**The Plymouth Owl robot**

Phil Culverhouse, William Stephenson,
Martin R Simpson and Clare Simpson.
Plymouth University

*NOTE: this module is experiment-based. As such you are expected to maintain a log book for the experiments you carry out and results you obtain. This will form part of the assessment.*

The Owl robot is a stereo camera host designed for the exploration of verging camera stereopsis. It has five degrees of freedom offering Neck rotate with Stereo eyes with pan and tilt. Local processing is by the Raspberry Pi dual camera compute board located at the base of the robot. An additional PCB provides an interface to the Pi for Servo control and an audio codec. The cameras are Pi HD cameras set to deliver stereo pairs at VGA resolution and streamed using RTP protocol web streaming at 30fps. See Fig. 1 for a photo of the robot, camera separation is 65mm.

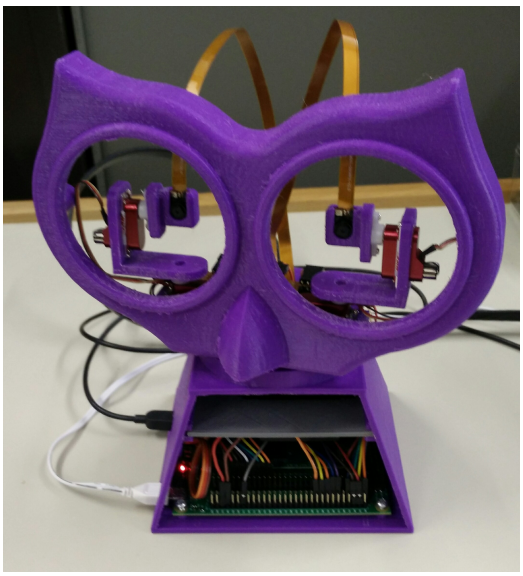


**Figure 1: The Plymouth Owl robot showing the stereo camera configuration (note: white cable is 5v power, black is USB host connection)**

A pair of MKS DS65k high-speed digital servos moves the eyes with a 333Hz period (see Table 1). Normal drive PWM is 3ms period, with a pulse width between 850us-2150µs (ie. 1300 PWM value range). The centre position is set at approximately 1,500µs and the servos have a dead band of one microsecond. The PWM range allows for 160° rotation, with one PWM step being 0.113°. NOTES:

1) This must be limited by inspection to the space available in the eye sockets. The neck servo is a standard PWM servo, a Corona DS558HV offering 160° rotation. Again, this must be limited to avoid attempting to drive past the end-stop.
2) It may vary between robots due to the seating of the plastic mounts on the servo drive splines

The processor is BCM2835 (the same installed in Raspberry Pi B+). The compute board was chosen as it offers dual DMA camera inputs, that facilitate the high-speed streaming of camera images to the internet. See Fig.2 for a photo of the computer board.

| MKS DS65K-V1 Gold Label Specifications | | |
|---|---|---|
| Body dimensions | 22 mm wide x 8.5 mm thick x 20.6 mm high | |
| Max dimensions | 28 mm wide x 8.5 mm thick x 24.3 mm high | |
| Stall torque (4.8V) | 1.85 kg.cm | 26 oz.in |
| Stall torque (6V) | 2.20 kg.cm | 31 oz.in |
| Weight | 6.5 g | 0.23 oz |
| Lead length | 10 cm | 4.0 in |
| Operating speed (4.8V) | 0.20 sec/60° | |
| Operating speed (6V) | 0.15 sec/60° | |
| Operating voltage | 3.8V to 6.0V | |
| Bearings | 2 ball bearings | |
| Gear material | 7075 Chrome-Titanium alloy | |
| Case material | CNC machined 6061 aluminium | |
| Motor type | Coreless motor | |
| Working frequency | 1520µs / 333hz | |
| Dead band | 1 µs | |
| Driver type | FET | |
| Digital | Yes | |
| Programmable | No | |
| Operating temperature | -10 to 60° C | |
| Supplied accessories | 1 plastic output arm, 1 arm retaining screw, DS6100 & DS65K RDS Adaptor, 4 mounting screws | |
| Optional accessories | Frame for MKS DS65K, MKS 1.5A SBEC, MKS 2A SBEC, MKS Plastic Servo Output Arms, MKS Heavy Duty Plastic Servo Output Arms, MKS Short Metal Servo Output Arms | |

**Table 1: MKS DS65k specification**

Note the PWM connections are at the rear of the compute board, and one of the two camera FFC connectors is shown at the front.

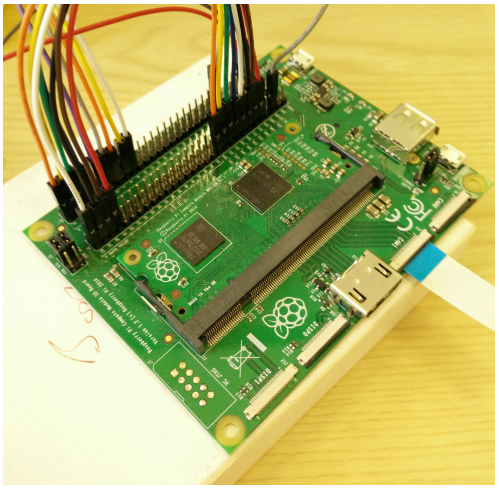


**Figure 2: The Raspberry Pi Compute board**

The Compute Board connects to a host via the USB, supporting RNDIS over USB (ie. Ethernet over USB). A simple web server has been configured for initial exploration of the robot using 10.0.0.10 IP address (host should be set to address 10.0.0.1 for example). A simple PWM demonstration programme is available at this

address. The V4L camera server can be accessed at 10.0.0.10:8080/stream/video.mjpeg/ (see linux-projects.org for details). See also Annex 8 for Pi stereoscopic camera capture issues on the OWL.

**Setting up the OWL Pi processor with a host**

Plug the OWL USB cable into your computer's USB socket. Always use the same socket when using Windows as the following setup is specific to a socket. I suggest on SMB302/303 computers that you use the Display USB which is on the left, top at the rear of the monitor. You then need to open the network settings on your machine to set the communications to the OWL as static address of 10.0.0.1. note that the OWL uses 10.0.0.10 (ie. same subnet if the subnet mask is 255:255:255:0)

See APPENDIX 9 for device settings for WINDOWS machines.

**Software – TCP Server on the Pi**

PFCsocket.py holds the Python code to open an IP socket using the TCP protocol to the hardwired host at address 10.0.0.10. It waits for packets that are 24 bytes long, but breaks the loop when a NULL packet is received. See Annex 2 for the Python source code. Code snippet 1 shows the TCP communications initialise sequence.

```
#set the socket comms up, use TCP as it is error correcting end to end.
soc = socket.socket()
host = '10.0.0.10' #ip of raspberry pi
port = 12345
soc.bind((host, port))
```

**Code Snippet 1: Python TCP init**

The packet is defined as a five-word string "Right-y Right-x Left-y Left-x Neck", where each is a four digit number that is expected to be in the range 1000-2000, which drive the servo PWM directly using PIGPIO (see abyz.co.uk/rpi/**pigpio**). For example: [1105 1240 1155 1400 1200]. Once started the server spawns a PIGPIO daemon to provide direct PWM control of the servos. The ports are initialized for range and period, see Code Snippet 2.

```
pi1 = pigpio.pi()
# set up servo ranges
pi1.set_PWM_range(16, 10000)
pi1.set_PWM_range(14, 10000)
pi1.set_PWM_range(17, 10000)
pi1.set_PWM_range(15, 10000)
pi1.set_PWM_range(13, 10000)

pi1.set_PWM_frequency(16,100)
pi1.set_PWM_frequency(14,100)
pi1.set_PWM_frequency(17,100)
pi1.set_PWM_frequency(15,100)
pi1.set_PWM_frequency(13,100)
```

**Code Snippet 2: PIGPIO servo PWM initialise code**

The server then waits for the host to send 24 byte packets. Upon receipt it replies ”OK” as an acknowledgement (Code snippet 3). It exits when a NULL length packet is received. This is typically when the host program closes the TCP port. Note that in Python indenting is important for WHILE and IF construction.

```
soc.listen(5)
comm, addr = soc.accept()
while True:
  packet = []
  packet=comm.recv(24) # max length of 5 ints between 1200-2000 each
  print >> sys.stderr, 'got string' # print string to console
  comm.send('ok')
```

**Code Snippet 3: start of packet loop**

The server also prevents the servos from being driven beyond the eye sockets limits or neck-rotate limits. See Code snippet 4 for the python code of range bounds limiting to avoid servo damage.

```
 #Get Data from Fields
 Rx = int(A[0])
 Ry = int(A[1])
 Lx = int(A[2])
 Ly = int(A[3])
 Neck = int(A[4])
 # range check to prevent servo overdrive
 if (Ry>2000):
   Ry = 2000
 if (Ry<1120):
   Ry = 1120
 if (Rx>1890):
   Rx = 1890
 if (Rx<1200):
   Rx = 1200
 if (Ly>2000):
   Ly = 2000
 if (Ly<1180):
   Ly = 1180
 if (Lx>1850):
   Lx = 1850
 if (Lx<1180):
   Lx = 1180
 if (Neck>1950):
   Neck=1950
 if (Neck<1100):
   Neck=1100
 # now set the servos to the programmed position
 pi1.set_PWM_dutycycle(16, Ry)
 pi1.set_PWM_dutycycle(14, Rx)
 pi1.set_PWM_dutycycle(17, Ly)
 pi1.set_PWM_dutycycle(15, Lx)
 pi1.set_PWM_dutycycle(13, Neck)
```

**Code Snippet 4: Remainder of packet decode and PWM loop**

A demonstration server interface program is available that tests the servos and allows the user to explore camera image capture and processing whilst positioning the servos. Note that the PWM value is written to the i/o register for that PWM wire (that goes to the servo pin).

*** NOTE: This is re-read by the PWM hardware on the Pi, 100 times a second (as the settings in Code Snippet 2), and the servo maintains its position until a different PWM value is written to the register by Pigpio.

The Pi server waits for packets on 10.0.0.10:12345 currently, the port address is defined in the server Python code, and also in the host C++ code. The C++ host calls have been created to initialise the link, then to send a packet and check the returned acknowledgement.

For example: u_sock = OwlCommsInit (int PORT, string PiADDR ); , where PORT will be declared as 12345, and PiADDR as 10.0.0.10.

And for the servo command, for example string OwlSendPacket (u_sock, CMD), where u_sock is returned from the init call above, and CMD is a character array holding something like this “1500 1500 1500 1500”, where the string defines Rx, Ry, Lx, Ly, Neck in sequence.

The Pi server range checks all PWM settings, as shown in Code Snippet 5, and limits the servo drive to be within these bounds. You must check that these range limits are good for your robot. The centre position needs to be both cameras upright and with no toe-in on the camera angle.

```
// OWL eye ranges
// SET UP MANUALLY FOR EACH ROBOT
int RyB = 1120; // (bottom) to
int RyT = 2000; //(top)
int RxR = 1890; //(right) to
int RxL = 1200; //(left)
int LyB = 2000; //(bottom) to
int LyT = 1180; //(top)
int LxR = 1850; // (right) to
int LxL = 1180; // (left)
int NeckR = 1100;
int NeckL = 1950;
int RxC=1545;
int RyC=1530;
int LxC=1545;
int LyC=1580;
int NeckC = 1530;
int Ry,Rx,Ly,Lx,Neck; // calculate values for position
int RyRange=RyT-RyB;
int RxRange=RxR-RxL;
int LyRange=LyT-LyB; // reflected so negative
int LxRange=LxR-LxL;
int NeckRange=NeckL-NeckR;
```

**Code Snippet 5: range limits on PWM servos in the host C++ control programme.**

### SSH terminal access to Raspberry Pi

It is easy to connect from a host to the Pi using a secure shell (SSH) such as putty (www.putty.org) that supports cryptographic network protocol to allow secure communications using an insecure network. Linux and MAC both use the 'ssh' terminal protocol to interact. In a terminal window type:

ssh 10.0.0.10<ret>

Using Putty, connect to 10.0.0.10, the Pi default **TCP/IP** (Transmission Control Protocol/Internet Protocol) at the default socket address (22). The first time this occurs the SSH protocol will prompt the user for confirmation that the quoted SSH key is acceptable. Answer YES to establish a permanent SSH table entry on the host for the OWL Pi computer. The connection will operate once the Pi has booted (about two minutes).

### Camera stream processing

The images from the Right and Left cameras are currently obtained directly from the webserver running on the Pi, at 10.0.0.10:8080/stream/video.mjpeg.
The server outputs both images at 640x480 resolution joined together into one 1280x480 image. The cameras are read at 30 frames per second (fps). The camera processing OpenCV code is required split the frame into Right and Left, then carry out any corrections and detections on the images. This additional processing WILL reduce the frame rate from 30fps to something lower.

```
const string source =  "http://10.0.0.10:8080/stream/video.mjpeg";
Mat Left;
Mat Right;

 VideoCapture cap (source);            // Open input
 if (!cap.isOpened())
 {
    cout  << "Could not open the input video: " << source << endl;
    return -1;
 }
 cv::Mat Frame;
 while (1){
    if (!cap.read(Frame))
    {
       cout  << "Could not open the input video: " << source << endl;
       break;
    }
    Mat Left (Frame, Rect(0, 0, 640, 480) ); // using a rectangle extract left camera frame
    Mat Right (Frame, Rect(640, 0, 640, 480) ); // and now the right frame
    imshow("Owl-L", Left); // display images
    imshow("Owl-R", Right);
    waitKey(20); // wait a little to allow the OS video processing stream to schedule the display on
the screen.
  }
```

**Code Snippet 6: the video stream open and ROI extraction code**

There is a known issue with linking to an IP video stream using the MJPEG codec on MacOS X, because the default video stream codec is MOV, not MJPEG format. This

means that the additional ffmpeg library needs to installed on the mac. However, there could a conflict in libraries that support both MOV and MJPEG codecs. Linux and WIN32/64 machines do not have this problem. Although the ffmpeg library is an externally specified code that must be included during the CMAKE stage of building OpenCV from source.

Code Snippet 6 shows how the video stream is opened and the frame is split into two regions of interest (ROIs).

