Man Overboard

Group number 18

AishwaryaMala GM, a.m.gurusamymuthuvelrabindran@student.utwente.nl, S2242842, Electrical Engineering
Christopher Benjamin Leonard, c.benjaminleonard@student.utwente.nl, S2378515, Embedded Systems
Chidura Bhanu Teja, b.t.chidura@student.utwente.nl, S2304791, Systems and Control

***Abstract*—The Man Overboard project is aimed at tracking the test buoy in the video given the initial position and then estimate its distance from the camera in meters. The algorithm proposed involves computing a motion model for the movement of the buoy. Using the motion model an ROI is defined on each video frame. This ROI is expected to contain the buoy's location within it. The ROI is then thresholded to help separate the buoy from its background and calculate its exact position. Then we used the difference in angle between consecutive buoy positions along with a spherical model for Earth to compute the distance of the buoy from the camera. For the first video frame, when the previous buoy location is not known, the angle between the buoy and the horizon was used for calculations.**

***Index Terms*—Distance Estimation, Object Tracking, Optic Flow Motion model.**

I. INTRODUCTION

The Man Overboard project is an attempt to use Image Processing and Computer Vision techniques to find a person lost in the sea – 'Man Overboard' and to track and to estimate the distance of the person location from the vessel to help and plan the rescue operations.

This project, in particular, concentrates on the tracking and the distance estimation parts of the Man Overboard project. Even when a person is spotted, it is difficult to keep track of the person's position, especially in a wavy sea environment. In this regard, this project aims at exploring the possibilities of tracking an object in a wavy sea environment given its initial position.

We have a video of a test buoy at sea as an input. Given the initial location of the buoy, the algorithm developed must be able to track and locate the buoy throughout and then calculate its distance from the camera in meters.

This report outlines the various steps that are part of the algorithm. As a preprocessing step, video stabilization was used to compensate for the unwanted camera movements during the recording process. The preprocessed video was then used to compute a motion model for the buoy. This model was then used to define a region of interest (ROI) for the search of the buoy. The buoy's exact locations are obtained from the ROI and these are further used to calculate the distance between the camera and the buoy.

II. METHODS AND MATERIALS

A. Materials

As input, we have a video of a test buoy in a wavy sea environment and the initial location of the buoy.

To help with the distance estimation aspect of the problem statement, we are given the height of the camera from the sea level during the recording and camera calibration images from which the intrinsic parameters of the camera could be obtained.

The algorithm was developed on MATLAB.

B. Methods

The aims of the project are to locate the buoy in each image frame of the input video and to estimate the distance of the buoy from the camera.

The input is an RGB video with 432 frames and each of dimensions 1080x1440. On seeing the input video, we can sense unwanted camera motions during the recording process and hence as a preprocessing step we stabilize the input video. The preprocessed video was then used to estimate a motion model for the buoy. For this, we used the optic flow computed using the Farneback method. Using the motion model and the initial position of the buoy we were able to successfully define an ROI of size 100 x 50 that enclosed the buoy. In order to locate the buoy within the ROI, the ROI was thresholded. This separated the buoy from its background and the centroid gave the exact location of the buoy.

For the second part of the aim, calculating the distance of the buoy from the camera, we first obtained the intrinsic parameters of the camera. This was done using the Camera Calibrator app of the Computer Vision toolbox in MATLAB. Using the intrinsic matrix of the camera, the location of each pixel in the camera coordinates could be obtained. These were further used to obtain the angles that they made with the principal axis of the camera. These angles are then combined with the spherical model of Earth to arrive at the distance of the buoy from the camera.

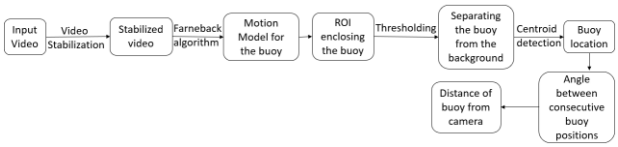

Fig.1. Flowchart of the proposed algorithm.

Below are each of the steps involved in the algorithm explained in detail.

