The Jelly Counter

Jonathan Kalani Wein Yau Steck, Claremont McKenna College

Mentor: James Bellingham

Summer 2014

Keywords: Computer Vision, Jellies, LRAUV

ABSTRACT

This paper describes an algorithm for visually counting and tracking jellies through frames of video on an embedded system. Visual tracking of jellies is necessary to answer certain biological questions about individual jellies instead of a population. Visual tracking was chosen because of the limited sonar target strength of jellies and their lack of bodily structure need for tagging. Algorithm development was performed on MATLAB. The C++ code, used to run on the embedded system, was later developed to mimic said MATLAB code but also edited to run in real time. The C++ code utilized the openCV libraries and dependencies. Blob analysis was used to process each frame of video and segment out jellies in the foreground. The end result of having a counting and tracking algorithm run in real time on a Jetson TK1 embedded system was reached.

1 INTRODUCTION

This paper describes an algorithm for the autonomous counting of jellies. Said code can be easily adapted for in situ biomass calculations as well as long term jelly tracking when combined with a control system. The project originates from the Monterey Bay Aquarium Research Institute (MBARI) under the jurisdiction of the long range autonomous under water vehicle (LRAUV) lab.

1.1 Motivation

Marine research encompasses ninety-five percent of the biosphere, the volume of space available to life on the Earth (Earle, 1995). There is a wealth of important and interesting biological questions yet to be answered. Marine biologists, unlike their terrestrial counterparts, cannot easily observe animal behavior in their native habitats (Rife, 2004). A tracking algorithm combined with the long range capabilities of our LRAUVs would prove the means to answer many new oceanic biological questions pertaining to specific organisms rather than populations. Jellies, in particular, offer many challenges for long term observations: the sonar target strength of gelatinous animals is very weak (Monger et al., 1998; Mutlu, 1996) and their gelatinous bodies provide little structural support for transmitter beacons. Visual tracking provides a solution to these problems.

Figure 1: Artist concept for a jelly tracking AUV

Computer vision combined with AUVs allow for longer duration tracking sessions without the costly presence of a manned ship. Our LRAUVs could track a single jelly for a week or more while minimizing operation costs.

1.2 Computer Vision: A Brief History

Computer vision can be tracked back to its roots in the 60s with the polyhedral recognizing "blocksworld" work of Lawrence Roberts (1965). He used mathematical models and projective transformations to analyze grey-scale pictures. His work laid the groundwork for edge detector algorithms (Rosenfeld, 1969; Duda and Hart, 1970) which in turn led to the modern edge detection: Sobel (1982), Canny (1986), Prewitt (1970), and fuzzy logic methods. As these edge detector algorithms were being honed, an alternate technique for image analysis was in its incipient stages. Image segmentation, picking out sets of associated pixels, was invented by Brice and Fenema (1969; 1970). Their work on digitized pictures treated connected, similar intensity pixels as "atomic regions of uniform gray scale" now called blobs. Image segmentation is important because of its utility and lack of complexity. To extract useful information from images it is necessary to segment objects from the background (Zhang, 2009). Much work has been published image segmentation (Pavlidis, 1972; Rosenfeld, 1979) and has grown exponentially since the 80s (Zhang, 2009).

1.3 Morphological Operations

Morphological filters are nonlinear signal transformations that locally modify geometric features (Maragos and Schafer, 1987). The basic morphological operators are dilation, erosion, opening, and closing. All the operations are based on a structuring element, a disk of radius *r* in this paper. The structuring element is sampled over each pixel and alters the image depending on the operation it is performing. Morphological dilation will basically thicken any lines and fill small holes. It is given by Equation 1 and visually represented in Figure 2.

$$A \oplus B = \bigcup_{b \in B} A_b$$

Equation 1: Morphological dilation – if the structuring element anchor point (middle pixel in this paper) contains a foreground point, the foreground then equals the union of the foreground and the structing element.

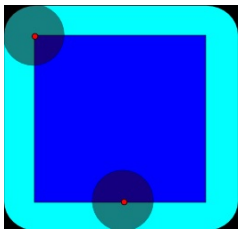

Figure 2: The dilation of the dark-blue square by a disk, resulting in the light-blue square plus the original dark-blue square (http://commons.wikimedia.org/wiki/File:Dilation.png#mediaviewer/File:Dilation.png).

Morphological erosion is the converse of dilation. It will basically thin lines and lessen noisy edges. It is defined by Equation 2 and visually represented in Figure 3.

$$A \ominus B = \{z \in E | B_z \subseteq A\}$$

Equation 2: Morphological erosion – when the structuring element is completely contained in foreground points, the new edge of the foreground is defined by the structuring element anchor (middle pixel).

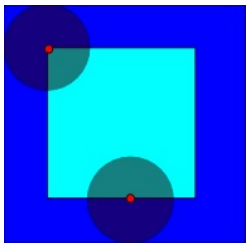

Figure 3: The erosion of the dark-blue square by a disk, resulting in the light-blue square (http://commons.wikimedia.org/wiki/File:Erosion.png#mediaviewer/File:Erosion.png).

Morphological opening is then defined as the erosion and then dilation of an image using the same structuring element. Morphological closing is the dilation and then erosion of an image using the same structuring element. Opening and closing are defined, respectively, in Equation 3 and Equation 4.

$$A \circ B = (A \ominus B) \oplus B$$

Equation 3: Morphological opening – erosion then dilation

$$A \bullet B = (A \oplus B) \ominus B$$

Equation 4: Morphological closing – dilation then erosion

The white top-hat transform (Bright and Steel, 1987) is then given by Equation 5 and will basically pick out objects in the foreground that are smaller than the structuring element and brighter than their surroundings.

$$T_w(f) = f - f \circ b$$

Equation 5: White top-hat filter – the original image minus the opened image

2 MATERIALS AND METHODS

Code was developed using a methodology similar to the MBARI CANON-HiPP (Controller, Agile, and Novel Observing Network-High Performance Processing) structure: "MATLAB →C→Sea". Code structure was developed in MATLAB, preserved in C++ while taking runtime into consideration, and finally ported to an embedded system.

2.1 MATLAB Background Flattening

Primarily, MATLAB code was written to count jellies moving through the video field. The Image Processing Toolbox and the Computer Vision System Toolbox were used but the Computer Vision System Toolbox was later eliminated from the code. Because of the forward scattering light due to the upward facing camera, much effort was spent attempting to flatten the variable background. Initially a test frame was taken every *n* frames and was subtracted from the subsequent *n* frames. This method assumed each test frame was the background for each of the following frames. This was improved upon by averaging *m* frames together and subtracting the average frame from the later *n* frames. Figure 4 shows a 3d heat plot of the background variance for empty frames while changing the spacing between test frames and the number of frames averaged.

