

2012

Application of Auto-tracking to the Study of Insect Body Kinematics in Maneuver Flight

Shreyas Vathul Subramanian
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all



Part of the [Mechanical Engineering Commons](#)

Repository Citation

Subramanian, Shreyas Vathul, "Application of Auto-tracking to the Study of Insect Body Kinematics in Maneuver Flight" (2012). *Browse all Theses and Dissertations*. 1091.
https://corescholar.libraries.wright.edu/etd_all/1091

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

APPLICATION OF AUTO-TRACKING TO THE STUDY OF INSECT BODY KINEMATICS IN MANEUVER FLIGHT

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Engineering

By

SHREYAS VATHUL SUBRAMANIAN
BTech., National Institute of Technology Karnataka

2012
Wright State University

WRIGHT STATE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

June 11,2011

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Shreyas Vathul Subramanian ENTITLED Application of Auto-tracking to the Study of Insect Body Kinematics in Maneuver Flight BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Engineering

Haibo Dong, Ph.D.

Thesis Director

George Huang, Ph.D., Chair

Department of Mechanical and Materials

Engineering College of Engineering and Computer

Science

Committee on Final Examination

Haibo Dong, Ph.D.

Nasser Kashou, Ph.D.

Hui Wan, Ph.D.

Andrew T. Hsu, Ph.D.
Dean, Graduate School

ABSTRACT

Subramanian, Shreyas Vathul. M.S.Egr. Mechanical Engineering Department, Wright State University, 2012. Application of Auto-tracking to the Study of Insect Body Kinematics in Maneuver Flight.

There is a need to explain the complex phenomena that underlies the seemingly effortless flight modes of the dragonfly (Infra -order *Anisoptera*). However, measuring the body kinematics during flight is labor intensive. Thus a robust system was developed that automatically tracks and quantifies the body kinematics of a dragonfly during voluntary and escape take-offs, as well as maneuvers. Ultimately, the tool, which was developed using a custom code in C++ using the open source library OpenCV (Open Computer Vision), would be used to analyze bulk samples of high speed videos providing raw images at the rate of approximately 1000 frames per second from pair-wise orthogonal positions in space. As a result, there would be a considerably large database of information which may then be used to formulate, generalize and classify standard flight strategies used. Perceptibly, there is also a need to validate the outputs of this tool by comparing it to the outputs of a manual reconstruction.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
Insect Flight.....	1
Existing Methodologies	2
Reconstruction methods.....	4
II. OBJECTIVE AND SCOPE OF THIS RESEARCH.....	5
Background and Motivation.....	5
Image Processing Strategies.....	7
Camera Projection matrix.....	7
POSIT.....	9
Haar Training.....	12
Template Matching	17
SURF.....	18
Methodology Comparison – Manual vs Automatic.....	20
Basics of Manual Tracking.....	20
Issues with raw input.....	21
Issues with Manual Tracking.....	24
Auto-Modeling via Image Processing.....	26
Mathematical Modeling of the Wings.....	31
III. AUTO-TRACKING TECHNOLOGY.....	36
Introduction	36
The Input Image	38
Processing a Single Image	40
Smoothing	40
Thresholding.....	41
Equalizing the Histogram.....	42
Morphological Operations.....	43
Edge Detection	44
Combining the Three Camera-Views	45
Contour Information	47
Constructing the Voxel Model	48
Analyzing Voxels	51
Calculating Pitch, Roll and Yaw	52
Processing the video	54
Graphical User Interface	57
Initial screen	57
Body-Wing modeling Tab.....	58

	Yaw-Pitch-Roll tab	59
	Other Tabs	60
Test Cases		61
	Test Case 1	61
	Test Case 2	64
	Test Case 3	66
	Other Cases (Plots only).....	69
	Test Case 4	69
	Test Case 5	69
	Test Case 6	70
	Test Case 7	70
	Test Case 8	71
	Test Case 9	71
	Test Case 10	72
	Test Case 11	72
	Discussion	73
IV RESULTS.....		78
	Classification.....	78
	Case Bifurcation	80
	Validation cases.....	82
	Pitch – Roll – Yaw plots	82
	Dragonfly 8 – an experiment.....	88
	Observations.....	90
	Dragonfly 8.5.....	90
	Dragonfly 8.3.....	91
	Dragonfly 8.6.....	91
	Dragonfly 8.7.....	92
	Other interesting maneuvers.....	93
	Dragonfly 11.1	93
	Dragonfly 12.6.....	93
	Conclusions	95
Summary		99
Future Work		99
Bibliography.....		101
Appendix 1 – Code		103
	Autotracking.....	103

LIST OF FIGURES

Figure	Page
1. Experimental set-up	5
2. Maya Reconstruction	6
3. Camera projection.....	8
4. Haar training intermediate output.....	15
5. Output of the template matching algorithm	18
6. Perfect matching using SURF	19
7. SURF - erroneous output	19
8. Issues with raw input	23
9. Light 'Bands' seen in the video	24
10. Artificial camera rig/set-up	25
11. <i>Left</i> -Original image (top view); <i>Right</i> -Thresholded and corrected image along with detected segments.....	28
12. Side view.....	28
13. 3D wire-frame rendering of the reconstructed body.....	29
14. Actual Left Fore-wing.....	33
15. Closed b-spline modelling the Left fore-wing	33
16. Digitized points along with the original picture	34
17. The original image with its histogram	39
18. Forcing the image to have a particular histogram	39
19. Segmentation using histograms	40