Video Stabilization: As already mentioned, as a preprocessing step we stabilized the input video to remove unwanted camera motions that were captured during the recording process. We used an existing video stabilizing algorithm provided by MathWorks which tracks a target area and identifies how much the target area has moved with respect to the previous frame. Using this information, the algorithm transforms each of the image frames in the video to remove unwanted camera motions and to generate a stabilized video.

Fig.2. Image marked with the target area that was chosen for tracking during video stabilization.

The reason for choosing the above-shown target area is that the landmass in this section of the image has its constant presence during the entire duration of the video and it can act as a reference to correct the distortions bought upon by camera tilting as we know that its angle of orientation around the optical axis must remain constant throughout. This section of the image could also, to an extent, help to correct the sudden up and down camera pitching movements as the position of the landmark does not change drastically within consecutive frames. Panning, the movement of the camera around the vertical axis will cause uneven zooming in the direction in which the camera is moved closer towards the scene. This could also be taken into account but we have not concentrated much on it for now as we did not sense much panning effect in the input video.

Increasing the size of the target could have increased the accuracy of the transforms computed while increasing the execution time. So we limited the area of the target at what we considered the optimum tradeoff point between accuracy and the time taken for execution. The camera motions were corrected using affine transforms as we need to preserve the ratios in the image. Below is the transformed image of the video frame shown in Fig.2. We see that the size of the image has changed and zeros are padded along the rows and columns to maintain the size. In order to remove the zero paddings, we resized each image frame to a size of 1166x1016. The size was arrived at using the maximum number of zero-padded rows and columns amongst all the stabilized video frames. The MATLAB code used for video stabilization and resizing is given in the appendix and their outputs are attached as video files.

Fig. 3. Transformed image

Motion Model Estimation: The aim is to determine the velocities in both x and y directions for the buoy in terms of pixels/frame. For this, we use the optic flow velocity vectors calculated using the Farneback algorithm.

Optical flow is a 2D vector field which shows the displacement or motion of an object, edge or corner in an image, between consecutive frames. There are 2 main variants of optical flow sparse and dense. Sparse optical flow picks out the interesting or the main feature (edges, corners) and provides flow vectors for the same. While dense optical flow provides flow vectors for each pixel of an image. It is slow and expensive but with higher accuracy, there is a trade-off between accuracy and speed.

The Farneback algorithm is a dense optical flow algorithm. The Farneback method [1] is a two-frame motion estimation algorithm that uses quadratic polynomials to approximate the motion between the frames and also uses polynomial expansion to accommodate the neighbours of the pixel. Gaussian filter is incorporated for smoothing neighbouring displacements. The Farneback algorithm generates an image pyramid, where each level has a lower resolution compared to the previous level. The algorithm can track the points at multiple levels of resolution, starting at the lowest level and continuing until convergence. Image pyramids are used to detect large movements in between 2 frames.

MATLAB has inbuilt functions for calculating optical flow

with 4 different methods and *opticalFlowFarneback* is one among them. Due to the small size of the buoy, it was not detected as a key point in any of the algorithms and hence we needed to use dense optical flow estimating algorithms. This is the reason why we had to choose the Farneback algorithm in spite of it being the slowest amongst all the algorithms available in MATLAB. We set the number of pyramid layers as 4, the size of the pixel neighbourhood as 7 and the Gaussian filter size as 20. Increasing the Gaussian filter size makes the algorithm more immune to noise and increasing the number of pyramids and pixel neighbourhood size increases the accuracy of the optic flow vectors.

From the Farneback algorithm, we obtained two matrices. One matrix containing the estimated velocity of each pixel in the x-direction and the other containing the velocity of each pixel in the y-direction. The velocities are of the unit pixels/frame. Given the approximate initial position of the buoy, we created a region around it of size 150x60. We knew that the region should be rectangular with length greater than the width as the displacement of the buoy is more on the x-direction than on the y-direction. The exact length and breadth of the region were obtained after several trials. We predicted the displacement of the buoy's position from the current frame to the next frame by taking the average of all the velocities of the pixels in the region defined around it. We moved the buoy position and the region around it using the displacement calculated and did the calculations for the next frame. This algorithm is explained in the pseudocode below. The MATLAB code of this algorithm is attached in the appendix.

b_pos = [385,470] #Initial position of the buoy
Loop through each of the image frames in the video:

- Calculate the velocity matrices.
- Define the boundary of the region around b_pos.
- Calculate the average of the velocities of the pixels that fall within the above-defined region as Vx and Vy.
- Save Vx and Vy as displacement corresponding to current frame.
- b_pos = b_pos + [Vx Vy]
  #Moving the buoy position. This will be the buoy position for the next frame.

