len(self.kp2) < 25):
   return

matches = self.flann.knnMatch(np.asarray(self.des1,np.float32),np.asarray(self.des2,np.float32
    ),k=2)

# store all the good matches as per Lowe's ratio test.
self.good = []
for m,n in matches:
   if m.distance < 0.7*n.distance:
      self.good.append(m)
if len(self.good)>MIN_MATCH_COUNT:
   self.is_tracking = True
   #print "GOOD"
   src_pts = np.float32([ self.kp1[m.queryIdx].pt for m in self.good ])#.reshape(-1,1,2)
   dst_pts = np.float32([ self.kp2[m.trainIdx].pt for m in self.good ])#.reshape(-1,1,2)

   #http://stackoverflow.com/questions/14995512/opencv-how-to-find-the-center-of-mass-centroid
       -for-motion-information
   for m in self.good:
      self.center[0] = self.center[0] + self.kp2[m.trainIdx].pt[1]
      self.center[1] = self.center[1] + self.kp2[m.trainIdx].pt[0]
   self.center[0] = self.center[0] / len(self.good)
   self.center[1] = self.center[1] / len(self.good)
   for m in self.good:
      self.variance[0] = self.variance[0] + pow(self.kp2[m.trainIdx].pt[1] - self.center[0],2)
      self.variance[1] = self.variance[1] + pow(self.kp2[m.trainIdx].pt[0] - self.center[1],2)
   self.variance[0] = self.variance[0] / len(self.good)
   self.variance[1] = self.variance[1] / len(self.good)
   self.std_dev[0] = math.sqrt(self.variance[0])
   self.std_dev[1] = math.sqrt(self.variance[1])

   #M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
   #matchesMask = mask.ravel().tolist()

   #h,w = img1.shape
   h = self.object.shape[0]
   w = self.object.shape[1]
   pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
   #dst = cv2.perspectiveTransform(pts,M)

   #img2 = cv2.polylines(img2,[np.int32(dst)],True,255,3, cv2.CV_AA)
   #cv2.polylines(img2,[np.int32(dst)],True,255,3, cv2.CV_AA)

   self.center[0] = int(self.center[0])
   self.center[1] = int(self.center[1])
   if (self.center[0] > 10 and self.center[1] > 10 and self.center[0] < self.img2.shape[1] -
       10 and self.center[1] < self.img2.shape[0] - 10):
      self.is_tracking = True
      cv2.line(self.img2,(self.center[1]-10,self.center[0]-10),(self.center[1]+10,self.center
          [0]+10),(0,0,0),5)
      cv2.line(self.img2,(self.center[1]-10,self.center[0]+10),(self.center[1]+10,self.center
          [0]-10),(0,0,0),5)
   else:
      self.is_tracking = False

#else:
```

```
        #print "Not enough matches are found - %d/%d" % (len(good),MIN_MATCH_COUNT)
        #matchesMask = None
    #draw_params = dict(matchColor = (0,255,0), # draw matches in green color
    #             singlePointColor = None,
    #             matchesMask = matchesMask, # draw only inliers
    #             flags = 2)

    #img3 = cv2.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
    #return True
    #return img3
    return

#http://stackoverflow.com/questions/20259025/module-object-has-no-attribute-drawmatches-opencv-
    python
def drawMatches(self):
    """
    My own implementation of cv2.drawMatches as OpenCV 2.4.9
    does not have this function available but it's supported in
    OpenCV 3.0.0

    This function takes in two images with their associated
    keypoints, as well as a list of DMatch data structure (matches)
    that contains which keypoints matched in which images.

    An image will be produced where a montage is shown with
    the first image followed by the second image beside it.