20. Image after smoothing	41
21. Image after thresholding	42
22. Image after histogram equalization	43
23. Image after eroding.....	44
24. Edge detection	44
25. Orthographic views of an object.....	45
26. Reconstruction errors while using only two views.....	46
27. Three views of the camera	46
28. Decomposing into sets of contours.....	48
29. Surface and extruded volume of one view	49
30. Two views of the reconstructed body	50
31. Voxel and pre-constructed bodies	50
32. The centroid line and orientation triangle	51
33. Trackbar controls	55
34. Displaying Euler angles in real time	56
35. Initial screen of the GUI Interface	57
36. The profile descriptor utility with instructions.....	58
37. Body-wing modeling complete	59
38. Yaw - Pitch -Roll calculation tab	60
39. Virtual marker and Utilities Tab.....	60
40. Test case 1 in time snapshots	62

41. Test case 2 in snapshots	64
42. Test case 2 in snapshots	66
43. Comparison of CG's : x-y-z vs v1-v2-v3	75
44. Demonstration of oscillating or peaking output	76
45. Other types of errors seen in output.....	76
46. Effects of smoothing on erroneous output	77
47. Case availability	79
48. Forced take-off (Left) vs. Natural take-off (Right)	79
49. Instacross magnitude from the v1 and v2 vectors shown	89
50. Instacross plot along with P-R-Y, DFLY 8.5.....	90
51. Instacross magnitude with P-R-Y angles, DFLY 8.3	91
52. Instacross magnitude with P-R-Y angles, DFLY 8.6	92
53. Instacross magnitude with P-R-Y angles, DFLY 8.7	92
54. Instacross magnitude with P-R-Y angles, DFLY 11.1	93
55. Instacross magnitude with P-R-Y angles, DFLY 8.7	94
56. Conventional aircraft vs a Dragonfly.....	96
57. Calculation of Centre of gravity and location of wing roots.....	98
58. Effect of flexible tail on Centre of Gravity	98
59. Multiple point tracking, Cicada.....	100
60. Multiple point tracking, Butterfly.....	100

LIST OF TABLES

Table	Page
1. POSIT	
POSIT algorithm outputs.....	11
2. Haar Training	
Types of testing samples	14
3. Haar Training	
‘Random’ testing samples	14
4. Haar Training	
Haar training outputs.....	17
5. Issues with Raw Input	
Effects of lighting.....	23
6. Issues with Manual Tracking	
Errors in orthographic modeling.....	24
7. Issues with Manual Tracking	
Using the artificial camera rig	26
8. Auto-Modeling via Image Processing	
Different views of the fully constructed body	30
9. Auto-Modeling via Image Processing	
Comparison of static reconstruction results.....	31
10. Test Cases	
Test case 1 - Plots and explanations.....	63
11. Test Cases	
Results of test case 2	65
12. Test Cases	
Results of test case 3	68

13. Test Cases	
Test Cases 4 and 5.....	69
14. Test Cases	
Test Cases 6 and 7.....	70
15. Test Cases	
Test Cases 8 and 9.....	71
16. Test Cases	
Test Cases 10 and 11	72
17. Test Cases	
Validation cases	82
18. Pitch-Roll-Yaw Plots	
Slow and high, set 1	83
19. Pitch-Roll-Yaw Plots	
Slow and high, set 2	84
20. Pitch-Roll-Yaw Plots	
Fast and medium, set 1	85
21. Pitch-Roll-Yaw Plots	
Fast and low, set 1	86
22. Pitch-Roll-Yaw Plots	
Fast and low, set 2.....	87

ACKNOWLEDGMENTS

I would like to thank my research advisor, Dr. Haibo Dong for his constant supervision throughout the entire period of my thesis study here at Wright State University. His guidance and timely advice has given me a sense of direction and confidence. Working with the members of Dr. Dong's Flow Simulation Research Group (FSRG) has been a great learning experience, and has given me a taste of what actual research is.

I. Introduction

Insect Flight

Any established biologist would subscribe to the tenet that true flight is shared only by insects, birds and bats. Insects are the only group of invertebrates known to have evolved flight. Flight in insects is believed to have developed more than 300 million years ago. Thus it is only natural to surmise that this is something that has been perfected over time. Insect wings are outgrowths of the insect exoskeleton, and are referred to as forewings and hindwings. The ability to fly is not determined by the number or the size of wings. Flight is one of the primary reasons that insects have survived in nature. Flight assists insects find food, locate mates and escape from predators. Flight in insects varies dramatically, from the clumsy patterns of beetles and butterflies to the acrobatic maneuvers of dragonflies and many true flies. Dragonflies are capable of astounding feats such as moving forward, backward, turning, and hovering at one position. Insects of the order Paleoptera (Example, dragonflies) have muscles inserted directly at the base of the wing. Flight in insects may also be gained by muscles which are not attached directly to the wings, but move the wings indirectly by changing the shape of the thorax. Thus the study of the body kinematics in flight may prove to be very important. The primary focus of many insect flight research teams around the world is to understand the actively

changing moment of inertia characteristics. Most of these teams use high speed photogrammetry as the main experimental tool.