End of Loop.

The buoy's movement is mainly due to the movement of the camera towards it. The algorithm above helped follow the buoy's movement during the initial frames when its surrounding was still and only the camera movement was captured by the optic flow velocity vectors. During the initial frames the sea waves around the buoy appear to be slow due to their distance but as the buoy came closer the sea waves became much faster and so the average velocity vectors surrounding the buoy became faster and soon the displacement captured by the model became much greater than the actual displacement of the buoy. Soon the region even moves out of the image and at this point, we stop.

We noticed that the model captured the buoy's movement the best between the frames 75 to 150. We chose to create a motion model of length 25 frames. We split the frames between 75 to 150 into groups of 25 and used the saved displacements to track the buoy. We selected the model that had the best performance and declared it as the motion model. Using this model we were able to track the buoy. We initially declared the region (150x60) that we defined earlier as ROI to search for the exact buoy position but we found that the model was able to contain the actual position of the buoy within a region of size 100x50. So we declared an ROI of size 100x50 around the buoy position.

Getting the exact location of the buoy: The ROI extracted from the previous step was converted to grayscale. The buoy is now a white object that has pixel intensities close to 1 in a background of grey that has pixel intensities close to 0.5. Thus, we thresholded the ROI with a threshold of 0.7. We were able to separate the buoy from its background but the problem with this method was that the foam/ the air bubbles on the sea waves were also white and had intensities close to 1. So in frames where the ROI contained the buoy along with foam, we got more than one white region in the thresholded image of the ROI. To overcome this problem, we computed the centroids of all the white regions and considered the centroid with the least distance from the previous buoy location. This logic will again fail in frames where the buoy is under the waves and is not visible. Hence we include another filtering criteria on the distance. We only considered the least distances that are within 10 units from the previous buoy location. The pseudocode for this algorithm is given below and the MATLAB code for the algorithm is attached in the appendix.

Loop through each of the image frames:

- Take the grayscale version of the ROI from the image.
- Threshold the ROI – Convert all the pixels that have intensity value above 0.7 as 1 and others as 0.
- Compute the centroids for all the white regions.
- Calculate the distance of each centroid from the previous buoy location and consider the centroid with the least distance.
- Check if the least distance is less than 10 units:
  - If yes then mark the buoy position
  - Else ignore

End of Loop.

Camera Calibration: In this part, we find the intrinsic parameters of the camera. This will be used later in the distance estimation part of the algorithm. We used the calibration images provided in the Camera Calibrator app under the Image Processing and Computer Vision toolbox of

MATLAB and this app returned the parameters of the camera as a *cameraParameters* object. This is shown in Fig.4.