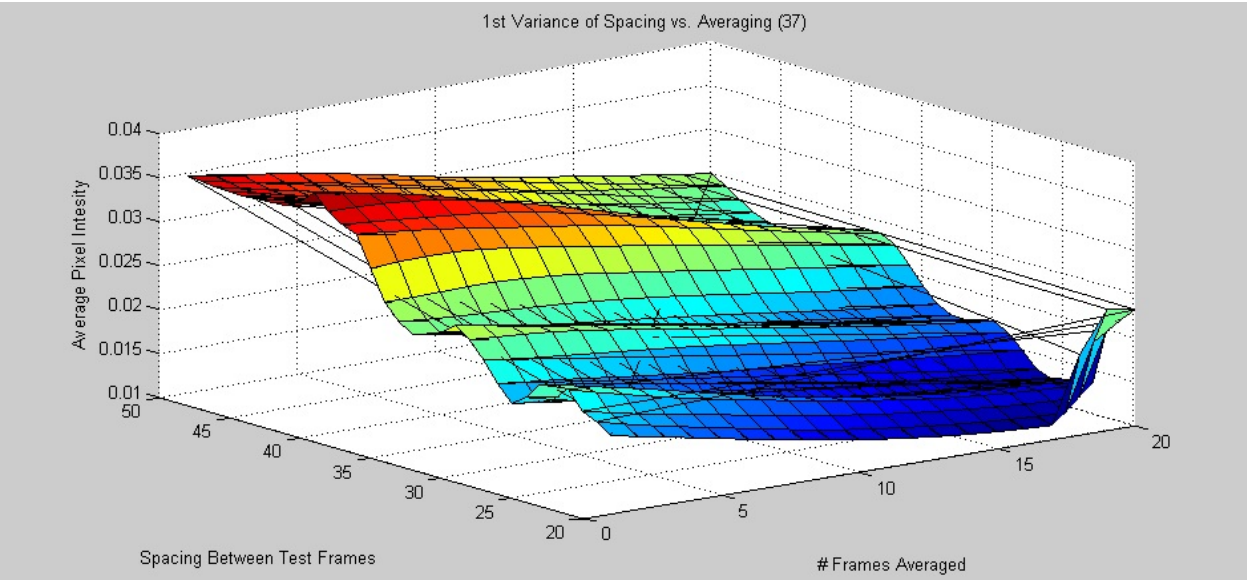

Figure 4: Plot of spacing between test frames and number of frames averaged vs. average pixel intensity

Average frame subtraction, however, did not flatten the background sufficiently for reliable blob analysis. Morphological filters were then tested. Figure 5 and Figure 6 show the pixel variance of negative frames when varying the spacing between test frames and the top hat structuring element size.

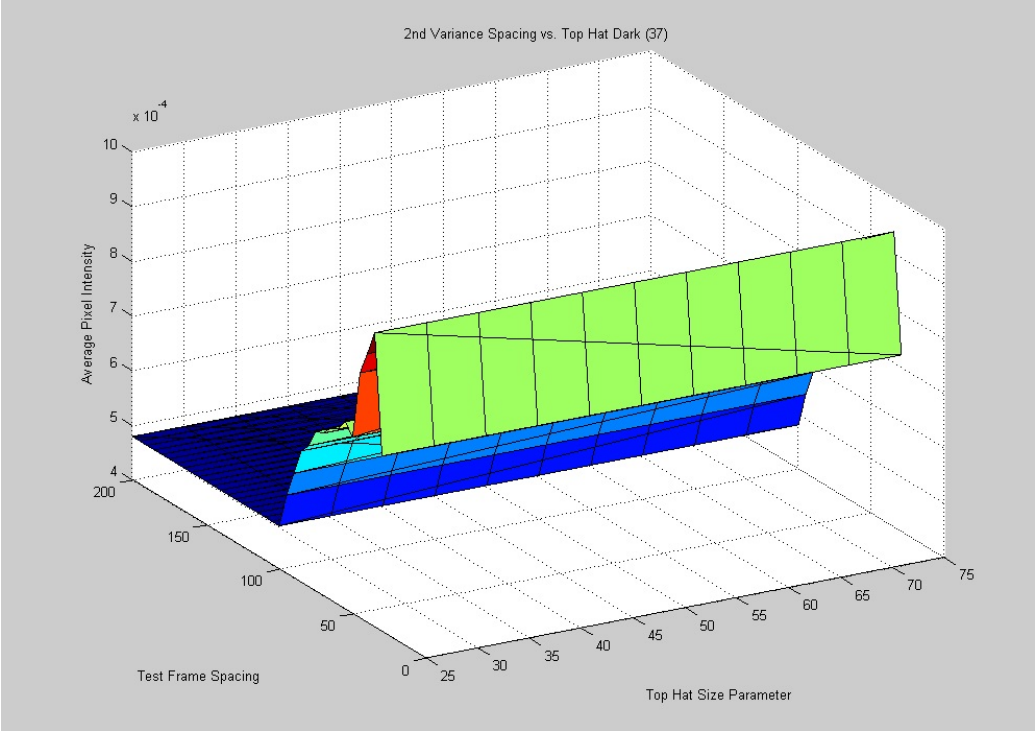

Figure 5: Plot of test frame spacing and top-hat structuring element radius vs. average pixel intensity in dark frames

Two separate plots were generated because the difference between light and dark frames (vertical depth of AUV) was found to be great. In fact following these plots, only the darker frames and sections of video were considered in code development. Figure 5, combined with runtime concerns, lead to dropping the average test frame approach. More specifically, the runtime concerns were the possibility of jellies in test frames and the computational cost of two separate background flattening functions.