Existing Methodologies

The experiments involved in the measuring of flight parameters, be it kinematics and dynamics, or flow visualization using computational fluid dynamics, involves the use of computers with high processing and graphical capabilities. Thus we may broadly classify insect study into two completely different interest groups-One, The study of low Reynolds number flyers in nature using CFD visualization and two, the study of the Kinematics and Dynamics involved in insect flight. Conceivably, both these streams are important to the study of insect flight. However, the processing time is as large as the time required to set up the cases in the afore mentioned classes of study. While the actual processing, for example CFD simulation of insects in free flight, takes up hours of computational time, the actual reconstruction of these insects in 3d is nothing less than manual labor. There have been a few attempts to automate the process of digitizing insect flight. For instance, the advances in high speed imaging studies applied to animal and fluid motion was studied by Launder et al(6). Here, pixel intensity is measured scene by scene or frame by frame, after which an accurate cross correlation set of peaks is generated. Using this information, a matrix of velocity vectors is then produced. Velocity data is obtained directly and the total body velocity (obtained as average of all velocity vectors) was reported to be quite accurate. The work of Hedrick et al(2) discusses the Discusses automation of video analysis for biomimetic systems with freely available software. The Advantage is that it does not require manipulation of the animal to be

measured. This will be one of the major requirements of the tool being developed here. 'Digitizing' of the video is achieved by manual mouse-clicks, much like the process of 'Manual Reconstruction' at the FSRG laboratory. Similar work was done by Combes et al (3) where the dramatic deflections during flight were addressed. The work of Sunada et al (9) also uses optical measurement to quantify the deformation and Motion of the wings of a Moth (*Mythimna Separata*).

A major contribution to the topic of automatic tracking of free flying insects was the research of Fontaine et al(4). The methodology was similar to capturing human motion using joints and links. This was done using two approaches-model based or direct reconstructions; the former being better when there are occlusions in the field of view. The *Drosophila* has a near cylindrical body and hence roll estimation proved to be very difficult. This is not the case in the dragonfly being studied here, and hence roll estimation is generally expected to be easier and more accurate.

Some research groups have used 3D high speed video to measure morphological parameters of the wing, for example Yenpeng Liu et al (7) from Beijing University of Aeronautics and Astronautics. A standard stereo camera set-up is used to shoot videos of the insects in flight, unlike the orthographic set up used here. The work of Simon Walker et al(10) talks about reconstruction of high resolution topologies of free flying as well as tethered insects. The author explains a method for measuring the flapping angle and torsional angle of a dragonfly wing using two sets of fringe patterns that were projected from orthogonal directions.

More specifically, Dragonfly maneuvers were studied by Hao Wang et al(12). Eight individuals were analyzed with 10 sequences each for turning flight and forward flight. A

pilot lamp induced the insect to fly across. The angle of the measured fringe is determined using a reference fringe that is predetermined. The wing base and wing tip are constructed using identifiable landmarks that are available readily. A Six DOF (Degree of Freedom) system is emulated which consists of yaw, pitch and roll as a possibility. The translation is measured with respect to the global origin, which is also predetermined. The kinematic parameters that were measured are wingbeat frequency, flapping angle, angle of attack (α), torsional angle, and camber deformation amongst others. Some of these parameters will be measured in the current work.

Reconstruction methods

As discussed by Wilmott et al (14), a common problem in biomechanical studies is the need to reconstruct three-dimensional motions from two-dimensional film images. This may be done using single images, or stereo images. Some experimental setups may even use three orthogonal cameras, or multiple randomly positioned cameras to shoot high speed images. The work mainly discusses the strips and planes method and compares it to the symmetry and landmark based procedures. The Symmetry method and the Planes method form one class of reconstruction methods, as they may be used together. The Strips and Landmarks methods were developed later and are relatively new strategies. Wilmott et al (14) also compare all the afore mentioned methods, when applied to the same experimental set up.

II. Objective and Scope of This Research

Background and Motivation

The manual reconstruction method used at the Flow Simulation Research Group (FSRG) is standard for any insect (or inanimate object) being filmed. The steps of involved are simple, but time intensive. These steps are discussed below:

1. Three High-speed Cameras aligned along the principle coordinate axes are used. The capturing system is a *Photron Fastcam SA3 60k* model (see fig. 1). The system is capable of capturing a maximum of 1000 black and white frames per second at full resolution (1024×1024) with a global shutter speed of 2 us. With 4 GB of onboard storage, this stand-alone, network-attached camera system allows for a maximum continuous record time, at optimum settings, of 2.726 seconds.

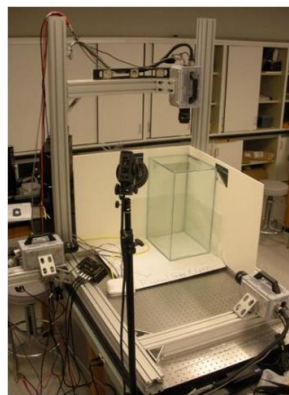


Figure 1: Experimental set-up

2. A similar scene is created in a 3D modeling software called Autodesk Maya, with the backgrounds set as the images from each camera. Then the body and the wings are tracked manually by eyeball, and the control points are digitized. This results in the animation of the insect, complete with moving, rotating and deforming set of body and wings. A snapshot of this animation sequence is shown below in fig. 2.