Now we have captured a pair of images, we can control the servos to take pictures from anywhere within the view window afforded by the Owl robot design. This may involve moving eyes and neck servos. The servos are fast, and so it is possible to move the cameras at least ten times per second and still get blur-free images that we can process to extract target range information. See the owl-1 main.cpp in the QT project folder

You should spend time understanding how to control the servos to make coordinated eye movements that mimic animal vision systems. You will then need to process the video stream to develop a closed-loop control system to direct the eye gaze toward meaningful targets. We will use Proportional control initially. Check Owl-1 for an example.

**Proportional Control**

There is much written about control theory, check the Internet resources. Check out http://blog.opticontrols.com/archives/344 control for dummies as well as https://en.wikipedia.org/wiki/Control_theory, which give a balanced view. Figure 6 depicts three controller options from Proportional, Integral and Derivative control (PID). We will consider Proportional control of plant, a servo, to a set point – the servo position.

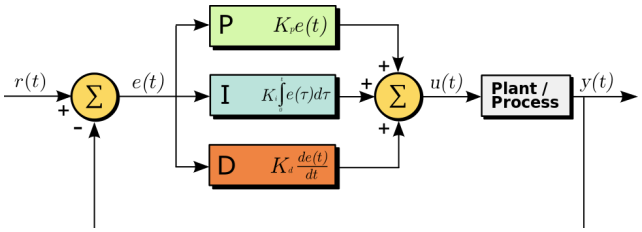



**Figure 6: PID control of a process (source: https://en.wikipedia.org/wiki/PID_controller)**

Proportional control measures the error in position *e(t)* at time *t* and applies a scale factor (*Kp*) to determine the change in servo PWM positional signal. The complication is that the positional error is measured from the camera image. A target is identified and its retinal (image) coordinates are determined through some target tracking technique. The offset from the current position to the desired is calculated and a conversion from camera coordinates to servo coordinates is made. The difference between the current servo position and the planned one is then applied to the servo PWM position value.

*Calculate the gain constant Kp, and show experimental evidence that your control loop is working. What settling time constant can you achieve?*

*What is the vestibula-ocular reflex? How might you implement this in the owl robot?*
An issue is mapping servo motion to changes in the camera image, so that a target can be tracked, or the eyes moved to centre on a new target using a mimetic saccade. Saccades are ballistic motions of mammalian eyes that allow top-down or bottom-up control https://en.wikipedia.org/wiki/Saccade. Eyes are also controlled by smooth-pursuit as they track moving objects https://en.wikipedia.org/wiki/Smooth_pursuit. Salience of items in the visual scene determines eye motion. If objects are moving faster than 30 deg/second a catch-up saccade is often required. (hint: check the servo maximal rate to see if this limits our owl robot in anyway).

*Calibrate servo-to-cameras to ensure that peripheral vision maps well into servo coordinates. Give a map of your calibration. What is the calibration procedure? You will need to think about this!*

**Owl-1 Demonstration program**
Owl-1 is the QT folder for the demonstration program. It comprises a) main.cpp, owl-comms.h (for TCP communications), owl-cv.h (for OpenCV correlation), owl-pwm.h (for definitions of PWM ranges).

NOTE: Program sources are kept in the Repository/Sources folder, Projects are created in Repository/Projects. This simplifies code reuse, as the sources folders can be common across projects. I recommend you adopt this philosophy.
Use the arrow keys to establish the maximal range boundaries in the X and Y axes for each camera (R and L cameras). Record these as RxM … LyM in the pwm.h definitions.

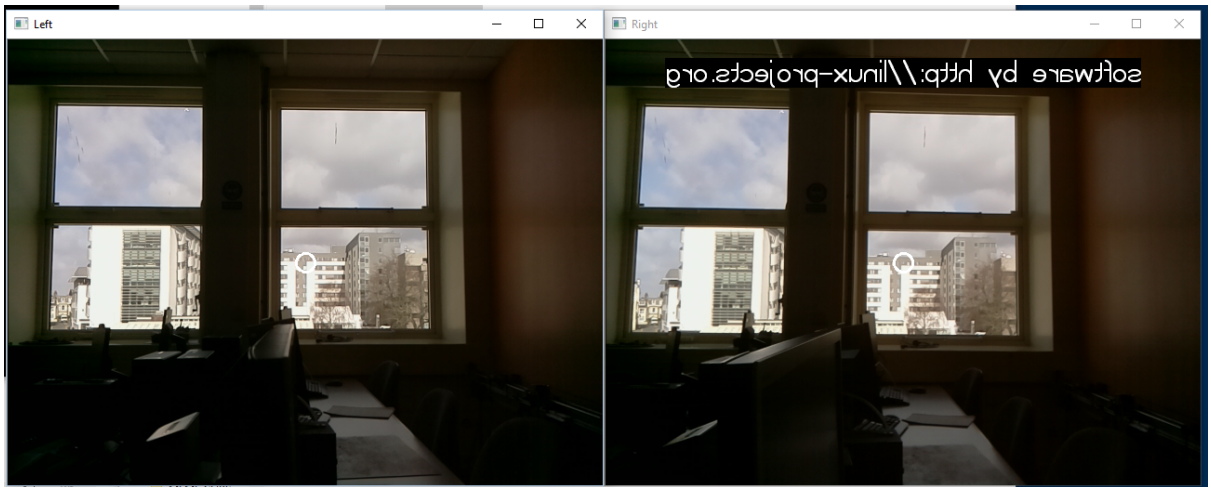



**Figure 7: left and right views of an Owl robot, aligned parallel at a distant target**

**Camera Alignment**
The OWL cameras must be checked for proper alignment. First set the OWL facing a distant target (Fig.7) and align the left and right cameras to be parallel. Look at the camera servos, they should be in approximately the same positions for both cameras (ie. X-axis parallel to eye socket, Y-axis horizontal). Then, facing the

chequerboard pattern, close enough to be able to see if the cameras at rotated. See

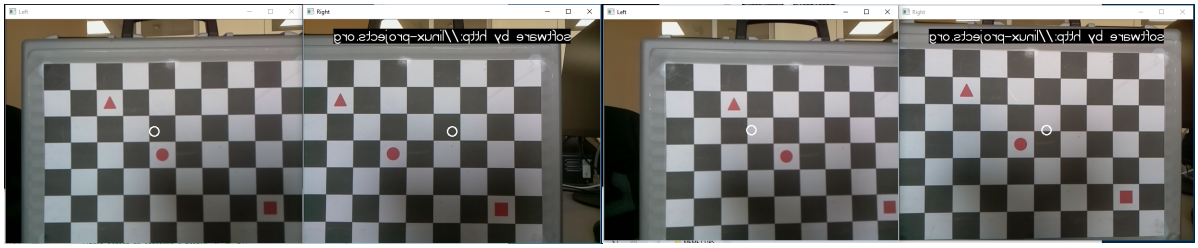


**Figure 8(a): left camera rotated, (b) left camera Y-axis shifted with right camera**

Fig.8a for rotation, Fig.8b for camera Y-axis mis-alignment. Fig.9 depicts what is an acceptybale level of camera rotation and mis-alignment in the Y-axis.

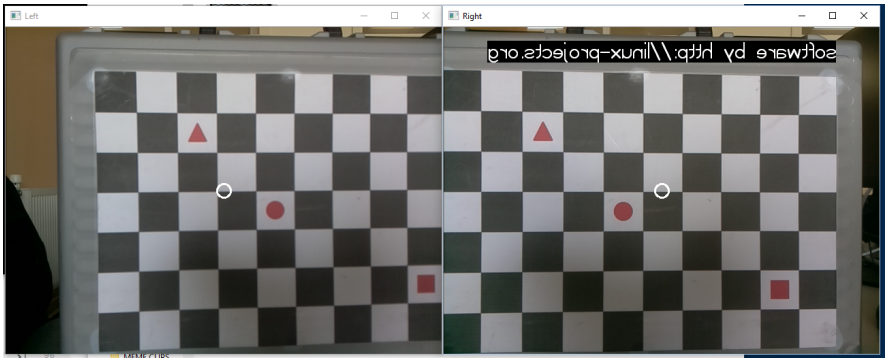


**Figure 9: cameras are aligned to an acceptable level**

You should not have to adjust the Y-axis much except to ensure that the eyes are horizontal. The X-axis toe-in (amount they face each other) should be small, such that the focal point of their gaze will be several metres away or more. In fact focal gaze point should at infinity for the stereo disparity calculations.

Your cameras are now in some form of scaled relationship between the servos and the field of view seen from the cameras. Using the arrow keys for gaze control again, move the Right camera to point at a suitable target for a template, press the ‘c’ key to capture a template from the centre FOV. At this point the template defined earlier is used to search the image from the Left camera. Using a Proportional control technique, the Left camera is moved such that the best correlation match is positioned at the central focus of attention in the left camera. See Fig.10 for a snapshot showing both cameras and the correlation result.

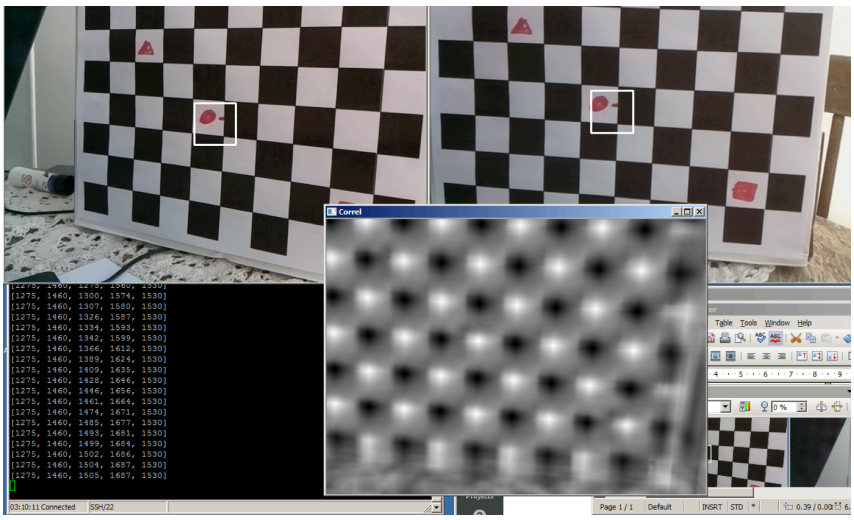



**Figure 10: Target correlation and servo control.**

The top of the figure shows Right and Left cameras, bottom left Pi server output**.** Centre bottom shows the correlation output for the current camera frames. Note that the correlation method can produce a match between the two camera frames even if there is some difference between the target shapes (due to perspective distortion). It is slower than image rectification, but it does allow the cameras to be moved and then targets correlated without re-calibration of the epi-polar geometry. If rubber-banding is added to allow affine distortions of the targets, then the processing rate drops.

**Vergence contro**l

Check, using break points set into the Owl-1 main.cpp code, the proportional control constants and scale factors. Make a note and change the gain constants and observe the system going out of control. Discover the settings that give you the fastest settling time to the target that is stable.

You will then need to set up the Right camera servo controls so it also tracks the target. I suggest you set up the target at know distance. Run in debug mode, and let OWL fixate on a target. Whilst it is still running, insert a breakpoint at the end of your distance calculation code block and then the debugger will stop at that location and show the variables. You can then compare theory of the geometry of that known target distance to what you calculate on the program. NOTES:

1) Model your equations in Excel to enable you to compare the output of the theory versus the robot.
2) You may need to change the pixels-to-servo-steps constant (currently estimated to be 0.113 degree per step). This is estimated by placing a target at the left edge of the camera field of view, record the servo position. Then move the servo to shift the target to the right edge of the camera field of view and measure the servo position. This value will be specific to a robot.
3) Ensure you use Radians when calculating sin(), cos(), not degrees
4) The inter-pupillary distance (IPD) was designed to be 65mm (see drawings), but can vary between robots.
5) Take measurements of actual distance -v- reported distance and construct graphs and discuss in your report to explore the errors in the system.

Consider a target for calibration of the servos/camera for vergence? Can you use the chequerboard pattern? What might be better?

READ http://docs.opencv.org/3.0-beta/doc/tutorials/calib3d/camera_calibration/camera_calibration.html and implement for your robot. download code from http://docs.opencv.org/3.0-beta/_downloads/camera_calibration.cpp or from the DLE in the CODE folder.

There is an issue of fixed focus lenses. You will need to understand the camera and its lens in a little more detail. The Pi CSI cameras are based upon the OV5647 camera chip from Omnivision. See http://www.arducam.com/spy-camera-raspberry-pi/. The sensor is five mega-pixel in a module with a fixed-focus lens. It therefore has a fixed field of view (FOV) of approximately 49° in the horizontal axis. See Annex 3a for details of the module design and Annex 3b for the camera sensor specification.

## VISION – EYES

Insects vary widely in the resolution and arrangement of eyes. The sensing components are called ommatidia that incorporate a lens and a single sensor, although some insects have lenses that are shared across a few sensors for colour sensing. These are arranged in clusters according to the size and specialization of the insect. For example, most ants have no more than 100 sensors in an eye. Wolf spiders have more than six eyes. Flies and wasps can have more than 1,000 ommatidia in an eye. The Mantis shrimp has colour vision and the eyes are steerable on stalks (see the video at http://www.nature.com/news/mantis-shrimp-s-super-colour-vision-debunked-1.14578). It has twelve different types of colour ommatidia in its eyes. They appear to be capable of differentiating spot colours across the spectrum. The shrimp uses these to discriminate prey (see http://www.biology.lu.se/research/research-groups/lund-vision-group for research in this animal vision systems). Jumping spiders (Salticidae) are renowned for a behavioral repertoire that can seem more vertebrate, or even mammalian, than spider-like in character (Harland et al., 2012). They possess a pair of front-facing primary high acuity eyes and a pair of outer secondary motion detecting eyes. See the video in http://www.danielzurek.com/spider-vision.html and Menda et al. 2014).

Most fish (Teleost) possess trichromatic colour vision. Elasmobranchs (sharks and rays) are monochromatic. Their eyes (Fig. 11) are structurally like primates being spherical in nature with a lens, a retina and blind spot where the optic nerve leaves the eye. Although the optics are slightly different, since fish live in water a denser medium. Surface dwelling fish have vision that is spectrally centred in the green region (approx. 500nm). Blue colours are lost in deeper water, and fish in this region have a peak sensitivity 475nm. Fish sight is essentially monocular with little stereo overlap.