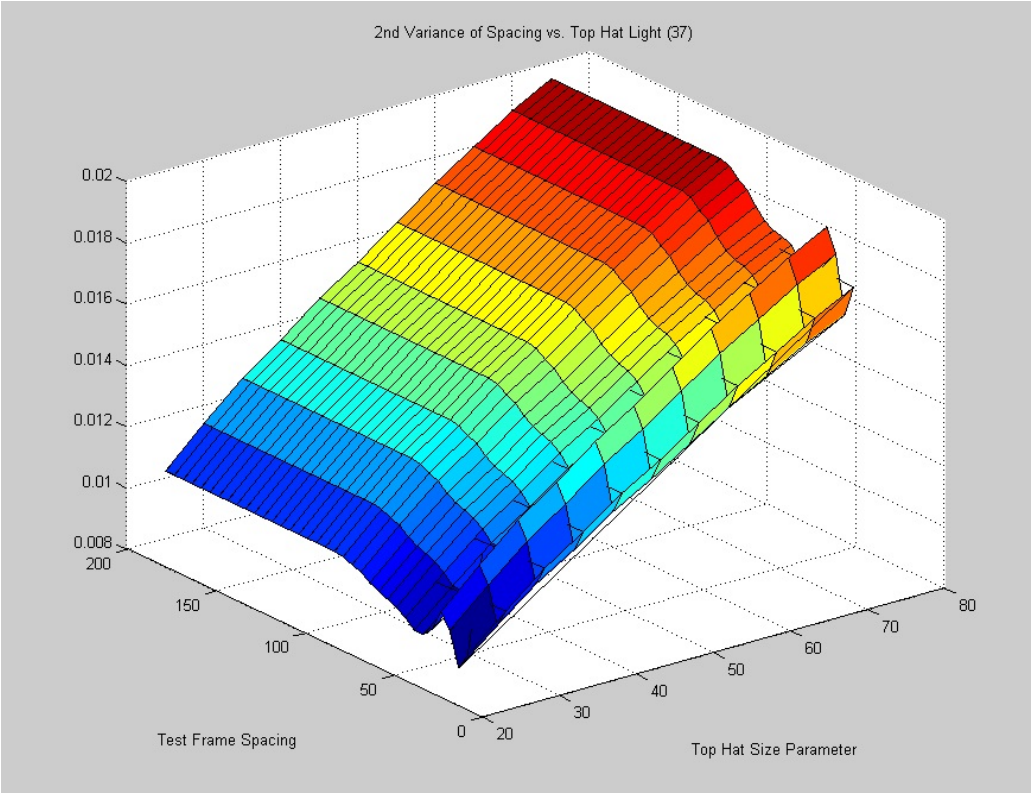

Figure 6: Plot of test frame spacing and top-hat structuring element radius vs. average pixel intensity in light frames

2.2 MATLAB Code Development

First, the video was read into MATLAB using the VideoReader function and set so it steps through each frame one at a time. The VideoReader function took the place of the vision.VideoFileReader object in the Computer Vision System Toolbox. Each 1080p frame was then cropped to 1100x600 pixels to ignore the unwanted edges of the video. The color video frame was then greyscaled. The complement of the image was then taken, effectively turning the dark jellies into bright spots on a dark background. This readied each frame for its set of morphological operations and image manipulations.

White top-hat filtering was then used on each frame to remove back ground variation. A circular structuring element with a radius of 60 pixels was utilized. This filtering reduced the noise sufficiently for thresholding. Because of the top-hat procedure a very low threshold was chosen: 0.03 (values between 0 and 1). A MATLAB specific “hole filling” function for binary images was then used to catch holes in the segmented jellies. Each image was then morphologically opened to clean up the noise and any other holes. A structuring element of

radius 20 was used. The resulting image was a sufficiently clean, segmented binary image. Due to the top-hat procedure, and all morphological filters, the noise on the edge of the image is the greatest. In order to detect the faintest possible jellies, the image was cropped a second time to remove the noisy edges; this allowed for a lower thresholding value without risking more false detects. The image was cropped to 1000x500 pixels. Foreground blobs were then labeled using the bwlabel MATLAB function. Bounding box, centroid, major axis length, and orientation were then calculated using the regionprops function.

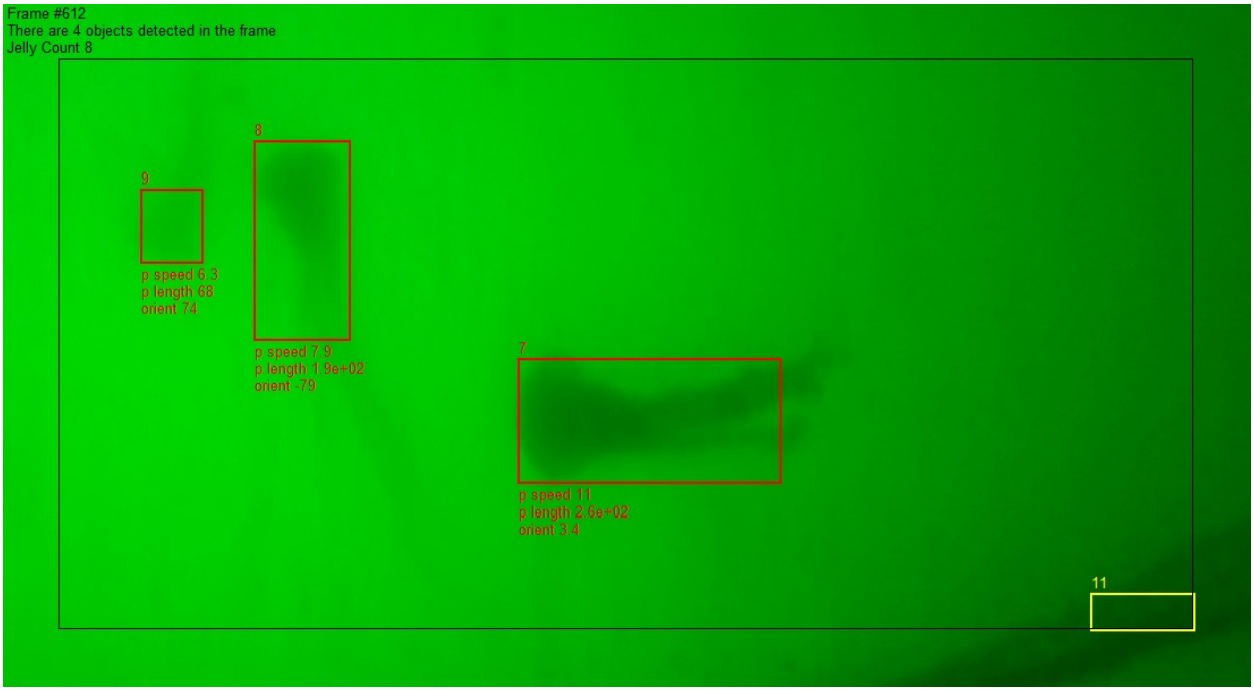

Figure 7: A frame of video processed by MATLAB displaying frame number, objects detected in frame, total jelly count, each region's unique id number, pixel speed, pixel length, orientation relative to the x axis, and color coded locking function.

In order for the computer to know what blobs are the same from frame to frame, a simple proximity function was utilized. If the centroid of the blob was within 50 pixels of the previous frame's blob it was recorded as the same by assigning it the same id number. Blobs could skip up to ten frames before losing the lock. Once blobs were established as the same, pixel speed was calculated and recorded. It was calculated as the pixel distance over the time step. Also pixel length and orientation relative to the x axis were calculated. Once blobs were detected the locking function was implemented. Basically, it takes ten hits of the same blob before changing

the bounding box color from yellow to red and starts recording data. Blob data was only displayed when the jellies were not touching the top or bottom of the frame to display only meaningful data.