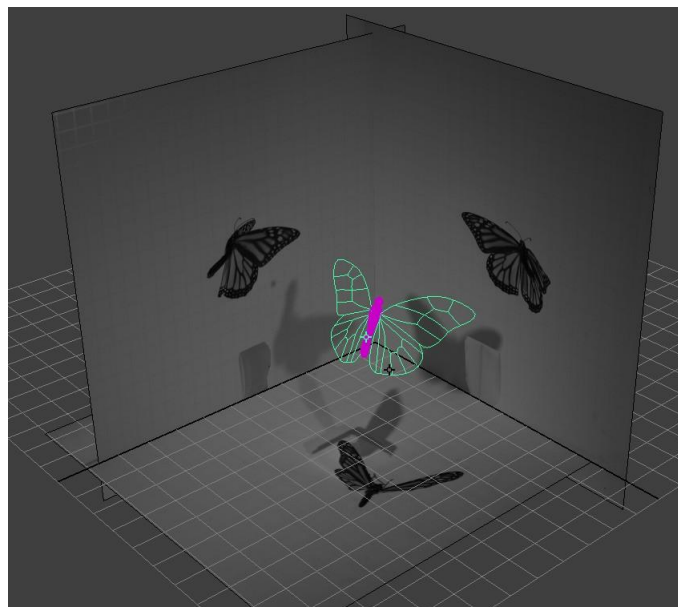


Figure 2 : Maya Reconstruction

Once the geometric information is available, it may be used to calculate the flow around the bodies or the kinematics involved in a flight sequence. Thus as we can see, the actual reconstruction is the ‘bottleneck’ that slows down the process. The motivation for the development of auto-tracking software is precisely this. Several questions remained unanswered; is it possible to create software that can minimize the time required for

processing? And if at all it is possible, will the software's output be comparable to the quality of output delivered by manual/human tracking. The use of OpenCV (Open Computer Vision) along with the C++ programming language is seen in a vast range of image processing applications. It is chosen over MATLAB because it more robust, and more flexible. There are several ways to tackle this problem, and these will be discussed in the next section.

Image Processing Strategies

Camera Projection matrix

The idea of 3D reconstruction from images is not new. However, we must know that the experimental set up described earlier is special. So, strategies that exist in other research contributed to learning, but did not contribute directly to the solution of this problem statement. For example, the research of Walker et al(10) talks about an early attempt at reconstructing tethered and free-flying insects automatically. Liu et al's discussion on pose estimation was important to this research, although the applications are completely different(15). Essentially, the problem of Pose estimation is the calculation of the value a single points position in 3D space from a 2D image. The additional information required is contained in a Projection Matrix. A projection Matrix contains information about the camera (or virtual camera) with which the image was shot (or is imagined to be shot with). The camera matrix gives us the information about the focus, the distortion vectors

and the sensor dimensions. This helps imagine how exactly the points were projected on the plane that we see the image in.

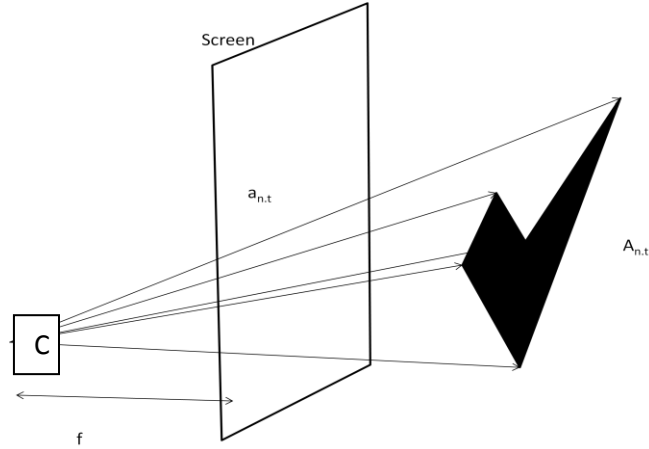


Figure 3 : Camera projection

The pin-hole camera projection model is shown in fig. 3. The point ‘C’ represents the pinhole camera, the vector ‘ A_n ’ denotes the set of all 3d model points and ‘ a_n ’ denotes the set of all corresponding points that are projected onto the screen. Liu et al’s method to calculate the values of the projection matrix P, is using error functions. Shape alignment and model based Pose estimation is done using three points. This gives a maximum of four solutions. Simply said, length of the vectors forms the focus point, along with the importance of geometric transformations. L_n is the vector that contains lengths of each model point from the camera. ‘P’ is the centre of perspective, that is, the position of the camera.

$$u_{n,t} = \frac{P(a_{n,t}, f)}{|P(a_{n,t}, f)|} PA_{n,t} = l_{n,t} u_{n,t}$$

Where

$$u_{n,t} = \frac{P(a_{n,t}, f)}{|P(a_{n,t}, f)|}$$

The algorithm has the information of both model points and image points at time $t=0$. Using Gaussian elimination and window based correlation, the point's in the next time step are estimated. This is done by minimizing the error function.

$$E^k(I, j, k) = \{ (l_{i,t}^k)^2 + (l_{j,t}^k)^2 - 2(l_{i,t}^k \times l_{j,t}^k) (u_{i,t}^k \cdot u_{j,t}^k) \}$$

Thus as we can imagine, for a model with four points, there are 6 error functions to minimize. For a model insect however, the algorithm described is much more complicated.