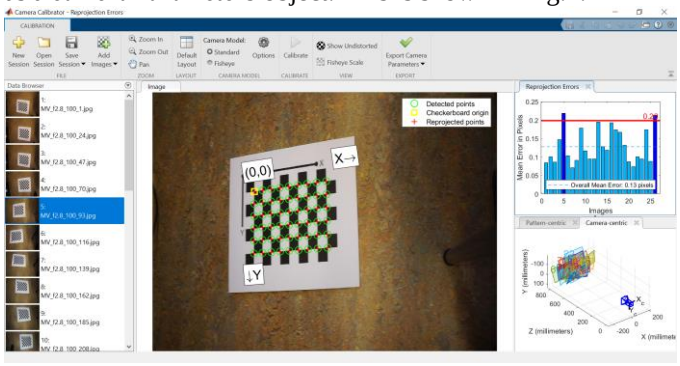

Fig.4. Camera Calibrator app

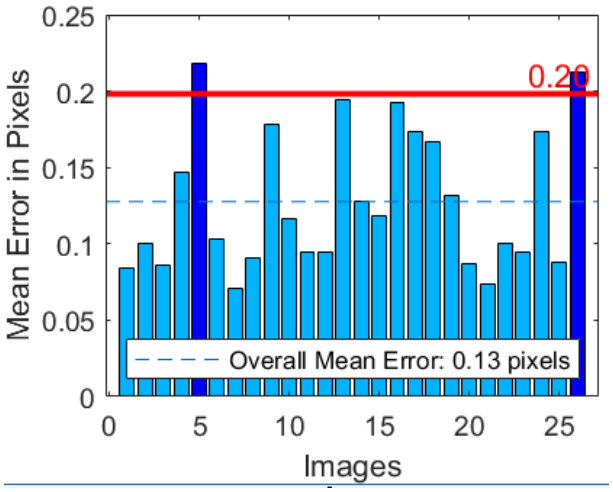

Fig. 5. Reprojection Errors graph.

The Reprojection errors graph shows two outliers – Image 5 and 25. These images were removed and the camera was recalibrated. The mean error reduced from 0.13 pixels to 0.12 pixels and the reprojection errors graph in Fig.6 was obtained. The calibration included 2 radial distortion coefficients and the skew and tangential distortion parameters were assumed to be zero. Adding skew, tangential or one more radial distortion parameter did not do any good on the overall mean error so it was kept zero.

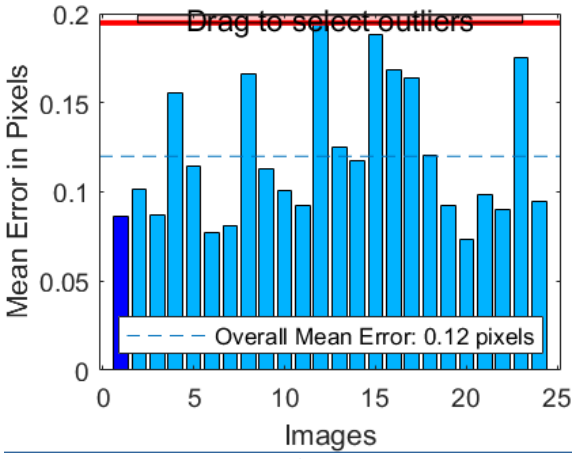

Fig.6. Reprojection errors graph after removing the outliers

Estimating the distance of the buoy from the camera: In this section, we used the height of the camera from the sea level – 2.5m, the spherical model of Earth and the buoy position in the image frames of the video to estimate the distance of the buoy from the camera.

For the first frame with the buoy, we use the pixel location of the horizon and the buoy location to compute the distance. The horizon spans from left to right of the image but we need to select one pixel location for our calculations. For this, we calculated the horizon pixel that was in line with the buoy location and the principal points.

Fig.7 shows the geometry of the method. δ (delta) is the angle of dip from the horizon. γ (gamma) is the angle between the buoy location and the horizon. β (beta) is the angle subtended by the buoy with the vertical. ***h*** is the height of the camera from the sea level. R is the radius of Earth (6.4×10^6 m). d is the distance that must be calculated [2].

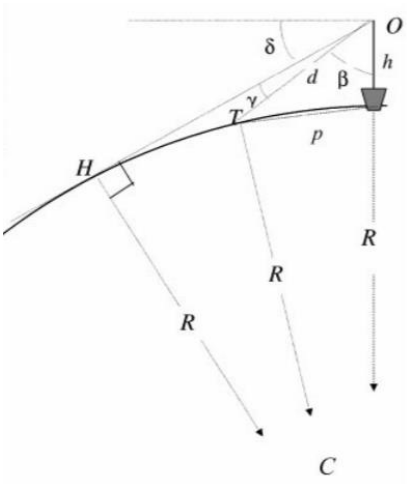

Fig.7. The geometry of the Spherical Earth model used to estimate distance.