A full MATLAB code copy can be found in Appendix A.

2.3 C++ Code Development

C++ code was then developed on a linux box and later ported to an embedded system – the Jetson TK1. The code was built on the standard libraries: stdio.h, iostream, stdlib.h and using OpenCV 2.4.8.2. It was developed to minic the MATLAB code but also run in real time on an embedded system. The Jetson TK1 came with precompiled OpenCV source code with built in CUDA optimizations.

Video files were opened using the VideoCapture function. This same function will open streams of video from webcams, using a similar syntax, and was programmed into the code. As with the MATLAB code, frames were read and processed sequentially. Unlike MATLAB, each frame was down sampled by a factor of ten in each dimension. Also only every other frame was analyzed for blob detection. The image was cropped to a 110x60 pixel box. It was greyscaled using the cvtColor function with the CV_RGB2GRAY flag. Matrix values were then inverted by subtracting them from the same size matrix filled with 255 at every value. The top-hat filter was then implemented using the morphologyEx function with a circular structuring element that fit inside a 12x12 box – radius 6. Matrix values, between 0 and 255, were thresholded with a level of 10. The ROC curve for thresholding values is given in Figure 8. Holes in the foreground blobs were then filled by morphologically opening the image with a structuring element of radius 2. Edge noise was reduced by re-cropping to a 100x50 pixel image. Contours were calculated using the findContours function.

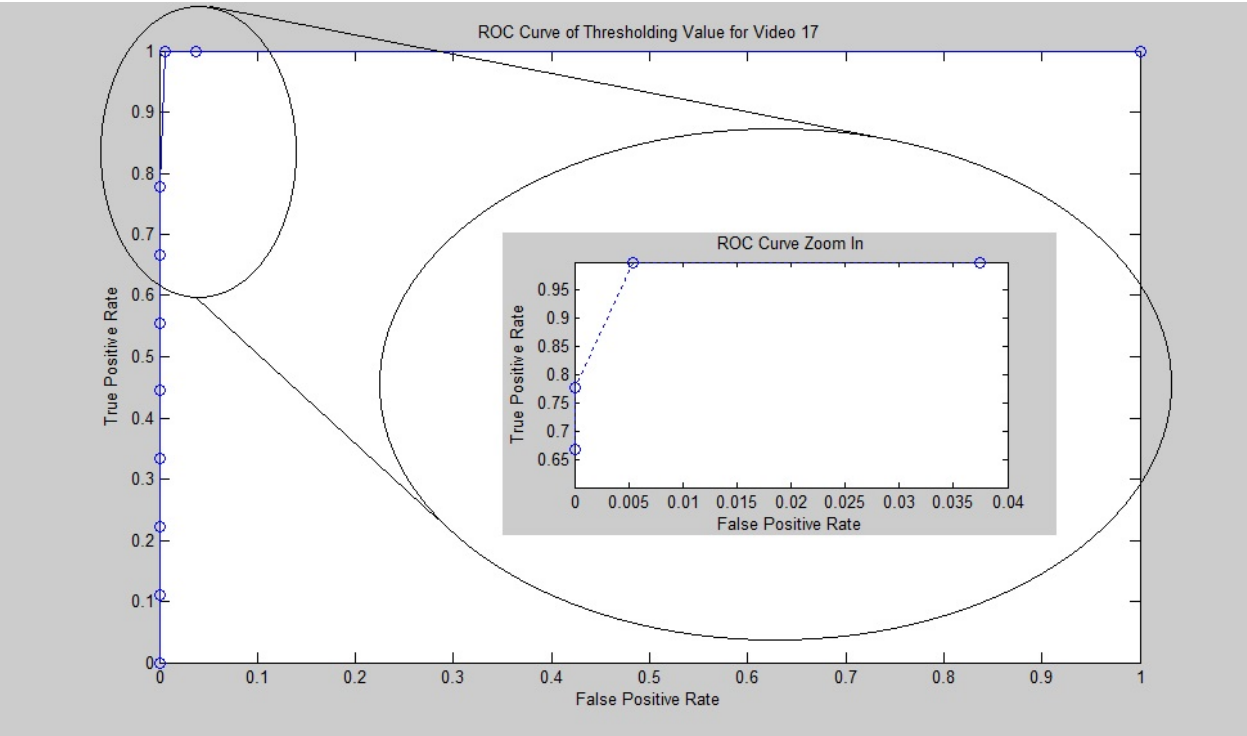

Figure 8: The ROC curve of thresholding values showing about 77% detection rate with no false positives and 100% detection rate with 0.5% chance of false detection was graphed.

For each contour, bounding boxes were calculated. A rotated bounding box was implemented instead of calculating bounding boxes, blob length, and blog orientation to x axis separately. This reduced overall computation and sped up the code. Rotated boxes were calculated by minimizing the area of the bounding box while rotating it around the centroid using the minAreaRect function.