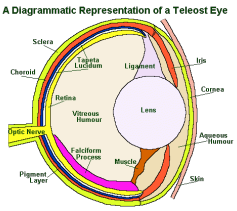



Figure 11: fish eye
(source:http://www.earthlife.net/fish/sight.html)

Reptiles can also perceive colour, but the ability is determined by evolutionary need. Chameleons can analyse image data from eyes that can move independently (see http://www.cogsci.nl/blog/bird-brains-and-fish-eyes/148-a-bit-about-the-evolution-of-eye-movements). Bird vision is divided into two types, forward facing eyes with excellent stereo vision, and sideways facing eyes with excellent all round vision (Fig. 12). As with fish the eyes are fixed in their sockets. Owls have large forward facing eyes, with 70º stereo overlap between the eyes. The neck can rotate 270º to provide all round vision.

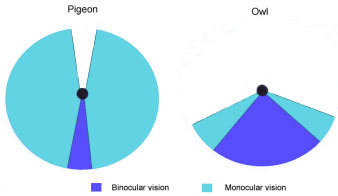



Figure 12: Bird vision field of view
(source: https://en.wikipedia.org/wiki/Bird_vision)

Humans, apes, and most, if not all, of the Old-World monkeys are trichromatic. Others are dichromat or monochromat. See Fig. 13 for a an example of how a scene will appear for these three types of vision.

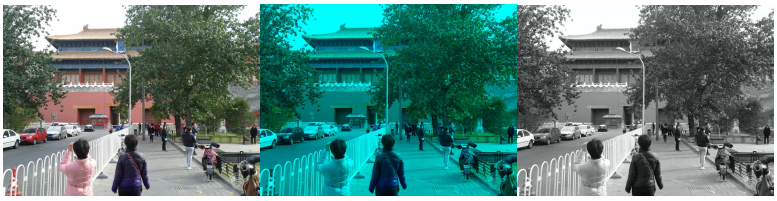


Figure13: A photo viewed by (a) a trichromat,
(b) a dichromat and (c) a monochromat

The human eye is more spherical than that in lower animals (Fig.14a). Rods are low light sensitive monochromatic sensors that dominate peripheral vision are the trichromatic sensors, which are densest in the foveal region of the retina (Fig 14b). There are approximately 126 million in each eye. The retina contains local band-pass filtering processing (amongst other processing structures)

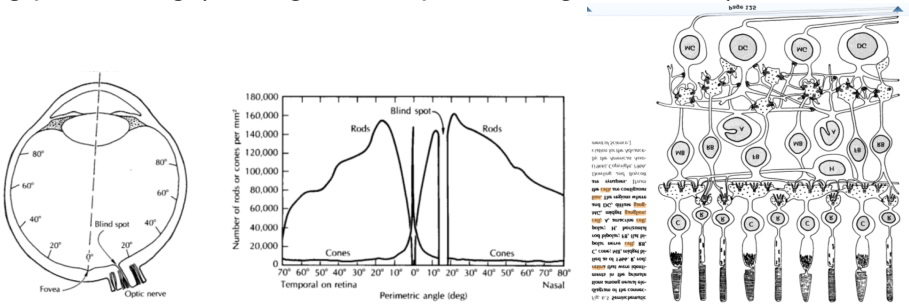



Figure 14: (a) the eyeball, (b) sensor packing density and
(c) retinal processing schematic cellular diagram (source: Cornsweet 1972).

Primates can move their eyes and only humans appear to use motion as a social feature for communication. There are four types of motion (i) saccades, (ii) smooth pursuit, (iii) vergence movements and (iv) vestibulo-ocular movements.
See https://www.ncbi.nlm.nih.gov/books/NBK10991/. Saccades are used for top-down (conscious) or bottom-up (unconscious). They are ballistic in nature and take about 100-200ms to set up, we can experience up to ten per second. Smooth pursuit is for tracking slow moving objects. Vergence is to control both eyes to steer inwards to a close target, such that the centre of the target has zero disparity between the two eyes. Vestibulo-ocular movements are used to compensate for head motion as the brain attempts to stabilise the eyes when they are fixated on a target, as the body is in motion. Many animals use saccades, including birds and insects, to scan a

part of the visual scene over their image sensors. Wasps will hover over a target, flying to and fro horizontally to scan it before deciding as to what to do. Primates will often saccade a target to establish target relevance prior to a decision.

## DEPTH PERCEPTION and STEREO VISION

Humans use several methods to gauge distance of objects. The obvious one is stereopsis. We have two eyes and in the region of shared field of view it is possible to calculate the disparity of points in the scene. Disparity is the amount of displacement on the retina a feature projects on each eye. The closer the distance, the more the disparity. In addition, focussing the eyes gives distance estimate. Humans possess steerable eyes, if the centre of the horizontal visual axis in each eye is moved to coincide with the same point on a feature, then the vergence angle (the toe-in) of the eyes can be used to estimate distance through triangulation. Finally, as we live in a three-dimensional moving world, perspective, shading and texture gradients, and analysis of feature and object motion can reveal the relative ordering of objects from close to far away. See https://en.wikipedia.org/wiki/Depth_perception for a more complete list.

Fig. 15 shows occlusion, perspective, relative size, and shading and texture gradients. These all contribute to the strong three-dimensional appearance of the scene. Robotic computational vision systems can analyse images for these properties. Let us review each in relation to the Plymouth Owl:

**Figure 15: Depth perception without stereopsis**

Stereopsis

There are several ways of implementing stereopsis, all require a mapping to be established between the cameras. The cheapest solution is to establish a common baseline between two calibrated cameras, and thus establishing a homography between points across both cameras. The cameras images are said to be rectified.

This allows epi-polar lines to be drawn that horizontally map (assuming cameras are separated horizontally) rows of pixels in each camera. This may involve affine transformations. See Annex 4a for details of camera calibration to correct for lens and other distortions. The 3 by 3 Essential matrix and the similar 3 by 3 Fundamental matrix give the mapping between cameras that are static and pointing at the same scene. Scenes can be rectified such that the row address in each camera have been mapped to the other. This makes it easy to calculate the disparity between similar features in each image. If the cameras are moved, then the calibration process must be completed again. See Annex 4b for details on epipolar geometry and image rectification.

See https://www.robots.ox.ac.uk/~vgg/hzbook/hzbook1/HZepipolar.pdf and http://www.cs.cmu.edu/~16385/lectures/Lecture18.pdf

Focusing the eyes

The Raspberry Pi CSI camera modules are fixed focus and so we have no ability to use sharpness of image as a criterion for estimating distance. But, in animals, muscles are adjusted to allow the lens to accommodate the target. Through training the distance to target can be estimated (see http://www.cambridgeincolour.com/tutorials/camera-autofocus.htm). Image sharpness can be estimated by performing Fourier analysis on the image content. A predominance of low frequency edges indicates blur. In fact, cameras often auto-focus using this technique, with hardware edge gradient processing being inside the camera (see http://www.cambridgeincolour.com/tutorials/camera-autofocus.htm). See Bajcsy 1998 p14 for a procedural description of focus-derived depth estimation. Also, review Banks et al. (2015) for details of animal eye iris and focusing.

Vergence angle

The geometry of a pair of cameras trained on a single target allows for triangulation to estimate distance. Fig. 16.

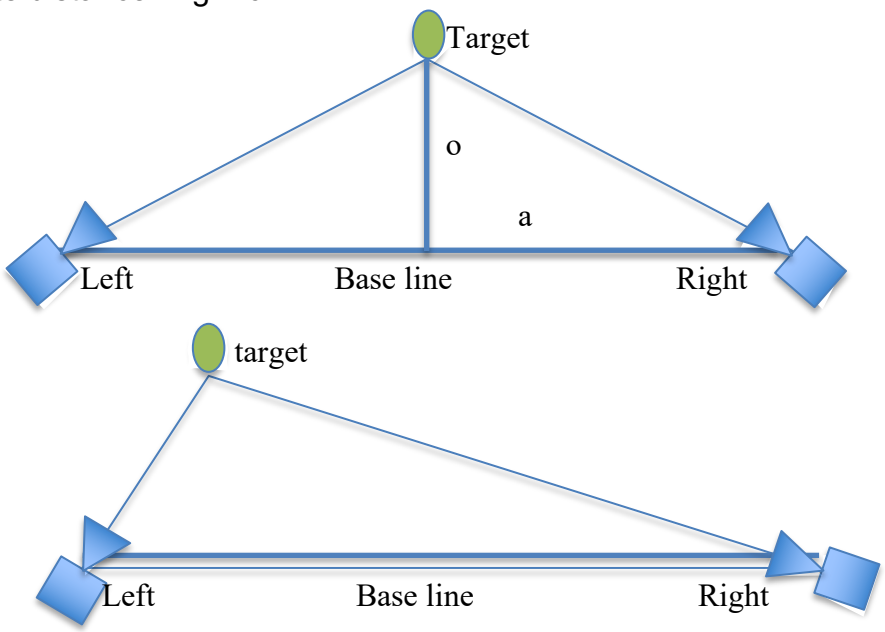



Figure 16: The geometry of vergence with (a) symmetric, (b) asymmetric cameras

If the cameras are symmetric (Fig.16a) then an equilateral triangle is assumed and the distance to the target is sin (theta) = o/a, where theta is in radians and a=half baseline. If asymmetric (Fig.16b) then applying the sin rule ( $\frac{\sin a}{A} = \frac{\sin b}{B} = \frac{\sin c}{C}$ ).
The OWL requires a mapping to be made from servo PWM drive to camera rotation angle. You will have to establish this. Ensure you record your measurements.
Vergence angle is a powerful tool for establishing $2^1/_2$D map (sketch) of the scene.
Review videos of animal eye/neck rotations to understand what is preferred in different animals – use this in your report to reference your choice of interaction.
See the Vision lecture notes for more detail.

Perspective

Two-dimensional projections of a three-dimensional world result in a perspective transform of scenes. This is shown in Fig.17 as the AMES room illusion. It emphasises that our vision system uses more cues than just stereo depth to understand a natural scene, although this is a very unnatural scene!

Figure 17: Ames Room (src: www.illusionworks.com)

The girl on the left is almost twice as far away from the observer as the girl on the right. However, when the room is viewed through the peephole, the stereo distances cannot be seen. Since you perceive the two people to be at the same distance from you, the one who has the larger visual angle appears larger. (From: http://www.cns.nyu.edu/~david/courses/perception/lecturenotes/depth/depth-size.html.

In real life, our conscious perceptions of the world, by default, give interpretations of the visual scene. See http://psych.hanover.edu/Krantz/SizeConstancy/index.html for an example. In this example, we compensate for distance in our perception of familiar things. This is called size constancy.

**Psycho-physics and neurological aspects.**

Primate eyes possess stereo saccadic eye movement that allows the central high acuity fovea of the eye to be brought to bear on a target of choice. This motion is ballistic (see https://en.wikipedia.org/wiki/Saccade and this video https://vimeo.com/22486450 as an example). The chameleon (see

https://vimeo.com/22486450) uses saccades but they are mostly unpaired movements in its eyes. Insects have fixed eyes and they scan a target in detail by moving in a horizontal manner in front of the target.

The fovea is a central very high acuity region of the retina that contains the highest number of colour sensitive retinal sensors in an eye, see Fig. 18 and Foundations of Vision by Wandell (1995). There are approximately 150,000 mm$^2$ colour sensors in the fovea, which subtends no more than 20° of visual angle.

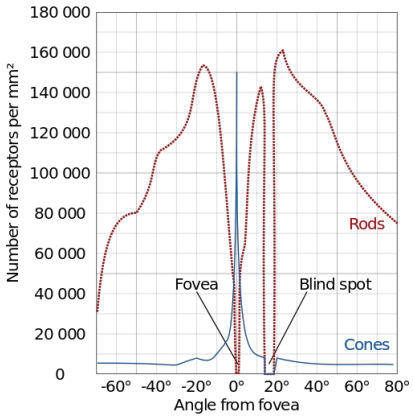



Figure 18. Primate fovea density of rods and cones (source: Wandell 1995)

When the fovea is moved to alight on a target it has been likened to a ‘spotlight of attention’ moving to a topic of interest in the visual world in a manner that supports the analysis of ‘what;’ and ‘where’ a salient target resides. The complication is that close targets will occupy much of the visual field, whereas distance targets will be small.

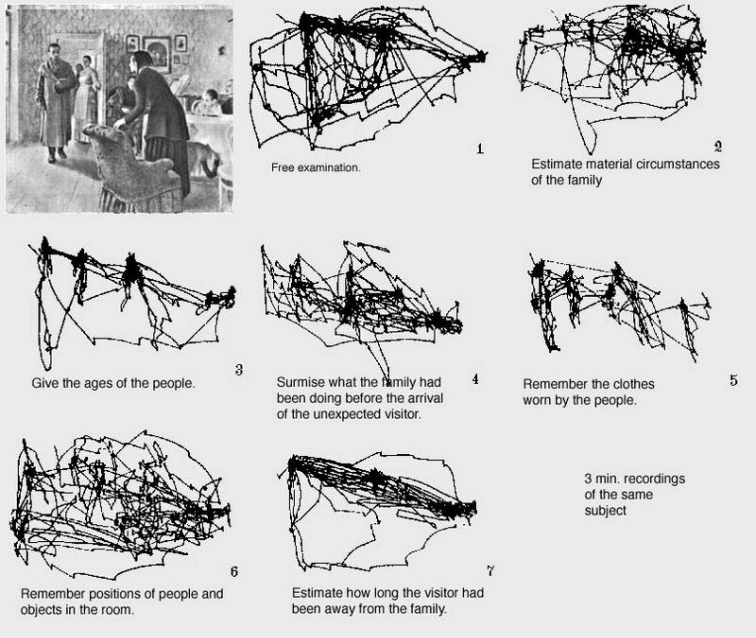



Figure 19. Saccadic eye motion changes according to the question being answered. (Source: ‘The visitor’ by Yarbus,1967 and also https://en.wikipedia.org/wiki/Eye_tracking).

Fig.19 shows an example of the many studies that have explored how our eyes are used to investigate the world. Note that Yarbus (1967) posed different questions that elicited different scans of the visual scene.

However, most studies have been on artificial scenes. Tatler et al (2011) argue that natural scenes offer a more complex situation, with both top-down and bottom-up search strategies being applied to control the saccadic motion across the scene. Tatler et al. suggest a more complex scenario. Fig. 20 shows a model of visual attention that is driven from spatial and colour contrast analysis of a scene.