POSIT

POSIT (Pose estimation from Iterations) is another algorithm that can be made to good use for this class of problems. Davis et al (17) discuss the development of a simple algorithm ("25 lines of code") which uses POS (Pose from orthography and scaling) in an iteration loop to find better scaled orthographic projections of feature points. This may be used for planar as well as non planar points with slight modifications. OpenCV has its own implementation of POSIT, which was used to test the usefulness of such an algorithm to the problem at hand. Let us say that the dragonfly's body could be represented by four points on the body. It is possible to then reconstruct all points by just knowing the locations of these points on the image.

The code calculates the Euler angles of an object of known geometry once we input the model coordinates. To explain what's happening in a more comprehensive way, the inputs to the code is the actual 3D coordinates. Based on a picture, the corresponding points on the picture and a few other factors like the camera projection matrix and the intrinsic (focal lengths etc) we can estimate the 3D pose of the object that caused this kind of a projection in space. A series of 40 orientations of the cube were tested. Now, to check if these predicted values are right, the code uses the estimate and re-projects each point into image space. Thus, there is an error; however this level of error is expected due to several factors. The rotation and translation matrices are also outputted along with the Euler angles.

The code needs at least three non-coplanar points to judge the pose of the objects. This may be used to get the Euler angles of the insect. The idea of transferring this to the image tracking code for an insect is not trivial, as the geometry tested here was a simple cube of side 15 units. A complex geometry like the dragonfly body may not be as simple. But identifying the corresponding critical points on a body continuously and monitoring their progress may help. The output of the code is shown below. The black points are the reprojected values. The red circles are the actual corners of the cube. The red, green and blue vectors are the sides of the cube that are measured. The output is shown with the corresponding output, which is the rotation matrix, translation matrix and Euler angles.

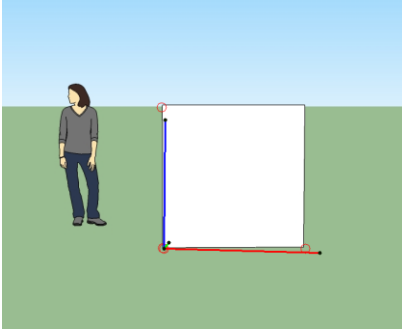
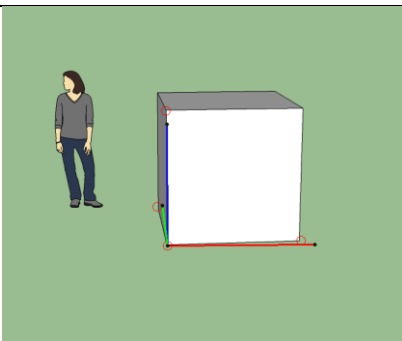
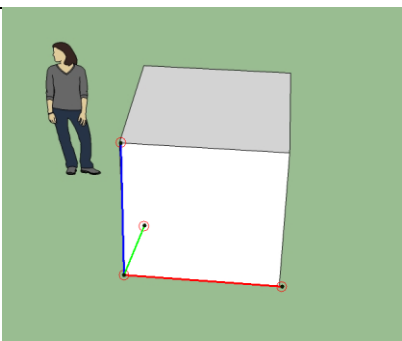
Sample	Output
	<pre> !! Sample : 5 !! -- SOURCE MODEL POINTS -- 0, 0, 0 0, 0, 15 15, 0, 0 0, 15, 0 -- SOURCE IMAGE POINTS -- 755, 658 767, 471 817, 532 550, 600 -- ESTIMATED ROTATION 0.788854 ! -0.614543 ! 0.0067481 -0.30962 ! 0.154431 ! -0.938236 0.575545 ! 0.738042 ! -0.0684513 -- ESTIMATED TRANSLATION 55.8113 ! 48.6408 ! 73.9222 euler angles: 95.2989,37.8291,-23.0791 </pre>
	<pre> !! Sample : 1 !! -- SOURCE MODEL POINTS -- 0, 0, 0 0, 0, 15 15, 0, 0 0, 15, 0 -- SOURCE IMAGE POINTS -- 597, 583 593, 348 832, 583 595, 582 -- ESTIMATED ROTATION 0.842272 ! 0.461498 ! 0.262964 -0.261996 ! 0.352536 ! -0.898374 -0.507302 ! 0.692272 ! 0.419604 -- ESTIMATED TRANSLATION 44.2097 ! 43.1729 ! 74.0531 euler angles: 58.7788,-32.0745,-18.0105 </pre>
	<pre> !! Sample : 2 !! -- SOURCE MODEL POINTS -- 0, 0, 0 0, 0, 15 15, 0, 0 0, 15, 0 -- SOURCE IMAGE POINTS -- 607, 599 604, 367 837, 590 589, 532 -- ESTIMATED ROTATION 0.8859 ! 0.447638 ! 0.121662 -0.270347 ! 0.0900345 ! -0.958544 -0.440035 ! 0.816283 ! 0.200779 -- ESTIMATED TRANSLATION 45.2407 ! 44.6444 ! 74.5316 euler angles: 76.1814,-27.6307,-17.767 </pre>
	<pre> !! Sample : 39 !! -- SOURCE MODEL POINTS -- 0, 0, 0 0, 0, 15 15, 0, 0 0, 15, 0 -- SOURCE IMAGE POINTS -- 553, 624 548, 418 799, 642 584, 548 -- ESTIMATED ROTATION 0.797306 ! 0.598828 ! 0.075549 -0.276183 ! 0.0218936 ! -0.960856 -0.577042 ! 0.745231 ! 0.182842 -- ESTIMATED TRANSLATION 42.1595 ! 47.5724 ! 76.2378 euler angles: 76.2148,-36.9435,-20.2161 </pre>