The position of a given pixel in the camera coordinates could be obtained by dividing (or multiplying the inverse) the pixel locations by the intrinsic matrix of the camera. Further, the angle subtended by this pixel at the camera can be calculated by finding the angle between the z-axis (0,0,1) and the camera coordinates. Using this method we calculated the angles made by the horizon pixel and the buoy position with the optical axis of the camera. The difference between these angles was taken as γ (gamma). δ (delta) and β (beta) were calculated using the equations below.

$$\beta = 90 - \gamma - \delta \quad (1)$$

$$\delta = \cos^{-1}(R/(R+h)) \quad (2)$$

The distance d was calculated as follows. The equation was obtained by applying the Law of sines on the triangle ΔOTC.

$$d = \frac{\sin\left[180 - \beta - \left(180 - \sin^{-1}\left(\frac{(R+h)\times \sin\beta}{R}\right)\right)\right]}{\sin\beta} \times R \quad (3)$$

From the second frame onward instead of using the horizon pixel, we took γ (gamma) as the difference between the angles made by the current buoy position pixel and the previous buoy position pixels. δ (delta) was calculated by added the previous γ (gamma) value.

$$\delta_{new} = \delta_{old} + \gamma_{old} \quad (4)$$

The pseudocode explaining the algorithm is given below and the MATLAB code could be found in the appendix.

Loop through the image frames of the video:

- Check if it is the first frame with a buoy location:
  - If yes:
    - Calculate horizon point.
    - point1 = horizon point
    - point2 = buoy location
    - $\delta = \cos^{-1}(R/(R+h))$
  - If no:
    - point 1 = previous buoy location
    - point 2 = current buoy location
    - $\delta = \delta + \gamma$
- $\beta = 90 - \gamma - \delta$
- Calculate the distance d according to (3).

End the Loop.

III. Results

Fig.8. An image frame from the output video.

The result is a video that contains the ROI as a yellow box, the buoy circled in red and the calculated distance written in a white box with red font colour. The Fig.8 shows the first image frame where the buoy was located. The distance calculated is around 120.9 m. The image here is not very clear. The output video is uploaded as a separate file and you could see a much clearer view of the results there.

The ROI at all times encloses the buoy and the red circle appears around the buoy whenever it is found. The distance starts with 120.9 m and reduces to 62.3 m.

IV. Discussion

There was no way to measure the performance of our distance estimation algorithm. The difference between the distance calculated in the first frame and at the last frame is about 58m but the buoy does look like it has moved that much distance. So, we are a little doubtful on this aspect of the results.

The ROI always encloses the buoy and does not seem to have any problem. The circle encircling the buoy also seems to work well except in a frame or two where the buoy is visible but the algorithm does not detect it. This might be the effect of the threshold value (0.7) that we have chosen. But reducing the threshold value will give rise to complications in other aspects so we decided to not disturb the balance. In Fig.9 we have shown one such frame where the buoy is visible but the algorithm has not detected it.

Fig. 9. A frame where the buoy is visible but was not detected by the algorithm.

V. Conclusion

The buoy tracking and detection results were good and hence this algorithm could be further enhanced and used to track an actual ‘man overboard’. The distance estimation algorithm uses a number of approximations such as assuming the Earth as a sphere and approximating the radius of Earth etc. But we believe it could still be used to get a rough estimate of the distance of the object or a person being tracked and it will come in handy while planning the rescue operation.