Saccades are high speed movements and can reach 900°/s in humans. This is faster than many small servo motors are capable of mimicking, and thus can potentially limit robot active vision systems. The owl robot uses MKS DS65K servos that can attain 300°/s at 4.5v supply voltage, and 450°/s at 6v. At best this is only half the velocity the human eye can achieve.

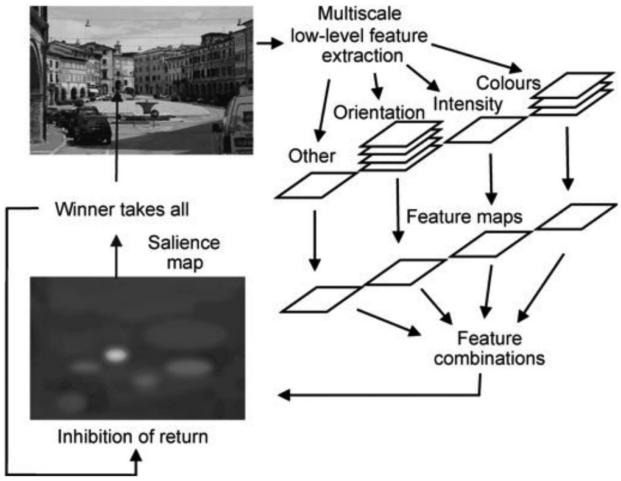



**Figure 1.**
Schematic of Itti's salience model. Figure from Land and Tatler (2009); redrawn from Itti and Koch (2000).

Figure 20: Itti's salience model of visual attention (from Tatler et al, 2011 Fig.1).

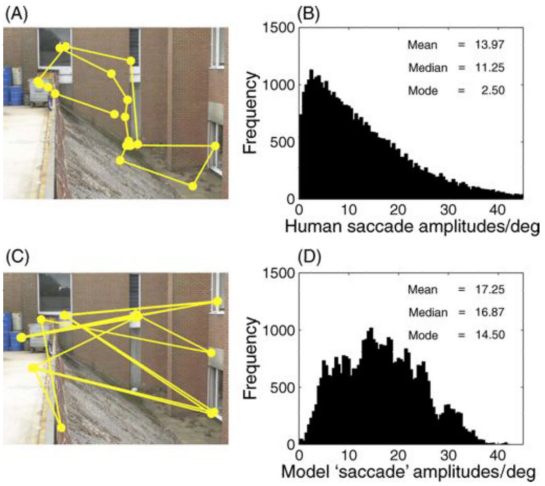



Figure 21: Comparisons of model and human saccades (from Tatler et al, 2011 Fig.2).

This model is compared to human performance on visual scenes. Some discrepancies arise (see Fig. 21). Read Feature Integration Theory by Triesman and Gallard (1980) in the slide overview. Fig.22 depicts the full model of attention, notice how complex it is, and how much processing must be done to decide what to attention to in the visual world.

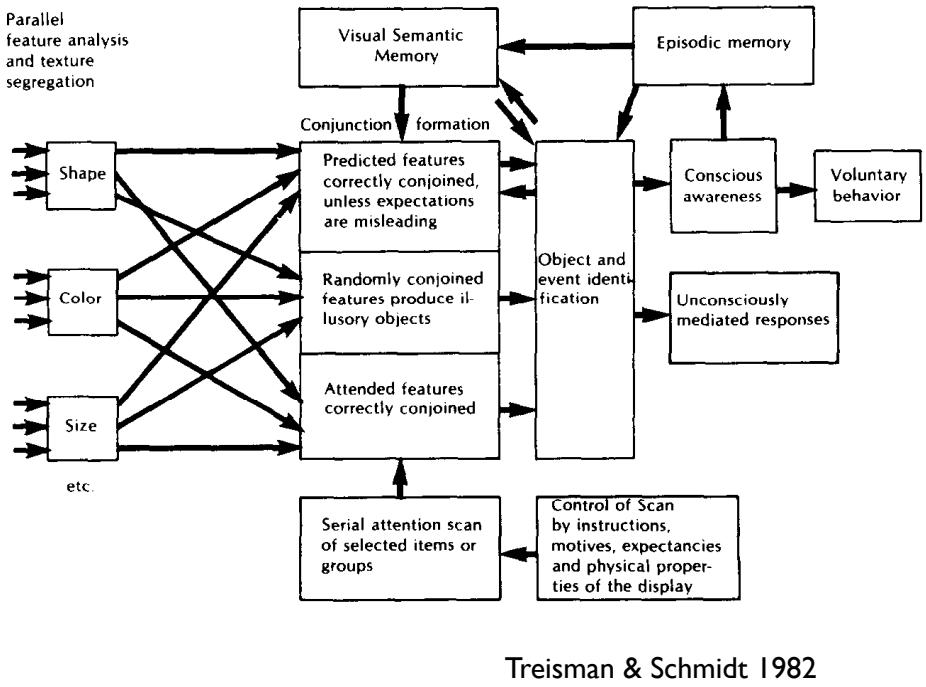



Figure 22: Treisman & Schmidt (1982) model of Feature Integration (from FIT slides, Treisman 1986).

Can you improve on this model? Using a low-pass filter to find the locations of large blobs in the scene, perhaps modified by colour saturation/hue and edge intensity. Saccade from strongest down. Make an experiment to compare to how a person watches a scene.

You could experiment with PyGaze (Dalmaijer et al. 2013) to compare your computer model with human gaze tracking using your web camera. See http://www.pygaze.org/2015/06/webcam-eye-tracker.

Use the Difference of Gaussians (DoG) filter to create bandpass-filtered images showing moderate sized blobs in the scene. Use this to direct saccades over time. Review the model described in Fig. 22 by Treisman & Schmidt (1982). See code snippet 7 for an example of how to calculate a band-pass filtered image using a DoG filter. Note that DoG filtering occurs in the ganglion cell outputs from the retina and is fed to the visual cortex for processing (see Marr 1982).

```
Mat DoG = img.clone();
Mat g1, g2;
int k=5; int g=9;
GaussianBlur(DoG, g1, Size(g,g), 0);
GaussianBlur(DoG, g2, Size(g*k,g*k), 0);
// following the ratio present in some ganglion cells in the fovea.
Mat result = (g1 - g2)*2;
cv::imshow("DoG", result);
cv::waitKey();
```

Code Snippet 7: DoG filter example in openCV

select your DoG filter spatial size manually, or create an image pyramid of sub-sampled filtered products using *pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 );*

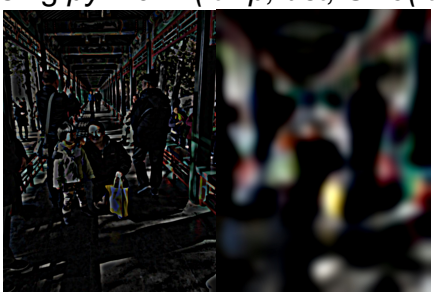


Figure 23: DoG filtered images
(a) kernel = 39x39, (b) kernel = 127x127

Explore the peaks in the output (see Fig.23 as an example), apply Itti's model and control your robot to saccade between the peaks (maxima) in the image. Look at Fig. 9, and compare where your eye's alight to your model's. Increase the DoG filter kernel width, or reduce the image size to see a low-resolution map of features that holds few blob peaks to saccade between.

## Itti & Koch's Salience Model using the OWL

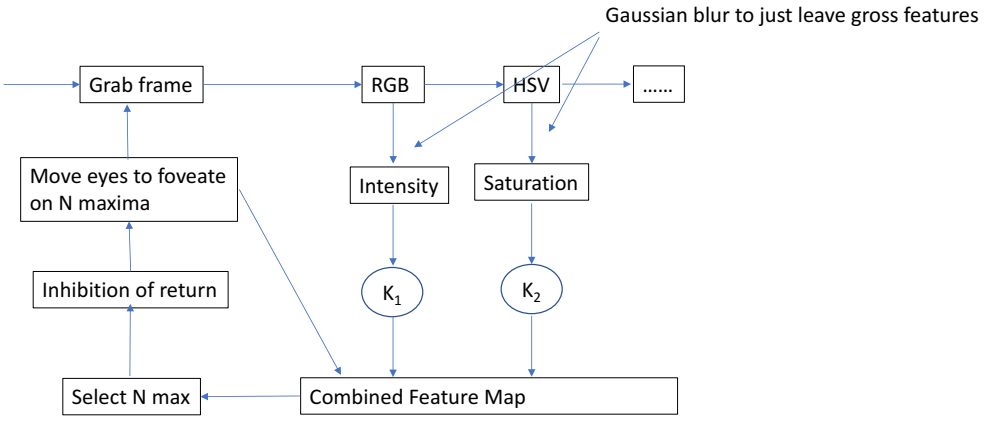



**Figure 24: Implementing Itti & Koch's salience model**
Legend: grab a frame, convert the RGB to gray scale by K1 as the intensity map. Convert RGB to HSV and extract Salience, look at cv::split() to extract the S matrix, as the salience map. Scale and add to combined feature map. Select N (N=1) strongest responding points in the map, and move eyes to foveate on that location. Take next camera frame, and repeat.

Fig. 24 shows a possible sequence of processing of each camera frame. Notice that the raw gray-levels in the intensity map are too detailed to be useful, so blur or differentiate and blur (using the DoG filter function described above).

See Figure 25 for an example screen shot of a student solution of Itti & Koch's model, that used three channels of information (Gaussian blurred gray intensity, Gaussian blurred colour Saturation, and DoG differentiated and blurred contrast).

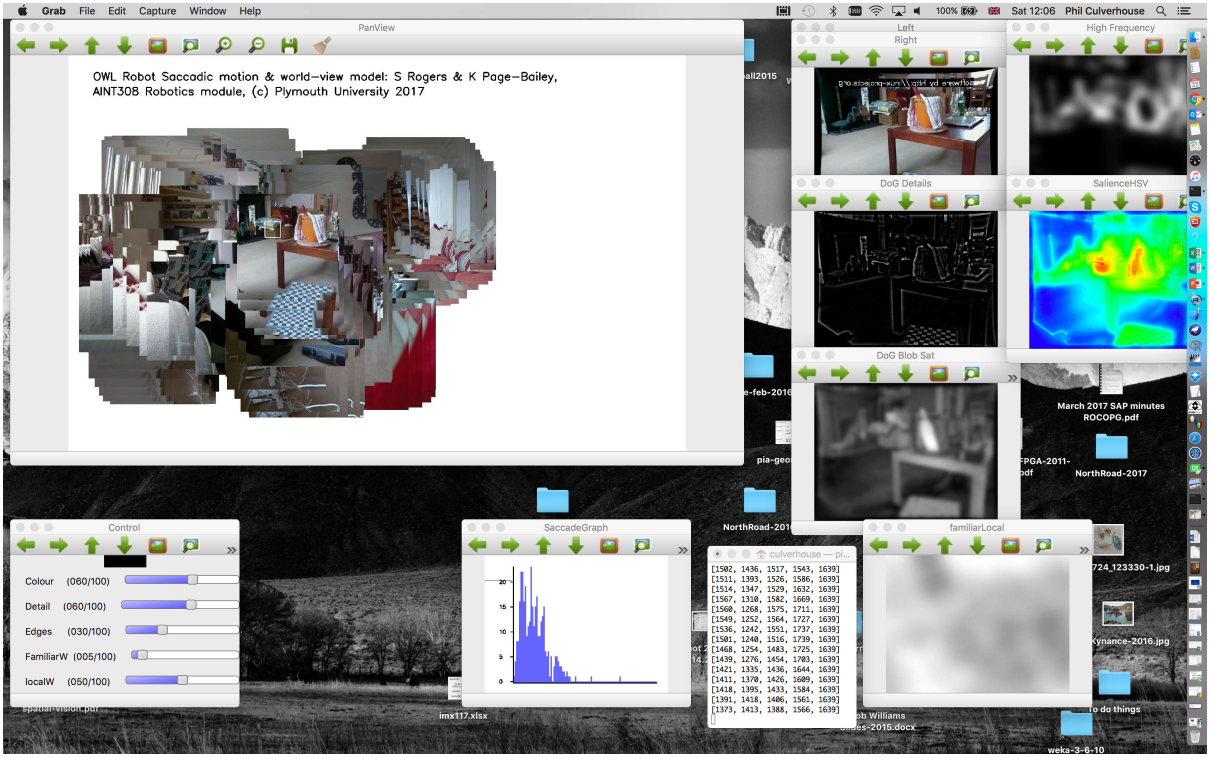



**Figure 25: Itti & Koch's salience model example**.
Legend: Top right hand side (top to bottom): input camera images (Left, Right), High Frequency feature map, Difference of Gaussian differentiated map, summed feature map, and HSV colour saturation map. Bottom (right to left): inhibition of return map (called familiarity map), list of servo packets, saccade histogram and feature-map coefficient control.

<u>Stereo vision and the preying Mantis</u>

From Rossel (1996), Praying mantids use binocular cues to judge whether their prey is in striking distance (see Fig. 26). When there are several moving targets within their binocular visual field, mantids need to solve the correspondence problem. They must select between the possible pairings of retinal images in the two eyes so that they can strike at a single real target. In this study, mantids were presented with two targets in various configurations, and the resulting fixating saccades that pre- cede the strike were analyzed. The distributions of saccades show that mantids consistently prefer one out of several possible matches. Selection is in part guided by the position and the spatiotemporal features of the target image in each eye. Selection also depends upon the binocular disparity of the images, suggesting that insects can perform local binocular computations. The pairing rules ensure that mantids tend to aim at real targets and not at ''ghost'' targets arising from false matches.

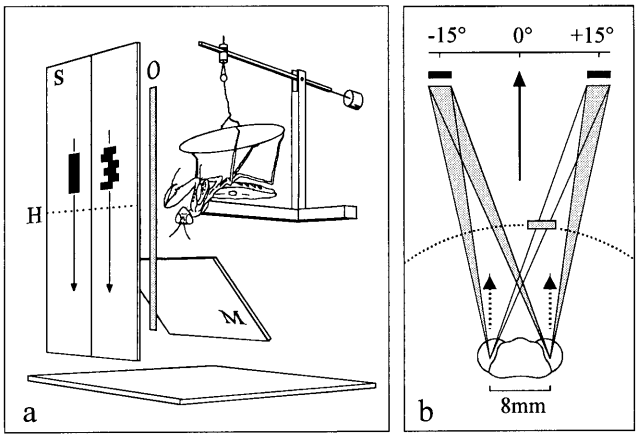



FIG. 1. (*a*) Experimental set-up with a praying mantis fixed up-side down to a holder. S, display screens with a simple and a complex target moving downwards; H, horizon of the mantid; M, mirror to monitor head saccades from below (the video camera is behind the insect); O, occluder to obsure targets for one or the other eye. (*b*) Illustration of the mantid head with projection of paired targets with a center-to-center separation of 30°. An occluder is adjusted to obsure the right target in the left eye view. Solid arrow, horizontal head angle relative to a position midway between targets; broken arrows, viewing directions of foveas; broken line, inner border of the catching zone (20 mm from the head; outer border is at 60 mm).