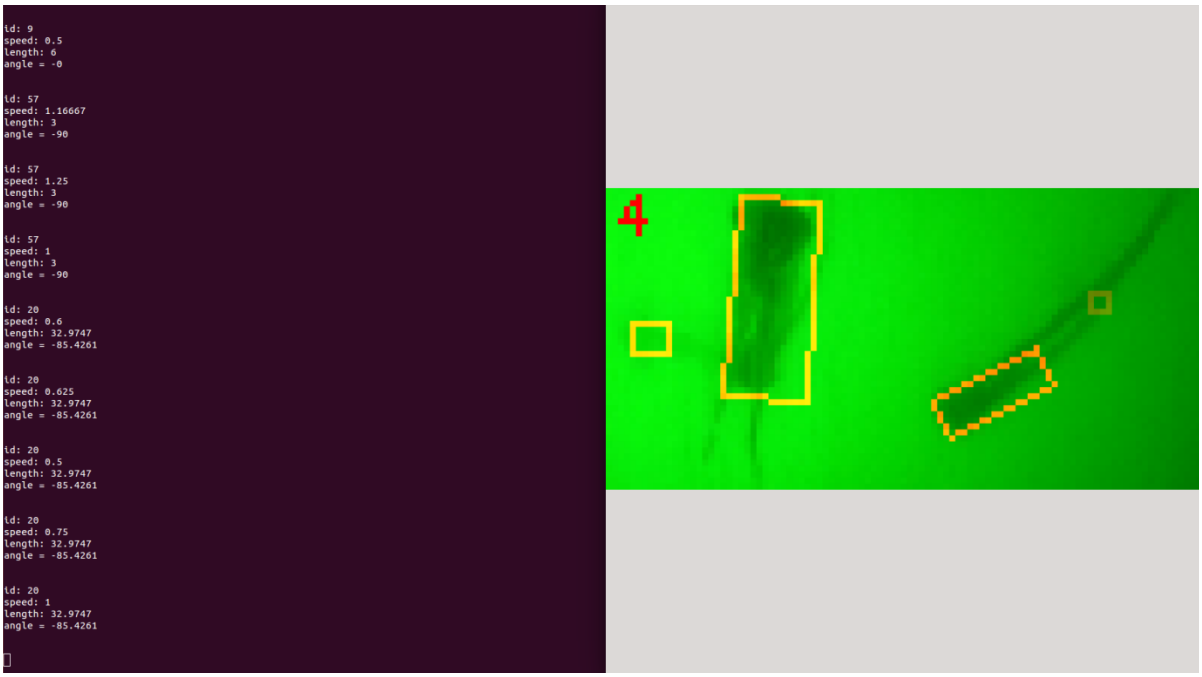

Figure 9: On the right, a frame of video processed by C++ code. On the left, terminal output displayed calculated pixel speed, pixel length, and orientation to the x axis.

Blobs were recognized as the same, frame to frame, with a similar proximity sensor to the MATLAB code. If the top left corner of the bounding box was within five pixels in the x direction and four pixels * difference in frame number in the y direction then blobs were recorded as the same. This was accomplished by assigning blobs a unique id number. Blobs can disappear for up to ten frames before losing lock. Once blobs had been established as the same, pixel speed, pixel length, and orientation to the x axis were calculated. Also the locking function was called which takes five positive hits to lock onto the blob and darkens the bounding box color. A frame capturing function was also added that captures the current frame and saves it as a JPEG.

A full C++ code copy can be found in Appendix B.

3 RESULTS AND DISCUSSION

3.1 False and Missed Detections

Sample videos taken with SCIPI (upward-facing GOPRO on Daphne, a LRAUV) were processed, each with a length of one minute. False detections and missed detections were recorded and plotted in Figure 10. Missed detects were due to either size or faintness. If jellies were smaller than three pixels they were deleted by the morphological opening with structuring element of radius two. If jellies were too faint they did not reach the thresholding level and were not segmented out of the background. Also a jelly appeared directly behind another jelly once, pictured in Figure 9. False detects were usually due to a jelly's close proximity to SCIPI causing a segmentation fault and double counting of the same jelly.

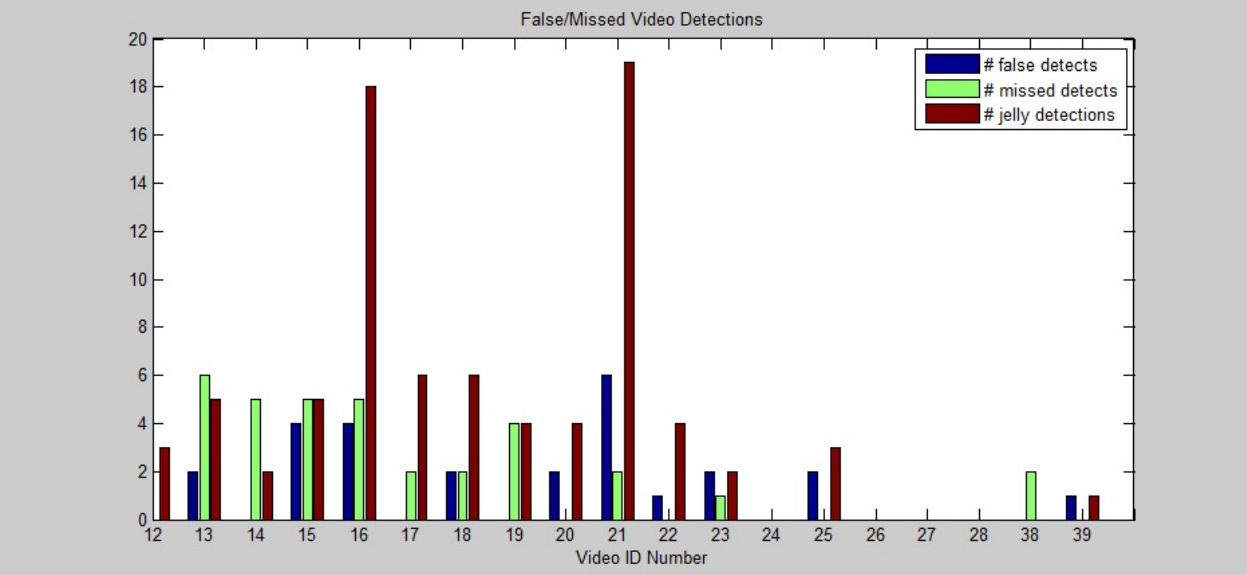

Figure 10: False, missed, and jelly detections for each sample video clip were plotted.

3.2 Benchmarking

In order for an AUV to track jellies its tracking algorithm has to run in real time. It is also necessary for a biomass measurement unless the video is recorded for post processing. Sample video was taken at 30 frames per second for one minute durations. The MATLAB code was only able to process an average of about 0.8 frames per second. This is far too slow and the reason the C++ code was developed.

MATLAB and various versions of the C++ code were benchmarked and compared using video ID number 16. The C++ code versions are named as the following: full – the C++ algorithm without either the downgraded video or every other frame sample (described in section 2.3), downgrade – the C++ code with downgraded video but without every other frame sample, and rotate – the C++ code with downgraded video, every other frame sample, and rotated bounding boxes. The MATLAB code was benchmarked on a Windows computer with a 32 bit double core 2.33 GHz processor with 4 GB of RAM. The C++ code was benchmarked on a Linux computer with a 64 bit 6-core processor with 8 GB of RAM and also the Jetson TK1 embedded system with 4-plus-1 quad core ARM processor with 2 GB of RAM and CUDA optimizations. Figure 11 shows the results of the benchmarking.