APPENDIX A

MATLAB SCRIPT FOR VIDEO STABILIZATION

```
%% Video Stabilization
% This example shows how to remove the effect of camera motion from a video stream.

%   Copyright 2006-2014 The MathWorks, Inc.

%% Introduction
% In this example we first define the target to track. We also establish a dynamic search
% region, whose position is determined by the last known target location.
% We then search for the target only within this search region, which
% reduces the number of computations required to find the target. In each
% subsequent video frame, we determine how much the target has moved
% relative to the previous frame. We use this information to remove
% unwanted translational camera motions and generate a stabilized video.

%% Initialization
% Create a System object(TM) to read video from a multimedia file. We set the
% output to be of intensity only video.

% Input video file which needs to be stabilized.
filename = 'MAH01462.MP4';
hVideoSource = VideoReader(filename);
%v = VideoWriter('newfile');
%v.FrameRate = hVideoSource.FrameRate;

%%
% Create a template matcher System object to compute the location of the
% best match of the target in the video frame. We use this location to find
% translation between successive video frames.
hTM = vision.TemplateMatcher('ROIInputPort', true,'BestMatchNeighborhoodOutputPort', true);

%%
% Create a System object to display the original video and the stabilized
% video.
hVideoOut = vision.VideoPlayer('Name', 'Video Stabilization');
hVideoOut.Position(1) = round(0.4*hVideoOut.Position(1));
hVideoOut.Position(2) = round(1.5*(hVideoOut.Position(2)));
hVideoOut.Position(3:4) = [650 350];

%%
% Here we initialize some variables used in the processing loop.
pos.template_orig = [550 460]; % [x y] upper left corner
pos.template_size = [300 40];   % [width height]
pos.search_border = [15 5];% max horizontal and vertical displacement
pos.template_center = floor((pos.template_size-1)/2);
pos.template_center_pos = (pos.template_orig + pos.template_center - 1);
W = 1440; % Width in pixels
H = 1080; % Height in pixels
sz = [1440 1080];
TargetRowIndices = pos.template_orig(2)-1:pos.template_orig(2)+pos.template_size(2)-2;
TargetColIndices = pos.template_orig(1)-1:pos.template_orig(1)+pos.template_size(1)-2;
SearchRegion = pos.template_orig - pos.search_border - 1;
Offset = [0 0];
Target = zeros(18,22);
firstTime = true;

%% Stream Processing Loop
% This is the main processing loop which uses the objects we instantiated
% above to stabilize the input video.
%open(v);
frame = 1;
while hasFrame(hVideoSource)
    inputrgb = im2double(readFrame(hVideoSource));
    input = rgb2gray(inputrgb);

    % Find the location of Target in the input video frame
    if firstTime
      Idx = int32(pos.template_center_pos);
      MotionVector = [0 0];
      firstTime = false;
    else
      IdxPrev = Idx;
```

```
      ROI = [SearchRegion, pos.template_size+2*pos.search_border];
      Idx = hTM(input,Target,ROI);

      MotionVector = double(Idx-IdxPrev);
    end

    [Offset, SearchRegion] = updatesearch(sz, MotionVector,SearchRegion, Offset, pos);

    % Translate video frame to offset the camera motion
    Stabilized = imtranslate(inputrgb,imref2d(size(input)),Offset);

    Target = Stabilized(TargetRowIndices, TargetColIndices);

    TargetRect = [pos.template_orig-Offset, pos.template_size];
    SearchRegionRect = [SearchRegion, pos.template_size + 2*pos.search_border];

    % Draw rectangles on input to show target and search region
    inputrgb = insertShape(inputrgb, 'Rectangle', [TargetRect; SearchRegionRect],'Color', 'white');
    % Display video
    hVideoOut([inputrgb Stabilized]);
    %writeVideo(v,Stabilized);
    if frame == 28
        break
    end
    frame = frame+1;
end

%% Release
% Here you call the release method on the objects to close any open files
% and devices.
%close(v);

%% Conclusion
% Using the Computer Vision Toolbox(TM) functionality from
% MATLAB(R) command line it is easy to implement complex systems like video
% stabilization.

%% Appendix
% The following helper function is used in this example.
%
% * <matlab:edit('updatesearch.m') updatesearch.m>

% Function to update Search Region for SAD and Offset for Translate

function [Offset, SearchRegion] = updatesearch(sz, MotionVector, SearchRegion, Offset, pos)

  % check bounds
  A_i = Offset - MotionVector;
  AbsTemplate = pos.template_orig - A_i;
  SearchTopLeft = AbsTemplate - pos.search_border;
  SearchBottomRight = SearchTopLeft + (pos.template_size + 2*pos.search_border);

  inbounds = all([(SearchTopLeft >= [1 1]) (SearchBottomRight <= fliplr(sz))]);

  if inbounds
      Mv_out = MotionVector;
  else
      Mv_out = [0 0];
  end

  Offset = Offset - Mv_out;
  SearchRegion = SearchRegion + Mv_out;

end % function updatesearch
```

MATLAB CODE FOR RESIZING THE VIDEO

```
close all
filename = 'newfile.avi';
hVideoSrc = VideoReader('newfile.avi');
hVideoOut = vision.VideoPlayer();
v = VideoWriter('newfile_crop');
v.FrameRate = hVideoSource.FrameRate;
i=1;
open(v)
while hasFrame(hVideoSrc)
```

```
    % Read in new frame
    img = readFrame(hVideoSrc);
    J = img(65:1080,275:1440,:);
    hVideoOut(J)
    writeVideo(v,J);
    i=i+1;
end
close(v)
```