Fig.26: Mantis experiment setup (from Fig.1 Rossel, 1996)

## Feature and object motion

Targets that exhibit relative motion with respect to the camera system can be tracked and distance estimated given certain priors. See Bouguet (2001) and http://docs.opencv.org/trunk/d7/d8b/tutorial_py_lucas_kanade.html. Also see Annex 7 for a copy of the LKdemo.cpp from OpenCV examples/cpp. Note that this is a monocular tracker. It is possible to run this for both eyes, and combine the results in some manner to establish disparity relationships. It will have unknown robustness, depending on your matching technique, the velocity of the relative motion and the processing frame rate of your computer.

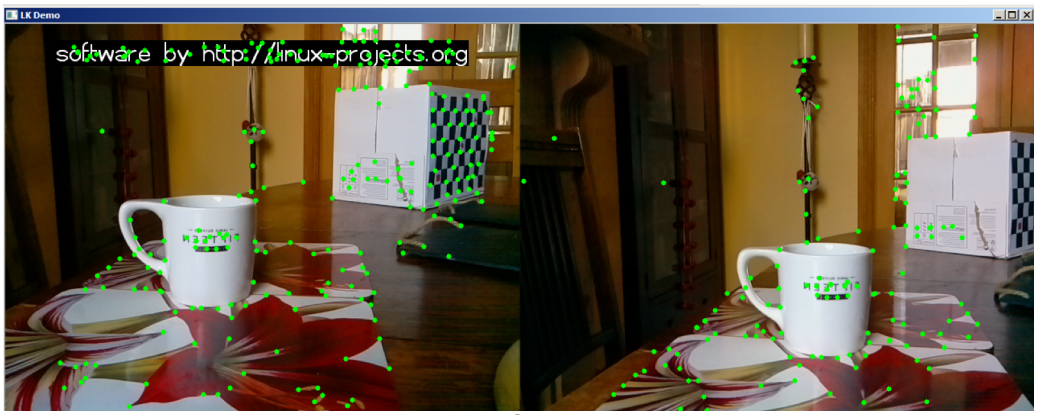



Figure 27: LKDemo example using OWL cameras

Fig. 27. Using standard area-scan cameras at 30fps (the Raspberry Pi CSI cameras in the OWL) only slow moving object velocities can be estimated, due to the time

taken to process images and the camera sample interval. (establish this theory for Masters only).

Shading and texture gradients

Primates get clues to the three-dimensional shape of an object through texture and shading illumination gradients (see Horn, 1989). Fig.28. See Annex 5 for an example code for texture gradient estimation. … and how to use (Masters only).


Figure 28: Texture flow example

**YOUR FINAL REPORT**

The assessment is in two stages, a first report and demonstration of Owl servo control, then a final report and demonstration of vision behaviour, include saccadic eye motion and depth estimation, with target tracking. This last task is hard, and you are reminded to keep notes on your experimental results, screen snapshots, tables of calibration of the cameras etc. as these will contribute to your mark.

**References**

Bajcsy R (1988) Active Perception. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-24. (accessed from http://ieeexplore.ieee.org/document/5968/?arnumber=5968 Nov. 2016).

Banks MS, Sprague WW, Schmoll J, Parnell JAQ and Love GD (2015) Why do animal eyes have pupils of different shapes? Science Advances Vol. 1(7), e1500391. DOI: 10.1126/sciadv.1500391.

Bouguet, J. Y. (2001). Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm. *Intel Corporation*, *5*(1-10), 4.

Cornsweet T (1972) *Visual Perception*. Academic Press. ISBN 9780323148214.

Duchowski A (2013) *Eye Tracking Methodology: Theory and Practice* Springer. IBSN=1447137507.

Dalmaijer ES, Mathôt S and Van der Stigchel S (2013) PyGaze: An open-source, cross-platform toolbox for minimal-effort programming of eyetracking experiments. Behav Res DOI 10.3758/s13428-013-0422-2.

Harland, D., Li, D., and Jackson, R.R. (2012). How jumping spiders see the world. In How Animals See the World: Comparative Behavior, Biology, and Evolution of Vision, O.F. Lazareva, T. Shimizu, and E.A. Wasserman, eds. (New York: Oxford University Press).

Menda G, Shamble PS, Nitzany EI, Golden JR and Hoy RR (2014) Visual Perception in the Brain of a Jumping Spider. Current Biology 24, 2580–2585, DOI: 10.1016/j.cub.2014.09.029 .

Rossel S (1996) Binocular vision in insects: How mantids solve the correspondence problem **(**target selection *Sphodromantis viridis*). Proc. Natl. Acad. Sci. USA. Neurobiology. Vol. 93, pp. 13229–13232.

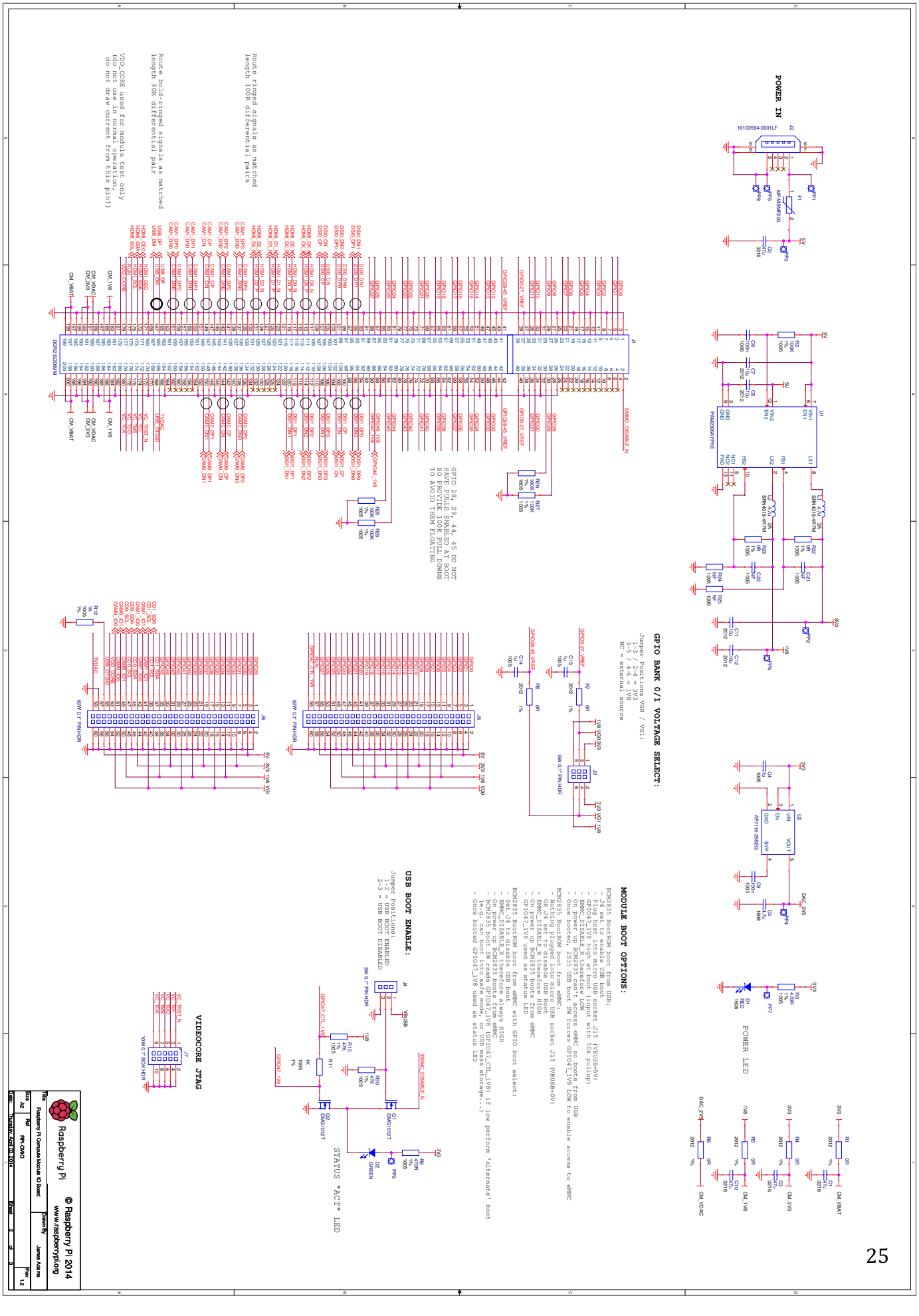
Treisman & Gelade, 1980; See *http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Lectures/FIT.pdf for a digestible version. Also see https://en.wikipedia.org/wiki/Visual_search#Feature_integration_theory_.28FIT.29*
Treisman A & Schmidt H (1982) "Illusory conjunctions in the perception of objects." *Cognitive Psychology*, Vol. 14, pp. 107–141.

Tatler, Hayhoe, Land & Ballard (2011) Eye guidance in natural vision: Reinterpreting salience. J Vis. 2011 ; 11(5): . doi:10.1167/11.5.5.
**(see** https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3134223).

Wandell BA. (1995) *Foundations of Vision*. Sinauer Associates, Inc., Sunderland, MA**.** DOI 10.1002/col.5080210213.

Yarbus, A. L. (1967), Eye Movements and Vision, New York: Plenum. (Originally published in Russian 1962).

## Annex 1. Schematic diagram of servo controller UPDATE



## ANNEX 2. PYTHON SOURCE Pi TCP SERVER

```
#!/usr/bin/python
###########
# PFC OCt 2016 basic Owl controller using IP sockets at address 10.0.0.10:12345
# socket code from https://docs.python.org/2/howto/sockets.html
# uses pigpiod.if from http://abyz.co.uk/rpi/pigpio/ to control servos on the Pi
# (c) Phil Culverhouse, CRNS, Plymouth University
###################
import socket
import sys # for stderr printing

#set the socket comms up, use TCP as it is error correcting end to end.
soc = socket.socket()
host = '10.0.0.10' #ip of raspberry pi
port = 12345
soc.bind((host, port))

# now set up the PWM server using pigpiod.if (python)
import pigpio
import time

pi1 = pigpio.pi()
# set up servo ranges
pi1.set_PWM_range(16, 10000)
pi1.set_PWM_range(14, 10000)
pi1.set_PWM_range(17, 10000)
pi1.set_PWM_range(15, 10000)
pi1.set_PWM_range(13, 10000)

pi1.set_PWM_frequency(16,100)
pi1.set_PWM_frequency(14,100)
pi1.set_PWM_frequency(17,100)
pi1.set_PWM_frequency(15,100)
pi1.set_PWM_frequency(13,100)

##############################
# now run the server loop
##############################
soc.listen(5)
comm, addr = soc.accept()
while True:
  packet = []
  packet=comm.recv(24) # max length of 5 ints between 1200-2000 each
  print >> sys.stderr, 'got string'
  comm.send('ok')
  if len(packet) < 24:
    print >> sys.stderr, 'received NULL packet, quitting'
    break
  #packet sent is Ry Rx Ly Lx N
  A = map(int, packet.split(' '))
  #A=[int(item) for item in packet.split() if item.isdigit()]
  print >> sys.stderr, A

  #Get Data from Fields
  Rx = int(A[0])
  Ry = int(A[1])
  Lx= int(A[2])
  Ly= int(A[3])
  Neck = int(A[4])
  # range check to prevent servo overdrive
  if (Ry>2000):
    Ry = 2000
  if (Ry<1120):
    Ry = 1120
  if (Rx>1890):
```

```
    Rx = 1890
  if (Rx<1200):
    Rx = 1200
  if (Ly>2000):
    Ly = 2000
  if (Ly<1180):
    Ly = 1180
  if (Lx>1850):
    Lx = 1850
  if (Lx<1180):
    Lx = 1180
  if (Neck>1950):
    Neck=1950
  if (Neck<1100):
    Neck=1100
  # now set the servos to the programmed position
  pi1.set_PWM_dutycycle(16, Ry)
  pi1.set_PWM_dutycycle(14, Rx)
  pi1.set_PWM_dutycycle(17, Ly)
  pi1.set_PWM_dutycycle(15, Lx)
  pi1.set_PWM_dutycycle(13, Neck)

# on exit from the loop (send a “” packet)
comm.close()
pi1.stop()
```

## Annex 3a: Camera and lens module information

The camera is an OV5647 from Omnivision, this has been assembled into a module by a far-eastern assembly plant, possibly www.KaiLapTech.com. It has a fixed-focus lens. The following is available on the Raspberry Pi discussion groups, and shows some educated reverse engineering on the camera module.