Table 1 : POSIT algorithm outputs

We can see that the algorithm works well for a standard cube. When applied to an insect however, there are many more complicated issues to be addressed. The insect's body is deformable, albeit in a limited way. The wings on the other hand may rotate, twist and deform randomly.

Machine learning is one method that may be used to detect special features. Currently these techniques are very popular for face tracking and human tracking. This leads us to Haar training.

Haar Training

Haar-like features owe their name to their similarity with Haar wavelets and are used in face detection algorithms (16). Viola and Jones (18) used the idea and developed the so called haar-like features. This method involves categorizing the subsections of an image using the intensities in its rectangular neighborhood. The most popular use of this method is for face recognition. For example, the eyes are tracked by recognizing the fact that generally the region around the eyes is darker than the forehead and cheek. So, we force the code to learn how certain features 'look' in the neighborhood of other non-objects. The classifiers are readily available for face detection, and specifically, even for eye, nose and mouth detection etc. But this kind of a database is not available for insect flight study. To study the usefulness of this method, 3000 white samples (with a white background), 5000 samples with a white-to-gray gradient background, and 3500 color samples (with random real world backgrounds) were used to create two databases. These

images were superimposed with randomly oriented 3D images of one particular wing (either one of the following – Right forewing, Right hind wing, Left forewing and Left hind wing). The openCV implementations of haar-training and haar-classifying were used to display results of the training. The motivation of this exercise is to judge if the random orientations of the body and wing may be ascertained accurately using this method. The sample size of 20 X 20 achieved the maximum possible hit rate, as mentioned in literature (Kuranov et al(19)). Further, if the number of stages to be trained is 20, we can expect a false alarm rate of $0.5^{20} = 9.6 \text{ e-}07$ and a hit rate of $0.999^{20} = 0.98$ or 98%. In some trials, the training finished at an intermediate stage if at all the minimum hit rate was exceeded. A few other occasions saw the training utility fail. A set of 3500 images were used for training. These were the set of negative images. Simply put, the more random the images are, the better the learning process. Several other subsets were used, such as 100, 500, 1000 etc, and I can conclude here that the more images we use, the better the results. Now, since the actual set up used in the high speed photogrammetry lab here has a white background for all three views, I also tried the training with 3000(case 1) and 5000(case 2) white to grey images. However the results were much better using the 3000 random background images clicked with the lab-provided camera outdoors. Examples of training images used are shown below:-