MATLAB CODE FOR MOTION MODEL

```
close all
filename = 'newfile_crop.avi';
hVideoSrc = VideoReader(filename);
NumFrames = hVideoSrc.NumFrames;
hVideoOut = vision.VideoPlayer();
Vx = zeros(NumFrames,1);
Vy = zeros(NumFrames,1);
opticFlow = opticalFlowFarneback('NumPyramidLevels',4,'NeighborhoodSize',7,'FilterSize',20);
imagePoints = [385,470];
frame = 1;
while hasFrame(hVideoSrc)
    % Read in new frame
    img = rgb2gray(im2single(readFrame(hVideoSrc)));
    flow = estimateFlow(opticFlow,img);
    boundary = [round(imagePoints - [75 30]);round(imagePoints + [75 30])];
    Vx(frame) = (sum(sum(flow.Vx(boundary(1,1):boundary(2,1),boundary(1,2):boundary(2,2)))))/(150*60);
    Vy(frame) = (sum(sum(flow.Vy(boundary(1,1):boundary(2,1),boundary(1,2):boundary(2,2)))))/(150*60);
    imagePoints = imagePoints + [Vx(frame) Vy(frame)];
    if (imagePoints(1)+75)>size(img,1)||(imagePoints(2)+30)>size(img,2)
        break
    end
    points = [imagePoints;boundary];
    img_p = insertMarker(img,points,'+','color','yellow');
    img_p = insertShape(img_p,'Rectangle', [boundary(1,1) boundary(1,2) 150 60]);
    frame=frame+1;
    hVideoOut(img_p)
end
```