This section has been taken from http://www.truetex.com/raspberrypi

***Raspberry Pi camera module stock lens characteristics:***
*My analysis demonstrates a 3.6mm focal length of the stock lens in the camera module, with an f/2.9 aperture, based on my physical measurements and the Omnivision OV5647 specs as follows:*
*Shooting a full-resolution still (2592 x 1944 pixels, 4:3 aspect ratio) displays a field of view about 194.5 inches wide at 195 inches distance. By optical trigonometry principles, this corresponds to a 2*atan(0.5*194.5/195) = 53 degrees horizontal field of view. Likewise the vertical field of view is 40 degrees, and the diagonal 66 degrees.*
*The active pixels of the sensor chip are 2592 x 1.4um = 3.629mm wide. Proportioning this by 195/194.5 = 3.6mm yields the lens focal length.*
*My microscopic examination of the lens entrance pupil with a 10x reticle shows it to be about 1.25mm diameter, so the f/number of this 3.6mm lens is 3.6mm/1.25mm = f/2.9.*
*For the 1080p video frame, the system uses the unscaled central 1920 x 1080 pixel crop. That is, the 1080p image is cropped without scaling from the central 1920 x 1080 pixels from the larger overall sensor frame. This is unlike a handheld camcorder, or video on a DSLR camera like the Canon 5D Mark II, where the 1080p image is scaled from the entire width of the overall sensor having a higher native resolution. This cropping versus scaling in the camera sensor is analogous to letterboxing instead of scaling on a display. The 1080p field of view with the 4mm lens is thus proportionately different from that of the full frame. The physical width of the central 1920 pixels spans 2.688mm on the sensor. This 1080p 16:9 frame cropping factor is 2952/1920 = 1.35x horizontally and 1944/1080 = 1.8x vertically from the 2952 x 1944 4:3 full frame. HD 1080p angular field of view is thus 39 degrees horizontally, 22 degrees vertically, and 45 degrees diagonally.*
*Cropping the 1080p image from the central area of the sensor, realizes less total image resolution, versus what is available if the imaged were scaled from the entire width. This is a consequence of the stock lens delivering less resolution than the sensor pixels. See below for a detailed analysis of the lens performance.*
*The dimensions of the sensor are 2592 x 1944 pixels, each 1.4 microns square, which overall provides a sensor area of about 10 square mm. The sensor diagonal is sqrt(2592**2 + 1944**2) = 3240 pixels, which spans 4.54mm. In video camera terms, where a so-called "1 inch" sensor actually is defined as a 16mm diagonal, the Raspberry Pi sensor is 4.54mm/16mm = 0.28 of a video camera "inch", which is to say a 1/3.5" format (slightly smaller than 1/3", slightly larger than 1/4", in terms of CCTV formats). (For historical reasons, video camera "inches" are 16mm. This strange characterization arose from the original video imaging technology, where a vacuum tube glass envelope of actual 1-inch outside diameter would encapsulate a smaller 16mm active device inside, and the size of the tube and not the sensor was the defining characteristic. In the era of the 1980s when semiconductor chips were first replacing video tubes, the chips came to be commercially identified with the size of the tubes they replaced, not the actual chip dimensions. The tube's imaging elements also were round instead of rectangular, so the diagonal dimension of a rectangular chip sensor was taken as the characteristic dimension for comparison.*

Lens calculations as from website are shown in Figure 1. Although an educated guess, these data suggest that the lens will keep in focus from approximately 28cm to infinity.

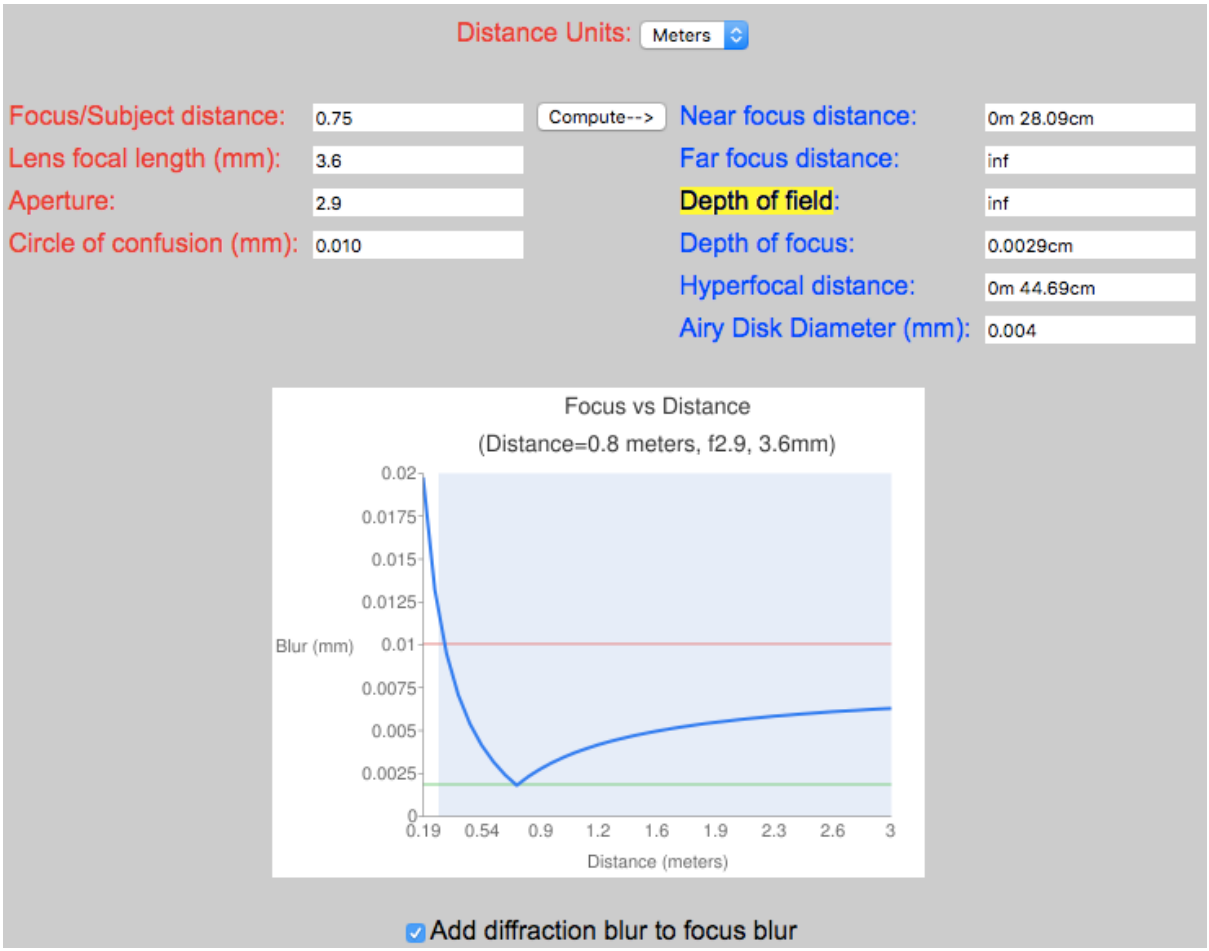
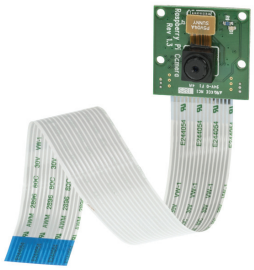



Figure 1: Pi CSI camera module lens calc. (http://www.tawbaware.com/maxlyons/calc.htm)

See also Forum comment from jbeale on focal length and angle of views i.e. f=3.6 and Angle of view 54x41 degrees:
https://www.raspberrypi.org/forums/viewtopic.php?f=43&t=150344 and
http://elinux.org/Rpi_Camera_Module#Technical_Parameters_.28v.1_board.29

## Annex 3b: OV5647 module overview

Raspberry Pi



# CAMERA MODULE

| | |
|---|---|
| **Product Name** | **Raspberry Pi Camera Module** |
| **Product Description** | High definition camera module compatible with the Raspberry Pi model A and model B. Provides high sensitivity, low crosstalk and low noise image capture in an ultra small and lightweight design. The camera module connects to the Raspberry Pi board via the CSI connector designed specifically for interfacing to cameras. The CSI bus is capable of extremely high data rates, and it exclusively carries pixel data to the BCM2835 processor. |
| **RS Part Number** | **775-7731** |
| **Specifications** | |
| **Image Sensor** | Omnivision 5647 CMOS image sensor in a fixed-focus module with integral IR filter |
| **Resolution** | 5-megapixel |
| **Still picture resolution** | 2592 x 1944 |
| **Max image transfer rate** | 1080p: 30fps (encode and decode)<br>720p: 60fps |
| **Connection to Raspberry Pi** | 15 Pin ribbon cable, to the dedicated 15-pin MIPI Camera Serial Interface (CSI-2) |
| **Image control functions** | Automatic exposure control<br>Automatic white balance<br>Automatic band filter<br>Automatic 50/60 Hz luminance detection<br>Automatic black level calibration |
| **Temp range** | Operating: -30° to 70°<br>Stable image: 0° to 50° |
| **Lens size** | 1/4" |
| **Dimensions** | 20 x 25 x 10mm |
| **Weight** | 3g |

**Accessories**

▲ Raspberry Pi Model B - **756-8308**

▲ Camera case **784-6193**

▲ 8GB SD card pre-programmed with NOOBS - **779-6770**

▲ Expansion board **772-2974**


▲ WiFi dongle **760-3621**

▲ 10400mAh Li-Ion battery pack **775-7517**


▲ Raspberry Pi user guide **768-6686**

RS

www.rs-components.com/raspberrypi

## Annex 4a: Camera lens distortion calibration

This is straightforward as OpenCV provides two camera calibration routines. See http://docs.opencv.org/3.1.0/d4/d94/tutorial_camera_calibration.html. See also http://wiki.ros.org/camera_calibration.

Annex 4b: Stereo camera epipolar geometry

(note figures are from R Hartley and A Zisserman (2004) Multiple View Geometry in Computer Vision, Cambridge University Press).

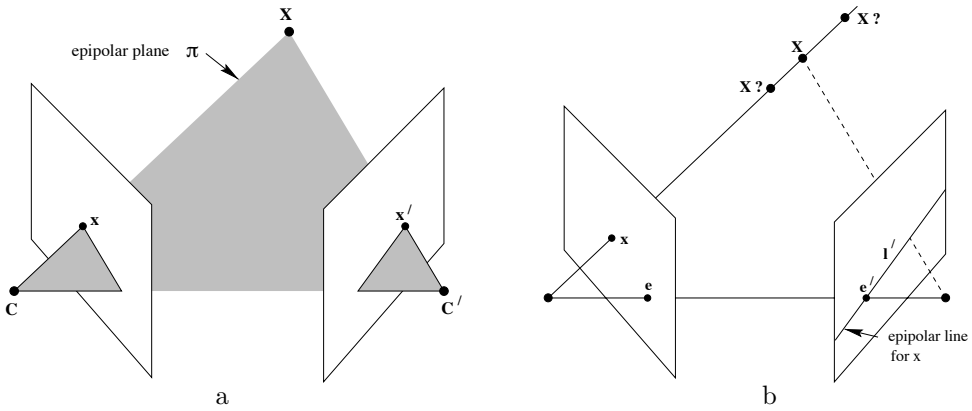



Fig. 8.1. **Point correspondence geometry.** *(a) The two cameras are indicated by their centres* $\mathbf{C}$ *and* $\mathbf{C}'$ *and image planes. The camera centres, 3-space point* $\mathbf{X}$*, and its images* $\mathbf{x}$ *and* $\mathbf{x}'$ *lie in a common plane* $\boldsymbol{\pi}$*. (b) An image point* $\mathbf{x}$ *back-projects to a ray in 3-space defined by the first camera centre,* $\mathbf{C}$*, and* $\mathbf{x}$*. This ray is imaged as a line* $\mathbf{l}'$ *in the second view. The 3-space point* $\mathbf{X}$ *which projects to* $\mathbf{x}$ *must lie on this ray, so the image of* $\mathbf{X}$ *in the second view must lie on* $\mathbf{l}'$*.*

A point in one camera image maps to the other camera such that the point can lie on the epipolar line *l'* see Hartley and Zisserman (2004) Fig.8.1. If the two cameras have been calibrated for lens distortions then the mapping is defined by the Essential matrix, which is a 3x3 matrix. It holds the mapping between the two cameras. If the position of a point is known in one camera image, it allows a line to be constructed in the other camera image that the point will lie on. A search along this line will reveal the offset and hence the disparity of that point between both cameras. See Fig.8.3 below.

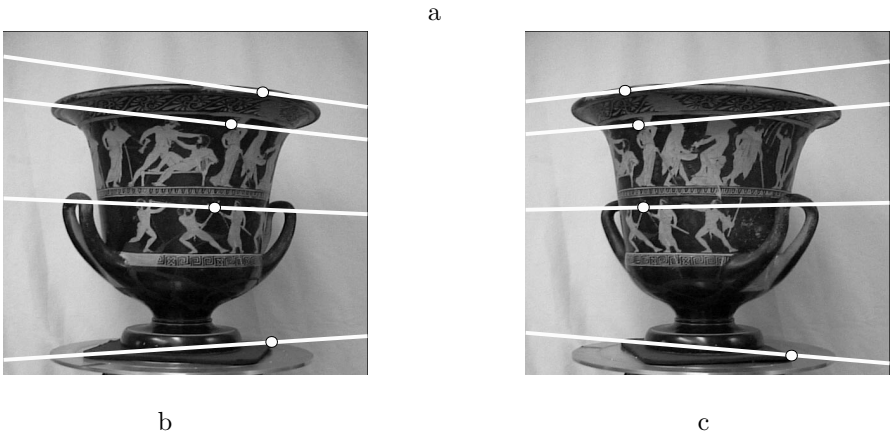



Fig. 8.3. **Converging cameras.** *(a) Epipolar geometry for converging cameras. (b) and (c) A pair of images with superimposed corresponding points and their epipolar lines (in white). The motion between the views is a translation and rotation. In each image, the direction of the other camera may be inferred from the intersection of the pencil of epipolar lines. In this case, both epipoles lie outside of the visible image.*

If the images are rectified, such that the epipolar lines are parallel, then the rows of pixels in one image will correspond to rows of pixels in the other image. In other words, a point at location row=10, col=10 in one image will be found in row 10 of the other, but in a different column. See Fig.3b1. notice how the scene is truncated in the rectified images.