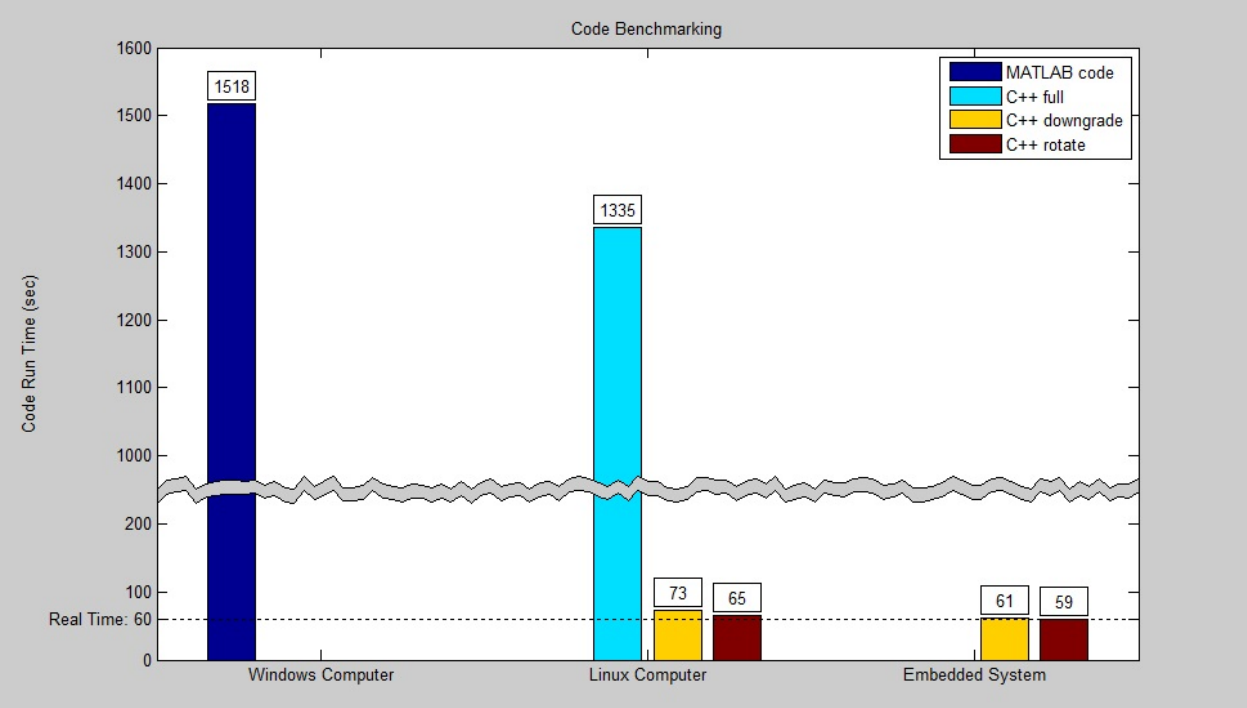

Figure 11: Benchmarking results of various codes on different machines.

MATLAB, though thought to be cumbrous, performed well with the matrix heavy calculations of digital image processing – only 14% slower than the comparable C++ full code. Down sampling the video so it contained 1/100 of the pixels sped up the video processing to near real time computations. Processing only every other frame brought processing down to real time on the Jetson TK1. The Jetson TK1 embedded system performs faster than the Linux computer even though it has a quarter of the memory because of the CUDA optimizations. The CUDA

optimizations basically source out computations to the GPU, which is generally slower than the CPU, but has 192 cores for parallel processing.

There was a couple of seconds variation in run time for each test; recorded values are a rounded average of three different runs of video number 16. It is also important to note that, contrary to all documentation, the VideoCapture function, utilized in the C++ code, will not grab frames faster than their recorded frame rate. A test script was made to confirm this and simply reading in all the frames, without displaying them or performing any manipulations, took 59-60 seconds. This means there should be some extra computational time available in the C++ rotate code to allow for either a biomass calculation or calling control functions without giving up running in real time.

4 CONCLUSIONS AND RECOMMENDATIONS

The C++ code running in real time on an embedded system shows the viability of a jelly tracking algorithm that can be used on AUVs. Also a simple calibration from pixel speed and size to actual speed and volume would allow the code to make biomass estimations on either captured video or in situ measurements.

4.1 Jelly Tracker Further Research

In order to implement a working jelly tracker in an AUV, further research and development is necessary. A control system that takes its input from the C++ rotate code will have to be developed. Also the infrastructure to allow for the streaming of video to the AUV's embedded system will have to be created.

Improvements to the C++ rotate code are also possible. Instead of just orientation of the detected blobs to the x axis, the direction of the jelly should also be included. This function could be implemented various ways. The possibilities include taking the weighted average of the blob's pixel positions and simply testing for the closest side of the oriented bounding box. This simple calculation should not change the run time of the algorithm significantly. Another improvement can be the inclusion of a cascade classifier. There are two steps of cascade classification: training and detection. Training data must be acquired. Two sets of data are necessary: positive samples and negative samples. Once these samples are collected and the classifier is trained then feature based jelly detection can commence. This will lower the chances of false detections but also add some computational cost. Unfortunately, the down welling light backlights the jellies causing complications for feature based detection. The variability of the jelly community combined with the variability of specific jellies sampled at different times also complicates the issue.

4.2 Biomass Calculation Further Research

Simple calibrations are necessary to implement the C++ rotate code for biomass calculations. The pixel speed must be converted to distance from the AUV. This conversion assumes the jellies are static relative to the AUV speed. The calculation is based on parallax, not from stereo cameras, but from the viewpoint difference as the AUV flies under the jelly and can

be calculated accordingly. Once distance from jelly has been established, pixel length can be converted to volume of the animal. Volumes and number of jellies can be combined for a simple biomass measure.

ACKNOWLEDGEMENTS

I would like thank my mentor, Jim Bellingham, for all his insights and creating the jelly tracking project. I would also like to thank the LRAUV lab for "moral support" as well as actual help. More specifically I would like to thank Brian and Jordan for teaching me a hatred for dependencies, Thomas for MATLAB magic, and others for all their help. A big thank you to the CANON-HiPP project for supplying both a Linux computer and a Jetson TK1 embedded system, it rips. The MBARI internship is awesome and I would like the thank George and Linda for running it. And finally the Packard Foundation for footing all the bills.

APPENDIX A:

MATLAB code: jellyCounter.m

```
tic
jVid = VideoReader('C:\Users\jsteck\Documents\MATLAB\videos\GOPR6616.MP4');
nFrames = jVid.NumberOfFrames;
id = 1;
jCount = 0;
c = cell([2000,5]); %c{:,1}frame#,c{:,2}id#,c{:,3}(1)centroidx,pixel distance traveled,#hits
cLength = 1;

for i = 1:nFrames
    frame = read(jVid,i);
    frameCrop = imcrop(frame,[550,150,1100,600]); %xmin ymin width height
    frameGrey = rgb2gray(frameCrop);
    frameComp = imcomplement(frameGrey);
    frameTopHat = imtophat(frameComp,strel('disk',60));
    frameThresh = im2bw(frameTopHat,.03);
    frameFilled = imfill(frameThresh,'holes');
    frameOpen = imopen(frameFilled,strel('disk',20));
    frameCrop2 = imcrop(frameOpen,[50,50,1000,500]);
    imshow(frameCrop);
    [frameLabeled,numObj]=bwlabel(frameCrop2);
    stats = regionprops(frameLabeled,'BoundingBox','Centroid','MajorAxisLength','Orientation');
    for k = 1:length(stats)

        c{cLength,1} = i;
        c{cLength,2} = id;
        c{cLength,3} = stats(k).Centroid;
        c{cLength,4} = 0;
        c{cLength,5} = 0;

        for m = cLength-1:-1:1
            if i-c{m,1}>10
                break
            end
            if abs(c{m,3}(1)-stats(k).Centroid(1)) < 50 && abs(c{m,3}(2)...
                    -stats(k).Centroid(2)) < 50
                c{cLength,2} = c{m,2};
                id = id - 1;
                c{cLength,4} = (stats(k).Centroid(2)-c{m,3}(2))/(i-c{m,1});
                c{cLength,5} = c{m,5}+1;
                break
            end
        end

        rect = rectangle('Position',stats(k).BoundingBox+[50,50,0,0],...
            'LineWidth',2,'Tag',int2str(k));
        numLab = text(stats(k).BoundingBox(1)+50,stats(k).BoundingBox(2)+40,...
            int2str(c{cLength,2}));
        if c{cLength,5}==10
            jCount = jCount + 1;
        end
        if c{cLength,5}<10
```

```
            set(rect,'EdgeColor','y');
            set(numLab,'Color','y');
        else
            set(rect,'EdgeColor','r');
            set(numLab,'Color','r');
        end

        if stats(k).BoundingBox(2) > 1 && stats(k).BoundingBox(2)+stats(k).BoundingBox(4) < 500 ...
            && c{cLength,5}>=10
            text(stats(k).BoundingBox(1)+50,stats(k).BoundingBox(2)+stats(k).BoundingBox(4)+60,...
                ['p speed ',num2str(c{cLength,4},2)],'Color','r');
            text(stats(k).BoundingBox(1)+50,stats(k).BoundingBox(2)+stats(k).BoundingBox(4)+75,...
                ['p length ',num2str(stats(k).MajorAxisLength,2)],'Color','r');
            text(stats(k).BoundingBox(1)+50,stats(k).BoundingBox(2)+stats(k).BoundingBox(4)+90,...
                ['orient ',num2str(stats(k).Orientation,2)],'Color','r');
        end

        cLength = cLength + 1;
        id = id + 1;

    end
    rectangle('Position',[50,50,1000,500]);
    text(5,10,['Frame #',int2str(i)]);
    text(5,25,['There are ',int2str(numObj),' objects detected in the frame']);
    text(5,40,['Jelly Count ',int2str(jCount)]);
    pause(.01);
end

toc
```

APPENDIX B:

C++ rotate code: jellyTrackerRotated.cpp

```
/*
* jellyTrackerRotated.cpp
*
* A Jon Steck Corporation Code
*
* "Please work"
*      -Jon
*
*--every other frame sample--
*
* rotated bounding boxes
*/

#include <opencv/cv.h>
#include <opencv2/opencv.hpp>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

//hide the local functions in an anon namespace/same as static function
namespace {
   void help(char** av) {
      cout << "The program captures frames from a video file, image sequence (01.jpg, 02.jpg ... 10.jpg) or camera
connected to your computer." << endl
           << "Usage:\n" << av[0] << " <video file, image sequence or device number>" << endl
           << "q,Q,esc -- quit" << endl
           << "space   -- save frame" << endl << endl
           << "\tTo capture from a camera pass the device number. To find the device number, try ls /dev/video*" <<
endl
           << "\texample: " << av[0] << " 0" << endl
           << "\tYou may also pass a video file instead of a device number" << endl
           << "\texample: " << av[0] << " TheKardashiansGoToTheJerseyShore.avi" << endl
           << "\tYou can also pass the path to an image sequence and OpenCV will treat the sequence just like a
video." << endl
           << "\texample: " << av[0] << " TheBachorette7.jpg" << endl;
   }

   int process(VideoCapture& capture) {
      time_t start = time(0);
           int n = 0;
      char filename[200];
      string window_name = "video | q or esc to quit";
      cout << "press space to save a picture. q or esc to quit" << endl;
      namedWindow(window_name, WINDOW_NORMAL); //normal window;
      Mat frame;
           int mem[10000][5];
           double speedSizeAngle[10000][3];
           int memStart = 0;
           int memEnd = 0;
```

```
    int frameNum = 1;
    int id = 0;
    int jCount = 0;

for (;;) {
   capture >> frame;
   if (frame.empty())
      break;
   if (frameNum % 2 == 0)
                {
                Mat dGrade;
    resize(frame,dGrade,Size(),.1,.1,INTER_LINEAR);
                Rect r(55,15,110,60);
                Rect r2(60,20,100,50);
                Mat vis = dGrade(r2);
                Mat small = dGrade(r);
                Mat grey;
                cvtColor(small,grey,CV_RGB2GRAY);
                Mat comp, temp;
                temp = Mat::ones(grey.rows,grey.cols,0);
    temp = temp*255;
                comp = temp - grey;
    Mat topHat, struct1;
    struct1 =  getStructuringElement(2,Size(12,12),Point(-1,-1));
                morphologyEx(comp,topHat,MORPH_TOPHAT,struct1);
    Mat thresh;
    threshold(topHat,thresh,10,255,THRESH_BINARY);
    Mat opened, struct2;
                struct2 = getStructuringElement(2,Size(4,4),Point(-1,-1));
                morphologyEx(thresh,opened,MORPH_OPEN,struct2);
                Rect r3(5,5,100,50);
                Mat smaller = opened(r3);

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
                //vector<Point2f> center(contours.size());
                //vector<float>radius( contours.size() );
    findContours(smaller,contours,hierarchy,CV_RETR_TREE,CV_CHAIN_APPROX_SIMPLE,Point(0,0));
                /// Approximate contours to polygons + get bounding rects and circles
                vector<vector<Point> > contours_poly( contours.size() );
    vector<RotatedRect> minRect( contours.size() );
                Mat drawing = Mat::zeros( vis.size(), CV_8UC3 );

                for( int i = 0; i < contours.size(); i++ )
                        {
                        minRect[i] = minAreaRect( Mat(contours[i]) );
                        //minEnclosingCircle( (Mat)contours_poly[i], center[i], radius[i] );
                        mem[memEnd][0] = frameNum;
                        mem[memEnd][1] = id;
                        id ++;
                        approxPolyDP( Mat(contours[i]), contours_poly[i], 3, true );
                        //boundRect[i] = boundingRect( Mat(contours_poly[i]) );
                        mem[memEnd][2] = int(minRect[i].center.x);
                        mem[memEnd][3] = int(minRect[i].center.y);
                        mem[memEnd][4] = 0;
                        speedSizeAngle[memEnd][0] = 0;
```

```
                        speedSizeAngle[memEnd][1] = 0;
                        speedSizeAngle[memEnd][2] = 0;
                        for( int k = memStart; k < memEnd; k++)
                                {
                                if(mem[memEnd][0]-mem[k][0] > 10)
                                        {
                                        memStart++;
                                        }
                                if(abs(mem[k][2]-mem[memEnd][2]) < 5 && abs(mem[k][3]-
mem[memEnd][3]) < 4*(mem[memEnd][0]-mem[k][0]))
                                        {
                                        mem[memEnd][1] = mem[k][1];
                                        mem[memEnd][4] = mem[k][4]+1;
                                        if(mem[memEnd][3] != 0)
                                                {
                                                speedSizeAngle[memEnd][0] = double(mem[memEnd][3]-
mem[k][3])/double(mem[memEnd][0]-mem[k][0]);
                                                if(minRect[i].size.height > minRect[i].size.width)
                                                        {
                                                        speedSizeAngle[memEnd][1] =
minRect[i].size.height;
                                                        }
                                                else
                                                        {
                                                        speedSizeAngle[memEnd][1] =
minRect[i].size.width;
                                                        }
                                                speedSizeAngle[memEnd][2] = minRect[i].angle;
                                                cout << "id: " << mem[memEnd][1] << endl;
                                                cout << "speed: " << speedSizeAngle[memEnd][0] << endl;
                                                cout << "length: " << speedSizeAngle[memEnd][1] << endl;
                                                cout << "angle = " << speedSizeAngle[memEnd][2] << endl;
                                                cout << "\n" << endl;
                                                }
                                        }
                                }

                        if(mem[memEnd][4] == 5)
                                {
                                jCount++;
                                }
                        if(mem[memEnd][4]>4)
                                {
                                for( int j = 0; j < 4; j++)
                                        {
                                        Point2f rect_points[4]; minRect[i].points(rect_points);
                                        line( drawing, rect_points[j], rect_points[(j+1)%4], Scalar(0,0,255), 1,
8);
                                        }
                                }
                        else
                                {
                                for( int j = 0; j < 4; j++)
                        {
                        Point2f rect_points[4]; minRect[i].points(rect_points);
                        line( drawing, rect_points[j], rect_points[(j+1)%4], Scalar(0,0,100), 1, 8);
```

```
                        }
                    }
                        memEnd++;
                        }

            /// Draw polygonal contour + bonding rects

            Mat combined;
            combined = vis + drawing;

                        string String = static_cast<ostringstream*>( &(ostringstream() << jCount) )->str();
            putText(combined,String,Point(1,7),FONT_HERSHEY_SIMPLEX,0.3,Scalar(0,0,255),1,8,false);

                        imshow(window_name, combined);
            char key = (char)waitKey(3); //delay N millis, usually long enough to display and capture input

            switch (key) {
            case 'q':
            case 'Q':
            case 27: //escape key
               return 0;
            case ' ': //Save an image
               sprintf(filename,"filename%.3d.jpg",n++);
               imwrite(filename,frame);
               cout << "Saved " << filename << endl;
               break;
            default:
               break;
            }
                        }
            frameNum++;
        }
            time_t end = time(0);
            double time = difftime(end,start);
            cout << "You are now " << time << " sec older" << endl;
        return 0;
    }
}

int main(int ac, char** av) {

    if (ac != 2) {
        help(av);
        return 1;
    }
    std::string arg = av[1];
    VideoCapture capture(arg); //try to open string, this will attempt to open it as a video file or image sequence
    if (!capture.isOpened()) //if this fails, try to open as a video camera, through the use of an integer param
        capture.open(atoi(arg.c_str()));
    if (!capture.isOpened()) {
        cerr << "Failed to open the video device, video file or image sequence!\n" << endl;
        help(av);
        return 1;
    }
    return process(capture);
}
```