    Keypoints are delineated with circles, while lines are connected
    between matching keypoints.

    img1,img2 - Grayscale images
    kp1,kp2 - Detected list of keypoints through any of the OpenCV keypoint
           detection algorithms
    matches - A list of matches of corresponding keypoints through any
           OpenCV keypoint matching algorithm
    """

    # Create a new output image that concatenates the two images together
    # (a.k.a) a montage
    rows1 = self.object.shape[0]
    cols1 = self.object.shape[1]
    rows2 = self.img2.shape[0]
    cols2 = self.img2.shape[1]

    image = np.zeros((max([rows1,rows2]),cols1+cols2,3), dtype='uint8')

    # Place the first image to the left
    image[:rows1,:cols1,:] = np.dstack([self.object, self.object, self.object])

    # Place the next image to the right of it

    image[:rows2,cols1:cols1+cols2,:] = np.dstack([self.img2, self.img2, self.img2])

    # For each pair of points we have between both images
    # draw circles, then connect a line between them
    for mat in self.good:

        # Get the matching keypoints for each of the images
        img1_idx = mat.queryIdx
```

```
            img2_idx = mat.trainIdx

            # x - columns
            # y - rows
            (x1,y1) = self.kp1[img1_idx].pt
            (x2,y2) = self.kp2[img2_idx].pt

            # Draw a small circle at both co-ordinates
            # radius 4
            # colour blue
            # thickness = 1
            cv2.circle(image, (int(x1),int(y1)), 4, (255, 0, 0), 1)
            cv2.circle(image, (int(x2)+cols1,int(y2)), 4, (255, 0, 0), 1)

            # Draw a line in between the two points
            # thickness = 1
            # colour blue
            cv2.line(image, (int(x1),int(y1)), (int(x2)+cols1,int(y2)), (255, 0, 0), 1)
        return image
```

7.8 Writer.py

```
import cv2
import time
import os
import Log

import inspect
def lineno(): return inspect.currentframe().f_back.f_lineno

class VideoWriter:
   def __init__(self, name, fps, size):
      self.fourcc = cv2.cv.CV_FOURCC(*'XVID')
      self.filename = 'output_'+time.strftime("%Y-%m-%d_%H.%M.%S")+'_'+name+'.avi'
      dirpath = os.path.join(os.path.dirname(__file__),'output')
      if not os.path.exists(dirpath):
         os.makedirs(dirpath)
      filepath = os.path.join(dirpath,self.filename)
      self.out = cv2.VideoWriter(filepath,self.fourcc, fps, size)
      self.myLog = Log.Log('Writer.txt',False)
      self.myLog.write(lineno(), "Created video file, "+self.filename)
   def write(self, frame):
      self.out.write(frame)
   def release(self):
      self.out.release()
      self.myLog.write(lineno(), "Closed video file, "+self.filename)
```

8 References

Alan. *Dawn Robotics*. n.p. Web. 23 Sept. 2014.
Alan. *Read the Docs*. n.p. Web. 23 Sept. 2014.
"Bitbucket. " *DawnRobotics / Py_websockets_bot*. Bitbucket, 28 July 2014. Web. 23 Sept. 2014.
"Computer Vision. " *TheFreeDictionary*. Farlex, Inc., 2002. Web. 11 Oct. 2014.
"Feature Homography Result." Screenshot. *OpenCV Tutorials*. n.p. Web. 11 Nov. 2014.
"New Servo Wiring." Photograph. *Dawn Robotics*. n.p. Web. 11 Oct. 2014.
OpenCV. n.p. Web. 23 Sept. 2014.
Nuttall, Ben. "GPIO. " *Raspberry Pi Documentation*. Raspberry Pi Foundation, 18 Sept. 2014. Web. 3 Nov. 2014.
Rayryeng. "module' object has no attribute 'drawMatches' opencv python " #26227854. 7 Oct. 2014. Forum post.
Vernon, David. "Vernon's Machine Vision. " *Vernon's Machine Vision Online*. ECVision Network on Cognitive Computer Vision, 2 May 2005. Web. 11 Oct. 2014.

Specific links from the websites in the list of references:

- Tutorial for building and setting up the Raspberry Pi Camera Robot
- Dawn Robotics' Raspberry Pi Camera Robot (Product Page)
- official OpenCV tutorials
- OpenCV Tracking Algorithm Tutorial