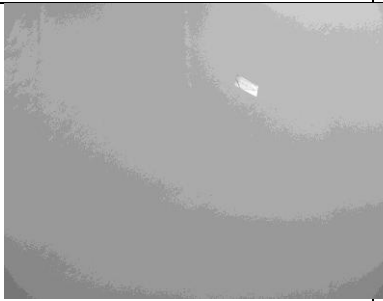


A 'white' testing sample	A 'gray' testing sample	A 'random' testing sample
		

Table 2 : Types of testing samples

Other 'random' testing samples are shown below :-

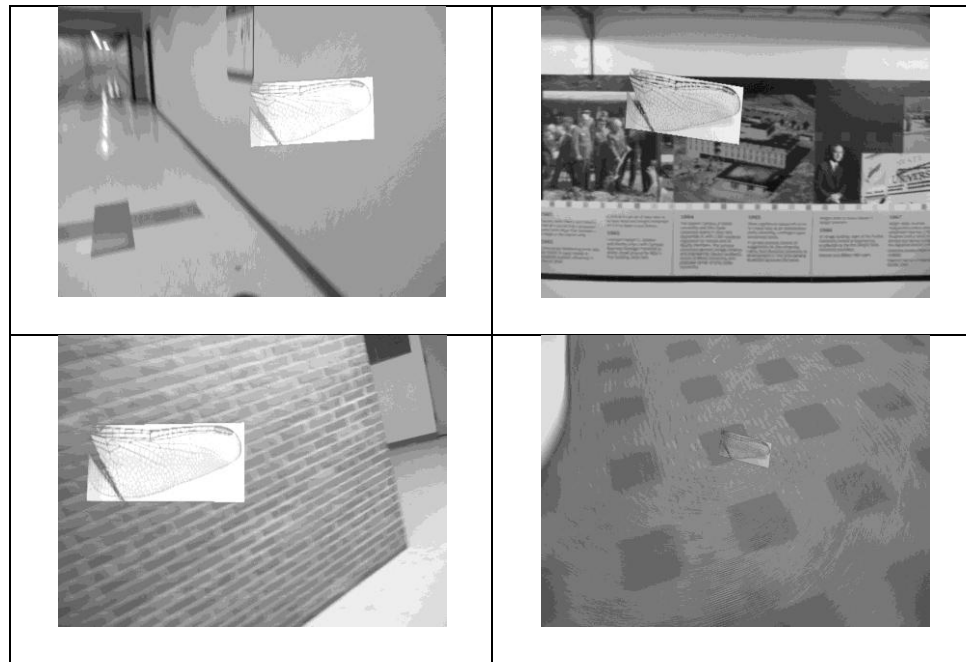


Table 3 : 'Random' testing samples

On average, a case run with about 3000 negative images completed all stages of training successfully with the hardware provided in 3 to 4 days. Although the 'training' period takes a long time, the actual identification or 'classification' is almost instantaneous, which is the biggest advantage of this method.

An intermediate stage output describes the state at which the training is. At stage 9, the output is as follows :-

```

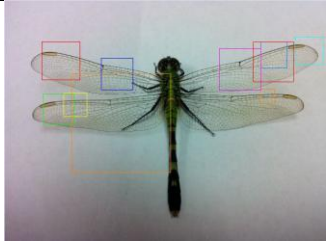
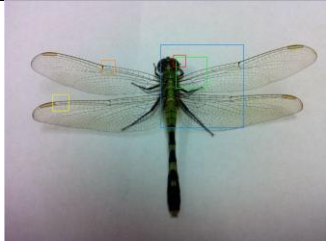
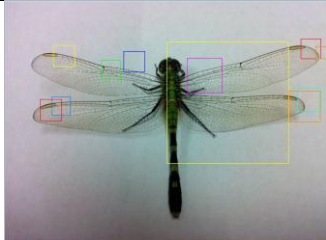
Parent node: 8
*** 1 cluster ***
POS: 2548 2563 0.994147
NEG: 1098 0.00264703
BACKGROUND PROCESSING TIME: 5.69
Precalculation time: 1.06
+-----+-----+-----+-----+-----+
| N | %SMP | F | ST. THR | HR | FA | EXP. ERR |
+-----+-----+-----+-----+-----+
| 1 | 100% | - | -0.671060 | 1.000000 | 1.000000 | 0.199397 |
+-----+-----+-----+-----+-----+
| 2 | 100% | - | -1.237876 | 1.000000 | 1.000000 | 0.176906 |
+-----+-----+-----+-----+-----+
| 3 | 100% | - | -1.820567 | 1.000000 | 1.000000 | 0.182666 |
+-----+-----+-----+-----+-----+
| 4 | 79% | - | -1.854327 | 0.999215 | 0.850638 | 0.150027 |
+-----+-----+-----+-----+-----+
| 5 | 78% | - | -1.972907 | 0.999215 | 0.871585 | 0.119857 |
+-----+-----+-----+-----+-----+
| 6 | 82% | - | -2.115036 | 0.999608 | 0.776867 | 0.090236 |
+-----+-----+-----+-----+-----+
| 7 | 72% | - | -1.883629 | 0.999215 | 0.624772 | 0.095996 |
+-----+-----+-----+-----+-----+
| 8 | 73% | - | -1.860241 | 0.999215 | 0.629326 | 0.065551 |
+-----+-----+-----+-----+-----+
| 9 | 72% | - | -1.916979 | 0.999215 | 0.481785 | 0.060340 |
+-----+-----+-----+-----+-----+
Stage training time: 2071.69
Number of used features: 18
Parent node: 8
Chosen number of splits: 0
Total number of splits: 0
Tree Classifier
Stage
+-----+-----+-----+-----+-----+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+-----+-----+-----+-----+-----+
0---1---2---3---4---5---6---7---8---9

```

Figure 4 : Haar training intermediate output

The columns above are indicative of the parameters that change throughout the training stages. Here, an intermediate 9th stage of training is shown. ‘N’ is the current feature for this cascade. ‘%SMP’ is percentage of samples used. ‘F’ is a parameter which is defines if symmetry is specified. ‘ST. THR’ is the stage threshold, ‘HR’ is the Hit rate based on stage threshold and ‘FA’ represents the False alarm rate based on stage threshold. The time taken for each stage of training is denoted in number of seconds, and the number of

features used is also output. The current stage of the classifier is given in a neat diagram format. To give you an example, 3000 negative images on the machine provided (Intel Pentium ® processor, 3.4 GHz, with 2 GB RAM and a 64 bit Windows operating system) takes about 3 days to complete. The all-white or grey images did not show good results as seen below. The results were erroneous, and even though the hit rate mentioned was achieved, the outputs were not useful. The results obtained from the right hind wing case are summarized below.

No. Of Samples	Case	Output
8	White	 A photograph of a dragonfly with several colored bounding boxes (red, blue, yellow, green) overlaid on its wings and body, indicating object detection results.
50	Gray	 A photograph of a dragonfly with several colored bounding boxes (red, blue, yellow, green) overlaid on its wings and body, indicating object detection results.
100	Gray	 A photograph of a dragonfly with several colored bounding boxes (red, blue, yellow, green) overlaid on its wings and body, indicating object detection results.