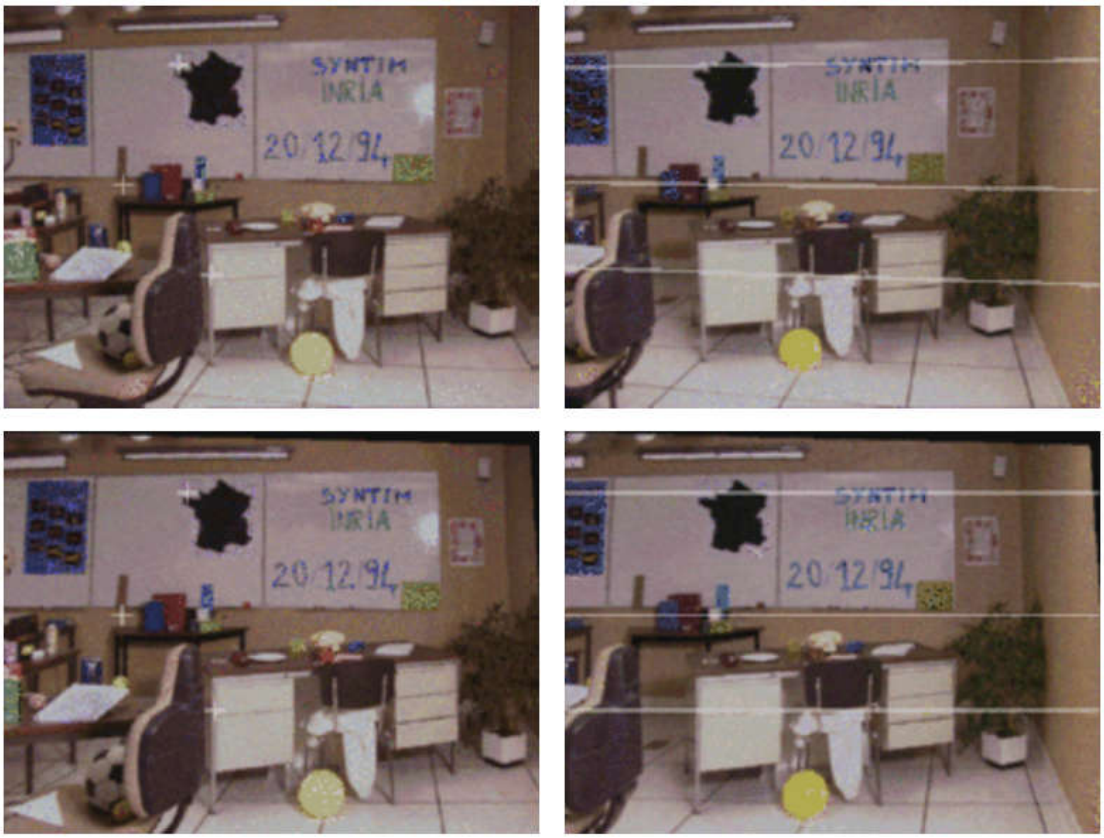



**Figure 3b1:** Top row: a stereo pair (Copyright SYNTIM-INRIA). Bottom row: the rectified pair. The right pictures plot the epipolar lines corresponding to the point marked in the left pictures (source http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO/node4.html#SECTION00022000000000000000

Epipolar lines and rectification save processing time, as the only other option is to search the image for the location of the corresponding point. This can be slow. The trade-off is that it may not be possible to establish a rectified image if a camera changes its view point compared to the other after calibration. If you have two agile cameras as the Owl robot, then we have little choice but to perform a global search for the best correlation of a feature or target between the two images. A serious issue is that the target may be very close to the cameras and so each camera will see quite a different image from the other. Matching in these circumstances may be difficult, or impossible.

OpenCV provides a set of calibration routines for epipolar geometry. See http://docs.opencv.org/3.1.0/da/de9/tutorial_py_epipolar_geometry.html.

SEE http://stackoverflow.com/questions/12794876/how-to-verify-the-correctness-of-calibration-of-a-webcam
Also read http://stackoverflow.com/questions/12794876/how-to-verify-the-correctness-of-calibration-of-a-webcam

The Essential matrix gives the translation and rotation of camera B in relation to camera A in camera coordinates. The Fundamental matrix includes camera intrinsic parameters, and maps between a pixel in one image to an epipolar line in the other. To establish a full correspondence a minimum of seven points across the scene have to be mapped. The OpenCV implementation requires eight, but it is recognised by some that more points may give a better solution. To this end it is normal to use a texture gradient feature extractor such as SIFT, or SURF, to select patches from the scene to establish correspondences. If we have, say, 300 possible features to be mapped, then it is normal to use a search technique called FLANN (Fast Approximate Nearest Neighbour), to find the best matches between the feature patches. It is needed since noise in the image (occlusion, shadows and so on) add ambiguity to the process. See http://docs.opencv.org/2.4/doc/tutorials/features2d/feature_flann_matcher/feature_flann_matcher.html and

Image rectification simplifies the subsequent image matching process, but it assumes the cameras maintain a fixed pose. See https://en.wikipedia.org/wiki/Image_rectification. And use opencv 3.x Samples/stereo_calib.cpp

Then rotated camera images are now projected on the same plane as each other and the cameras can be used to generate calibrated, rectified stereo pairs to extract disparities from. Depth maps can formed also. See opencv 3.x Samples/stereo_matcher.cpp.

Note both programs are on the DLE.

## Annex 5: Texture gradients from OpenCV3.1 Samples folder
Adapted from Python

```
#include <opencv2/opencv.hpp>
#include <vector>

using namespace std;
using namespace cv;

// FROM http://stackoverflow.com/questions/14234384/translate-numpys-array-reshape-to-opencv-equivalent/14381250
// run with NingNing-b.jpg
// PFC Dec 2016

int main (int argc, char** argv)
{
    cv::Mat img = cv::imread(argv[1]);
    cv::Mat dst, dstA;
    pyrDown( img, dst, Size( img.cols/2, img.rows/2 ));

    cv::Mat gray;
    cv::cvtColor(dst, gray, CV_BGR2GRAY);
    // to preserve the original image
    cv::Mat flow = gray.clone();
    int width = img.cols;
    int height = img.rows;
    int graySize = width * height;
    // "brighten" the flow image
    // C++ version of:
    // vis[:] = (192 + np.uint32(vis)) / 2
    for (unsigned int i=0; i<(graySize*3); ++i) {
        img.data[i] = (uchar)((192 + (int)img.data[i]) / 2);
    }
    Mat eigen = cv::Mat(gray.size(), CV_32FC(6) );
    cv::cornerEigenValsAndVecs(gray, eigen, 15, 3);
    // this is the equivalent to all the numpy's reshaping etc. to
    // generate the flow arrays
    // simply use channel 4 and 5 as the actual flow array in C++
    std::vector<cv::Mat> channels;
    cv::split(eigen, channels);

    int d = 12;
    cv::Scalar col(0, 0, 0);
    // C++ version of:
    // points =  np.dstack( np.mgrid[d/2:w:d, d/2:h:d] ).reshape(-1, 2)
    // including the actual line drawing part
    for (unsigned int y=(d/2); y<flow.rows; y+=d)
    {
        for (unsigned int x=(d/2); x<flow.cols; x+=d)
        {
            if (x < flow.cols && y < flow.rows)
            {
                cv::Point p(x, y);
                float dx = channels[4].at<float>(p) * (d/2);
                float dy = channels[5].at<float>(p) * (d/2);
                cv::Point p0(p.x - dx, p.y - dy);
                cv::Point p1(p.x + dx, p.y + dy);
                cv::line(flow, p0, p1, col, 1);
            }
        }
    }
    cv::imshow("FLOW", flow);
    cv::waitKey();
    return 0;
}
```

## Annex 6: DFT from OpenCV samples

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

#include <iostream>

using namespace cv;
using namespace std;

static void help(char* progName)
{
cout << endl
<<  "This program demonstrated the use of the discrete Fourier transform (DFT)." << endl
<<  "The dft of an image is taken and it's power spectrum is displayed." << endl
<<  "Usage:" << endl << progName << " [image_name -- default lena.jpg] " << endl <<
endl;
}

int main(int argc, char ** argv)
{
    help(argv[0]);
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";
    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT,
Scalar::all(0));

    Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
    Mat complexI;
    merge(planes, 2, complexI);// Add to the expanded another plane with zeros
    dft(complexI, complexI);// this way the result may fit in the source matrix

    // compute the magnitude and switch to logarithmic scale
    // => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
    split(complexI, planes); // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
    magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
    Mat magI = planes[0];
    magI += Scalar::all(1);                    // switch to logarithmic scale
    log(magI, magI);
    // crop the spectrum, if it has an odd number of rows or columns
    magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));
    // rearrange the quadrants of Fourier image,so that the origin is at the image
center
    int cx = magI.cols/2; int cy = magI.rows/2;
    Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
    Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
    Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
    Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right
    Mat tmp;           // swap quadrants (Top-Left with Bottom-Right)
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);
    q1.copyTo(tmp);    // swap quadrant (Top-Right with Bottom-Left)
    q2.copyTo(q1);
    tmp.copyTo(q2);
    normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values
into a
    // viewable image form (float between values 0 and 1).
    imshow("Input Image"       , I   );    // Show the result
    imshow("spectrum magnitude", magI);
    waitKey();
    return 0;
}
```

## ANNEX 7: LKdemo.cpp for motion tracking

This implements the Lucas-Kanade tracking algorithm utilising the 'Good Features to track' corner detector. Source: OpenCV 2.4.9/Examples/CPP/LKdemo.cpp

```
#include "opencv2/video/tracking.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

#include <iostream>
#include <ctype.h>

using namespace cv;
using namespace std;

static void help()
{
    // print a welcome message, and the OpenCV version
    cout << "\nThis is a demo of Lukas-Kanade optical flow lkdemo(),\n"
            "Using OpenCV version " << CV_VERSION << endl;
    cout << "\nIt uses camera by default, but you can provide a path to video as an
argument.\n";
    cout << "\nHot keys: \n"
            "\tESC - quit the program\n"
            "\tr - auto-initialize tracking\n"
            "\tc - delete all the points\n"
            "\tn - switch the \"night\" mode on/off\n"
            "To add/remove a feature point click it\n" << endl;
}
Point2f point;
bool addRemovePt = false;
static void onMouse( int event, int x, int y, int /*flags*/, void* /*param*/ )
{
    if( event == CV_EVENT_LBUTTONDOWN )
    {
        point = Point2f((float)x, (float)y);
        addRemovePt = true;
    }
}

int main( int argc, char** argv )
{
    help();
    VideoCapture cap;
    TermCriteria termcrit(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.03);
    Size subPixWinSize(10,10), winSize(31,31);
    const int MAX_COUNT = 500;
    bool needToInit = false;
    bool nightMode = false;

    if( argc == 1 || (argc == 2 && strlen(argv[1]) == 1 && isdigit(argv[1][0])))
        cap.open(argc == 2 ? argv[1][0] - '0' : 0);
    else if( argc == 2 )
        cap.open(argv[1]);
    if( !cap.isOpened() ) {
        cout << "Could not initialize capturing...\n";
        return 0;
    }
    namedWindow( "LK Demo", 1 );
    setMouseCallback( "LK Demo", onMouse, 0 );

    Mat gray, prevGray, image;
    vector<Point2f> points[2];
    for(;;) {
        Mat frame;
        cap >> frame;
        if( frame.empty() )
            break;
        frame.copyTo(image);
```

```
        cvtColor(image, gray, COLOR_BGR2GRAY);
        if( nightMode )
            image = Scalar::all(0);
        if( needToInit ) {
            // automatic initialization
            goodFeaturesToTrack(gray, points[1], MAX_COUNT, 0.01, 10, Mat(), 3, 0,
0.04);
            cornerSubPix(gray, points[1], subPixWinSize, Size(-1,-1), termcrit);
            addRemovePt = false;
        }
        else if( !points[0].empty() ) {
            vector<uchar> status;
            vector<float> err;
            if(prevGray.empty())
                gray.copyTo(prevGray);
            calcOpticalFlowPyrLK(prevGray, gray, points[0], points[1], status, err,
winSize,
                                 3, termcrit, 0, 0.001);
            size_t i, k;
            for( i = k = 0; i < points[1].size(); i++ ){
                if( addRemovePt ) {
                    if( norm(point - points[1][i]) <= 5 ) {
                        addRemovePt = false;
                        continue;
                    }
                }
                if( !status[i] )
                    continue;

                points[1][k++] = points[1][i];
                circle( image, points[1][i], 3, Scalar(0,255,0), -1, 8);
            }
            points[1].resize(k);
        }
        if( addRemovePt && points[1].size() < (size_t)MAX_COUNT ){
            vector<Point2f> tmp;
            tmp.push_back(point);
            cornerSubPix( gray, tmp, winSize, cvSize(-1,-1), termcrit);
            points[1].push_back(tmp[0]);
            addRemovePt = false;
        }

        needToInit = false;
        imshow("LK Demo", image);

        char c = (char)waitKey(10);
        if( c == 27 )
            break;
        switch( c ) {
        case 'r':
            needToInit = true;
            break;
        case 'c':
            points[0].clear();
            points[1].clear();
            break;
        case 'n':
            nightMode = !nightMode;
            break;
        }
        std::swap(points[1], points[0]);
        cv::swap(prevGray, gray);
    }
    return 0;
}
```

## Annex 8: Stereo camera support PI Compute board

See https://www.raspberrypi.org/forums/viewtopic.php?f=43&t=85012

15/03/2018 Stereoscopic camera capture - now implemented. - Raspberry Pi Forums

# Stereoscopic camera capture - now implemented. Quote

Hi All.

Wed Aug 20, 2014 2:58 pm

So the initial stereoscopic camera code has just been submitted, and Dom is intending to do a release probably tonight. This has been slightly rushed through to try and get it into the wild, so please report back if you hit issues, **BUT** there are restrictions - break the rules at your peril!
There is one new MMAL parameter that is valid on all 3 of the camera output ports - *MMAL_PARAMETER_STEREOSCOPIC_MODE*. It takes a *MMAL_PARAMETER_STEREOSCOPIC_MODE_T*

Code:

```
typedef enum MMAL_STEREOSCOPIC_MODE_T {
   MMAL_STEREOSCOPIC_MODE_NONE = 0,
   MMAL_STEREOSCOPIC_MODE_SIDE_BY_SIDE = 1,
   MMAL_STEREOSCOPIC_MODE_TOP_BOTTOM = 2,
   MMAL_STEREOSCOPIC_MODE_MAX = 0x7FFFFFFF,
} MMAL_STEREOSCOPIC_MODE_T;
typedef struct MMAL_PARAMETER_STEREOSCOPIC_MODE_T
{
   MMAL_PARAMETER_HEADER_T hdr;
   MMAL_STEREOSCOPIC_MODE_T mode;
   MMAL_BOOL_T decimate;
   MMAL_BOOL_T swap_eyes;
} MMAL_PARAMETER_STEREOSCOPIC_MODE_T;
```

(Please see IL config *OMX_IndexParamBrcmStereoscopicMode* and *struct OMX_CONFIG_BRCMSTEREOSCOPICMODETYPE* if you insist on doing IL. Doh, it should be struct OMX_PARAM_BRCMSTEREOSCOPICMODETYPE to be consistent. Can't be bothered to change it!).

*mode* sets up how the stereoscopic image is to be packed - either with the two images side by side, or one above the other (top/bottom). *decimate* is also sometimes referred to as half/half mode. The output frame ends up still being the same size as the original (eg 1920x1080), but the individual images are squashed (2:1) in one dimension to fit them both in (ie each eye would become either 960x1080, or 1920x540, but with non-square pixels).

*swap_eyes* allows putting the image for the right eye on the left/top, and left eye on the right/bottom. Display and H264 generally want swap_eyes=FALSE, JPEG stereoscopic wants swap_eyes=TRUE. NOT IMPLEMENTED YET!

The rules:

This is for the compute module only and requires two camera modules (obvious I know!). It is not possible to run it on a standard Pi as only one camera interface is exposed. Please don't ask. You **MUST** enable stereoscopic mode before enabling the camera component or setting the camera number. Stereoscopic mode needs to take control of both cameras and allocate memory differently (read more!), and those actions occur early on. There is a basic check to try and avoid you enabling stereoscopic too late. You can disable it again after the component has been enabled without issue other than having used more memory/resources than necessary.

You will need to instruct TVService to switch your TV into a stereoscopic mode before running your camera command if you want to view the stereoscopic output. For me that is something like



Code:

```
tvservice --explicit="CEA_3D_TB 4"
tvservice --explicit="CEA_3D_SBS 16"
```

*tvservice -m CEA* will give you a list of the modes your TV supports. The camera number parameter now selects the first channel camera. The second channel is always the other one (seeing as there are only 2 interfaces on this chip) The MMAL port format defines the overall image buffer for both images. I haven't written the userland raspistill/vid code cleanly as yet, but for a full side by side 5MPix JPEG you will want to execute something like *raspistill -w 5184 -h 1944 -3d sbs -o stereo.jps*. There are definite restrictions to what can be displayed both on putting things on the display, and what the display can then drive. Exact limits aren't established yet (and you may be able to exceed them with overclocking), but 1080P half/half will certainly work as it is almost no extra over normal 1080P. 720P full frame also works. The hardware H264 block has an internal line buffer that will limit you at a maximum of 1920 wide. Please do not exceed that. There is another limitation on the frame height at 1344, so I'm afraid that also means that 720P top/bottom full frame is also out. The width of each image must also be a multiple of 128 (overall width a multiple of 256), so 1920x1080 doesn't work (1920/2 = 960. /128 = 7.5) . raspivid command will be something like *raspivid -w 1792 -h 1080 -3d tb -hh -o stereo.264*. This is NOT MVC encoding, just H264 with the two images squashed into one. My understanding from Dom is that XBMC (and omxplayer) will pick up stereoscopic encoding from the filename if it is correctly formatted for it It looks like file.3DSBS.mp4, file1.HSBS.mp4, file2.3DTAB.mp4, and file3.HTAB.mp4 are the four options, but please check XBMC docs/forums for more info - I am no expert on the playback side. If you ask for an I420 or similar encoding, then you

should get a buffer the size of the MMAL port format with the two stereoscopic images packed in the desired format. All the camera settings (AGC, AWB etc) are slaved off the primary sensor. Covering the lens on one sensor will either have no effect (if the primary) or result in the other eye over saturating (if the secondary). This is expected behaviour. This will NOT be supported through V4L2 - the V4L2 API currently doesn't appear to have any calls or parameters relating to stereoscopic video, and I'm not going to be adding them

This code is a little rushed and is quite likely to have a few little gremlins in it. If you have a compute module with two cameras and are able to play about with this, then please do and report any issues (as long as you have obeyed the rules above) on this thread. I still have a little over a week to fix things up, but it would really help if you can give command-lines and/or test programs that demonstrate the issues (I know I haven't done the raspistill/vid mods yet!)

Have fun. **edit** Correction that 1920 wide will not work with H264 encode as it is not a multiple of 256.

## Appendix 9: Setting up the Pi server on a PC or Linux Box

The Pi compute board communicates to its host via USB. Networking is supported through the RNDIS TCP/IP over USB link. It is defaulted to the network address 10.0.0.10 at present (see Figs 3 and 4). An Apple MAC will connect automatically, but a PC and linux both require the driver set to the address 10.0.0.1.

**WIN7: See Fig A9-3 to A9-5 for the PC under WIN7.**

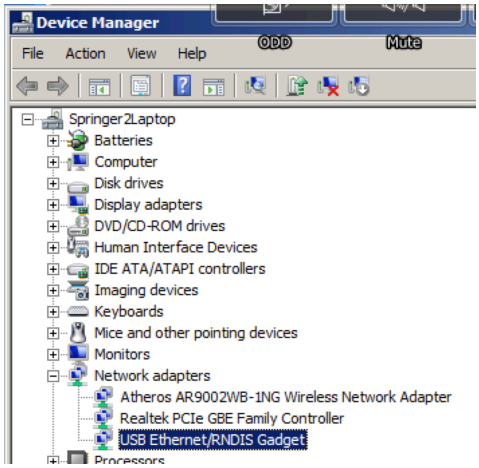



**Figure 3: The Pi appears as an RNDIS network adapter in Device Manager**

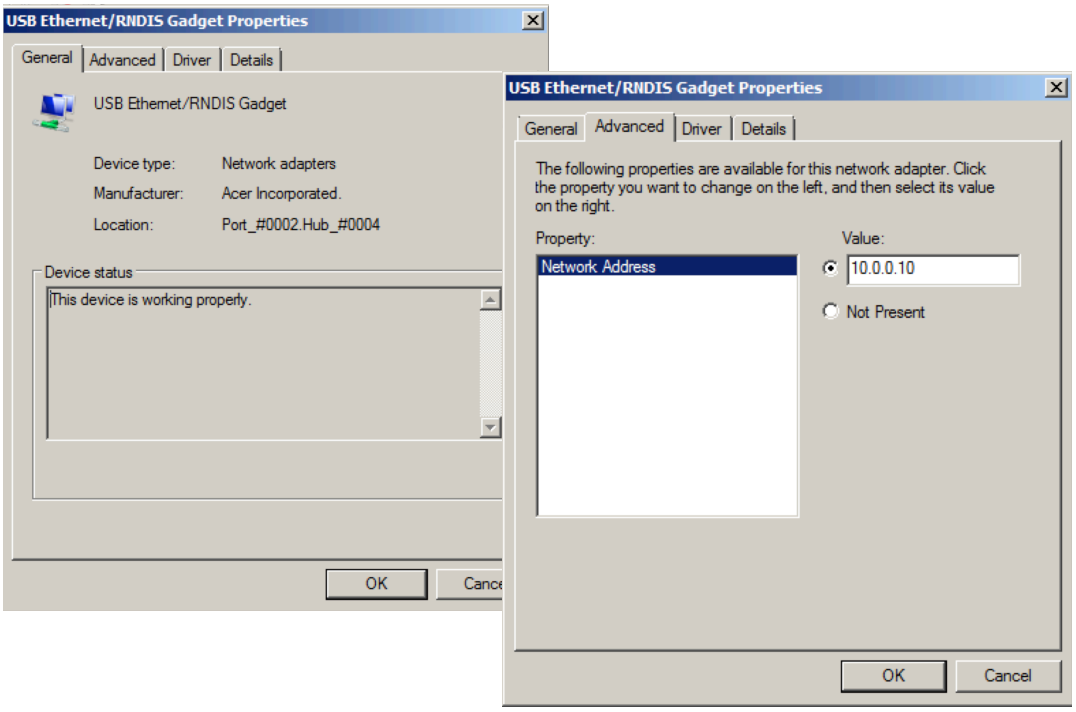



**Figure 4: Setting the IP address for the Pi**

Conceptually, the Pi sits anywhere in IP address space. We set the Pi up to respond to IP address 10.0.0.10. This must be set inside the communicating host computer. First, set the low-level device in the device manager, See Figs 3 and 4 to see what to do.

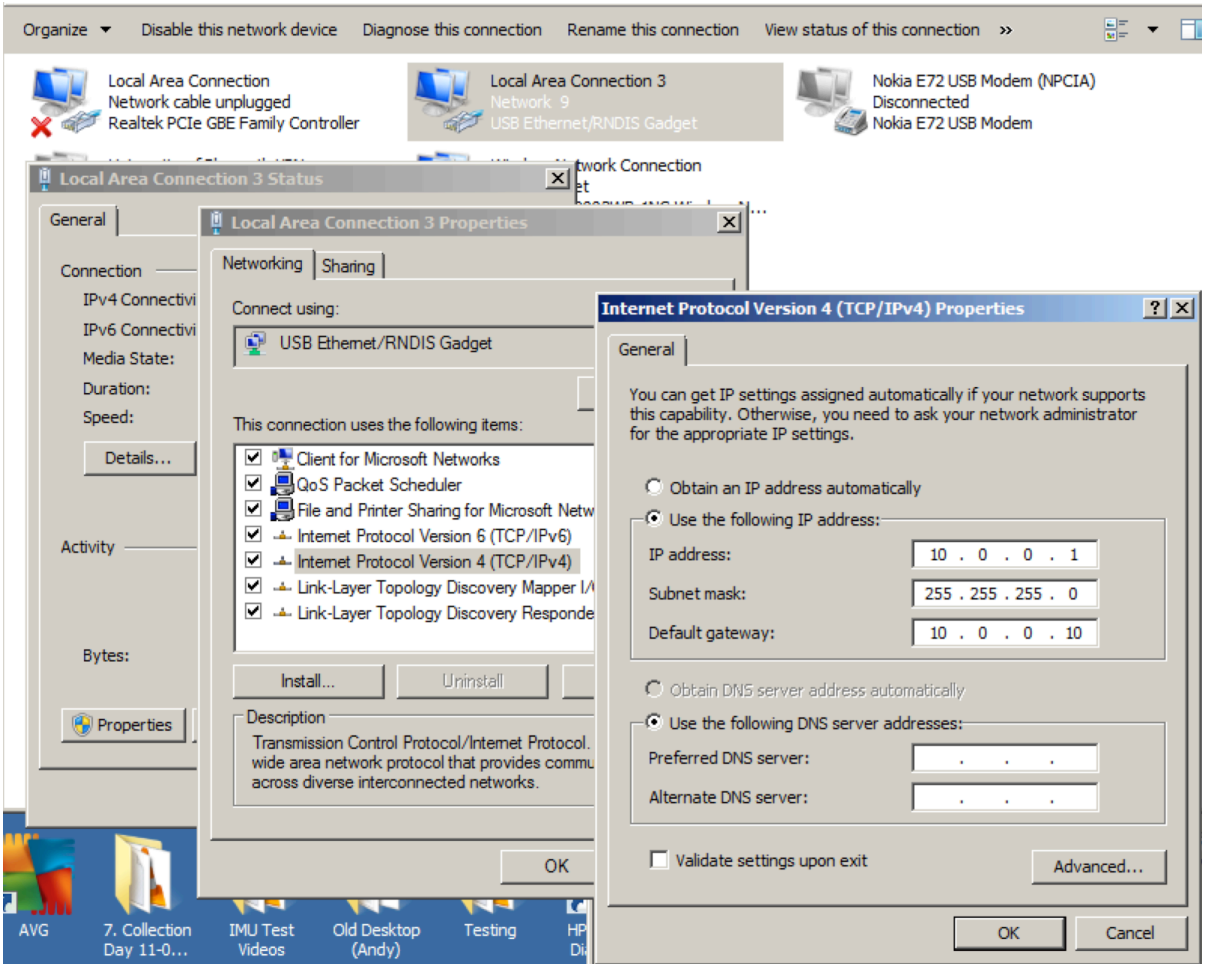



**Figure 5: PC network settings to set the IP route from the PC to the Pi**

**WIN10: See Fig A9-6 to A9-9 for the PC under WIN10.**

It has changed under win10. The device configuration is more automatic.

To summarise, the students can do this by opening Control Panel and clicking Network and Sharing Centre (Fig,6)

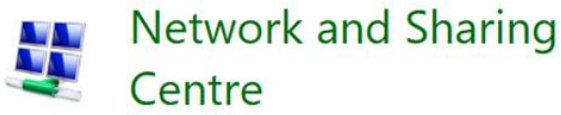



**Figure. 6 control panel icon**

Then, in the left hand menu, click ‘Change adapter settings’ (Fig.7).


**Figure 7. Change adapter settings**

Right click the USB Network Device and click properties (enter password when prompted), see vFig.8.

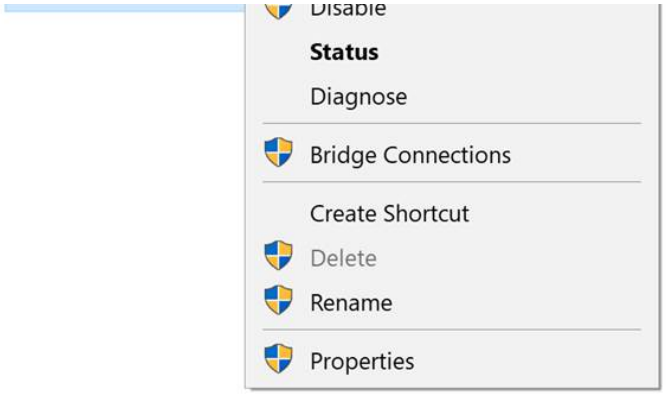



**Figure.8 Network device menu**

Select 'Internet Protocol Version 4 (TCP/IPv4)' and click Properties, see Fig.9

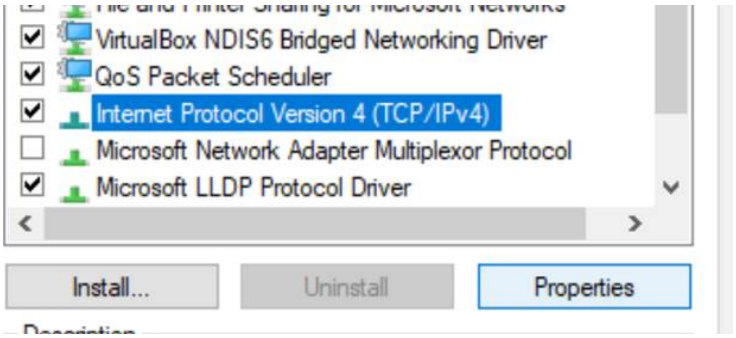



**Figure 9: IP settings (v4)**

Users can then enter manual IP address settings for the device, as for win7. Use 10.0.0.1, with a sub-net mask of 255:255:255:0.