MATLAB CODE FOR BUOY POSITION DETECTION AND DISTANCE CALCULATION

```
close all
filename = 'newfile_crop.avi';
hVideoSrc = VideoReader(filename);
NumFrames = hVideoSrc.NumFrames;
%v = VideoWriter('detected_distance');
hVideoOut = vision.VideoPlayer();
imagePoints = [385,470];
skyline = [0 500];
Motion = [Vx_a(126:150,1) Vy_a(76:100,1)]; %After trails
i = 1;
p_loc = [50,25];
buoy_loc = [0 0];
R = 6.4 * 10^6;
h = 2.5;
buoy_distance = 0;
buoy_first_frame = false;
%open(v)
while hasFrame(hVideoSrc)
    % Read in new frame
    img_RGB = readFrame(hVideoSrc);
    img = rgb2gray(im2single(img_RGB));
    imagePoints = imagePoints + Motion(i,:);
    img_p = insertShape(img_RGB,'Rectangle', [(round(imagePoints(1))-50) (round(imagePoints(2))-25) 100 ...
50]);
    ROI = img((round(imagePoints(2))-25):(round(imagePoints(2))+25),(round(imagePoints(1))- ...
50):(round(imagePoints(1))+50));
    ROI_t = imbinarize(ROI,0.7);
    stats = regionprops(ROI_t,'Centroid');
    if size(stats,1)~=0
        if size(stats,1)>1
            d = zeros([size(stats,1),1]);
            for j = 1:size(stats,1)
                d(j,1) = sqrt(sum((stats(j).Centroid - p_loc).^2));
            end
            [~,j] = min(d(:));
            if min(d(:)) < 10
                img_p = insertMarker(img_p,(imagePoints + stats(j).Centroid - …
[50,25]),'o','color','red','size',5);
```

```
                p_loc = stats(j).Centroid;
                buoy_loc = (imagePoints + stats(j).Centroid - [50,25]);
                if buoy_first_frame == false
                    alpha = acosd(R/(R+h));
                    m = (cameraParams.PrincipalPoint(1)-(buoy_loc(1)+274))/ ...
(cameraParams.PrincipalPoint(2)-(buoy_loc(2)+64));
                    skyline(1) = (500-cameraParams.PrincipalPoint(1)+(m*cameraParams.PrincipalPoint(2)))/m;
                    p1 = cameraParams.IntrinsicMatrix/[skyline 1];
                    theta1 = atand(norm(cross(p1,[0 0 1]))/dot(p1,[0 0 1]));
                    p2 = cameraParams.IntrinsicMatrix/([buoy_loc 1]+[274 64 0]);
                    theta2 = atand(norm(cross(p2,[0 0 1]))/dot(p2,[0 0 1]));
                    beta = theta1 - theta2;
                    gamma = 90 - alpha - beta;
                    buoy_distance = sind(180 - gamma - (180 - asind((R+h)*sind(gamma)/R)))*R/sind(gamma);
                    img_p = insertText(img_p,(buoy_loc + [10 10]),['Distance:' num2str(buoy_distance) ...
'm'],'BoxColor','white','TextColor','red');
                    buoy_first_frame = true;
                else
                    alpha = alpha + beta;
                    theta1 = theta2;
                    p2 = cameraParams.IntrinsicMatrix/([buoy_loc 1]+[274 64 0]);
                    theta2 = atand(norm(cross(p2,[0 0 1]))/dot(p2,[0 0 1]));
                    beta = theta1 - theta2;
                    gamma = 90 - alpha - beta;
                    buoy_distance = sind(180 - gamma - (180 - asind((R+h)*sind(gamma)/R)))*R/sind(gamma);
                    img_p = insertText(img_p,(buoy_loc + [10 10]),['Distance:' num2str(buoy_distance) ...
'm'],'BoxColor','white','TextColor','red');
                end
            end
        else
            d(1,1) = sqrt(sum((stats(1).Centroid - p_loc).^2));
            if d(1,1) < 10
                img_p = insertMarker(img_p,(imagePoints + stats(1).Centroid ...
[50,25]),'o','color','red','size',5);
                p_loc = stats(1).Centroid;
                buoy_loc = (imagePoints + stats(1).Centroid - [50,25]);
                if buoy_first_frame == false
                    alpha = acosd(R/(R+h));
                    m = (cameraParams.PrincipalPoint(1)-(buoy_loc(1)+274))/ ...
(cameraParams.PrincipalPoint(2)-(buoy_loc(2)+64));
                    skyline(1) = (500-cameraParams.PrincipalPoint(1)+(m*cameraParams.PrincipalPoint(2)))/m;
                    p1 = cameraParams.IntrinsicMatrix/[skyline 1];
                    theta1 = atand(norm(cross(p1,[0 0 1]))/dot(p1,[0 0 1]));
                    p2 = cameraParams.IntrinsicMatrix/([buoy_loc 1]+[274 64 0]);
                    theta2 = atand(norm(cross(p2,[0 0 1]))/dot(p2,[0 0 1]));
                    beta = theta2 - theta1;
                    gamma = 90 - alpha - beta;
                    buoy_distance = sind(180 - gamma - (180 - asind((R+h)*sind(gamma)/R)))*R/sind(gamma);
                    img_p = insertText(img_p,(buoy_loc + [10 10]),['Distance:' num2str(buoy_distance)...
'm'],'BoxColor','white','TextColor','red');
                    buoy_first_frame = true;
                else
                    alpha = alpha + beta;
                    theta1 = theta2;
                    p2 = cameraParams.IntrinsicMatrix/([buoy_loc 1]+[274 64 0]);
                    theta2 = atand(norm(cross(p2,[0 0 1]))/dot(p2,[0 0 1]));
                    beta = theta1 - theta2;
                    gamma = 90 - alpha - beta;
                    buoy_distance = sind(180 - gamma - (180 - asind((R+h)*sind(gamma)/R)))*R/sind(gamma);
                    img_p = insertText(img_p,(buoy_loc + [10 10]),['Distance:' num2str(buoy_distance)...
'm'],'BoxColor','white','TextColor','red');
                end
            end
        end
    end
    %writeVideo(v,img_p)
    hVideoOut(img_p);
    if i == 25
        i = 1;
    else
        i = i+1;
    end
end
%close(v)
```

REFERENCES

[1] de Boer, J., & Kalksma, M. (2015). Choosing between optical flow algorithms for UAV position change measurement. 12th SC@ RUG 2014-2015, 69.

[2] Gordon, J., 2001. Measuring the range to animals at sea from boats using photographic and video images. *Journal of Applied Ecology*, *38*(4), pp.879-887.