REFERENCES:

Brice, Claude R.; Fennema, Claude L. (1970). "Scene Analysis Using Regions." *Technical Note 17.*

Bright, David S.; Steel, Eric B. (1987). "Two-dimensional top hat filter for extracting spots and spheres from digital images." *Journal of Microscopy*. 146:191-200.

Canny, John. (1986). "A Computational Approach to Edge Detection," *Pattern Analysis and Machine Intelligence.* Vol PAMI-8. Issue 6. 679-698.

Duda, Richard O.; Hart, Peter E. (1970). "Experiments in scene analysis." No. SRI-TN-20. STANFORD RESEARCH INST.

Earle, S. A. (1995). "Sea Change: A Message of the Oceans." Ballantine Books.

Fennema, Claude; Brice, Claude. (1969). "A Region-Oriented Data Structure." *Technical Note 7. AI Center.*

Maragos, P.; Schafer, R. W. (1987). "Morphological filters – Part 1: Their set-teoretic analysis and relations to linear shift invariant filters." *IEEE Transactions on Acoustics, Speech and Signal Processing.* 35:1153-1169.

Monger, B. C.; Chinniah-Chandy, S.; Meir, E.; Billings, S.; Greene, C. H.; Wiebe, P. H. (1998). "Sound scattering by the gelatinous zooplankters Aequorea victoria and Pleurobrachia bachei." *Deep-Sea Research II*. 45:1255-1271.

Mutlu, E. (1996). "Target Strength of the common jellyfish (Aurelia aurita): a preliminary experimental study with a dual-beam acoustic system." *ICES Journal of Marine Science*. 53:309-311.

Pavlidis, Theodosios. (1972). "Segmentation of pictures and maps through functional approximation." *Computer Graphics and Image Processing* 1.4. 360-372.

Prewitt, Judith M. (1970). "Object Enhancement and Extraction." *Picture Processing and Psychopictorics 10*. 15-19.

Rife, Jason. (2004). *Automated Robotic Tracking of Gelatinous Animals in the Deep Ocean*. Ph.D. Thesis. Dept. of Mechanical Engineering. Stanford University.

Roberts, Lawrence G. (1965). “Machine Perception of three-dimensional solids.” *Optical and Electro-optical Information Processing*. 159-197.

Rosenfeld, Azriel. (1969). “Picture Processing by Computer.” *ACM Computing Surveys*. 147-176.

Rosenfeld, Azriel. (1979). "Fuzzy digital topology." *Information and Control* 40.1. 76-87.

Sobel, Michael E. (1982). “Asymptotic Confidence Intervals for Indirect Effect in Structural Equation Models.” *Sociological Methodology*. Vol 13. 290-312

Zhang, Yu-Jin. (2009). "Image Segmentation in the Last 40 Years." *Encyclopedia of Information Science and Technology, Second Edition.* 1818-1823.