3000	Random	
5000	Random	

Table 4 : Haar training outputs

As we can see from the results, more number of random images gives better results. However, the result is never actually an accurate one. The probability that the particular sub-image is detected accurately increases. Thus this method may not be very useful. Two other methods of feature specific detection that have no relation to neural networks and training as seen above are discussed.

Template Matching

Template matching is a very rapid implementation that uses the same concept of matching a sub-image that is on a super-image. As shown below, zero errors were seen in the openCV implementation of template matching. However, the downside is that the method is very idealized. The program moves the sub-image block all over the super-image and tries to find an exact match. A fore wing obtained from the same super image of a dragonfly in top view, say, will obviously give a very accurate result. On the other

hand, the left forewing of one particular dragonfly will not yield a good result when compared with that of another. Nevertheless, the implementation of this method is perfect when the inputs are relevant.

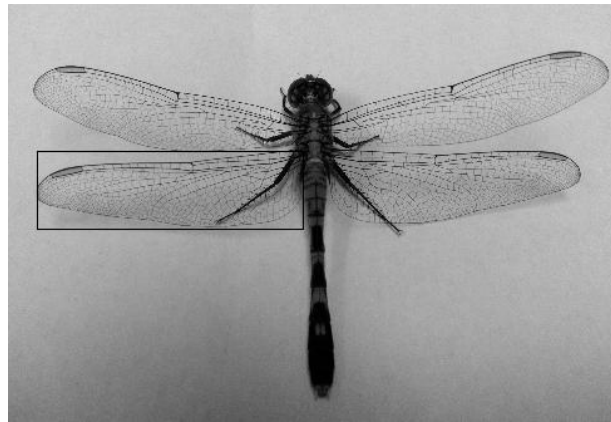


Figure 5 : Output of the template matching algorithm

SURF

Another method that identifies the “best” features to track on say, a wing and then match those to another photo using a nearest neighbor approach is the SURF method. SURF (Speeded Up Robust Features) uses important critical pixels and tries to match the sub-image to the super-image. However, the drawbacks of this method are exactly the same as the case described above(Template matching). The red circles are the features on the wing that are used. The white lines denotes a matching correlation. The white box is the estimated position of the object. As seen in fig. 6, the code does a great job when the left hind wing of a particular dragonfly is analyzed for features (see the red circles in the window named “object”) and successfully identifies the wing on the super-image by matching the features.

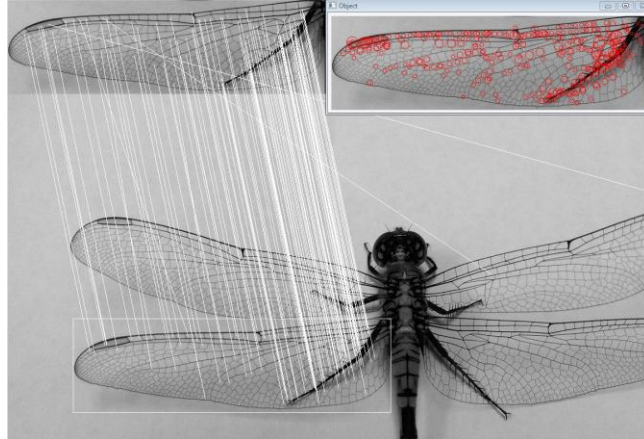


Figure 6 : Perfect matching using SURF

Shown below in fig. 7 is a test sample with the original Left Hind wing of the dragonfly, tested with another completely different dragonfly super-image. To add some more difficulties to this test of robustness of the code, a poor quality image with poor lighting is used. As shown above, the code fails miserably. This is because the wing of a dragonfly is like a fingerprint; even though you may be able to find a few matching features, it is not possible to successfully report a hit.

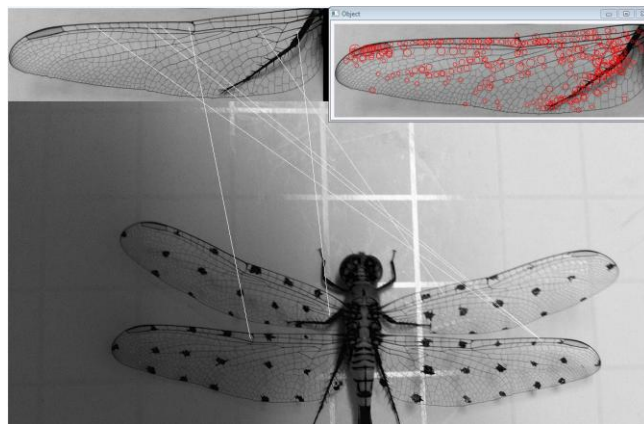


Figure 7 : SURF - erroneous output

Methodology Comparison – Manual vs Automatic

In order to compare the results of a successful automatic reconstruction, it may be useful to follow the same methodology that is followed when attempting to track or reconstruct the dragonfly flight video manually.

Basics of Manual Tracking

As mentioned earlier, this is done using a 3D modeling and animating tool, specifically Autodesk Maya(20). The general steps to achieve a manual reconstruction are outlined below:-

- 1) The individual image planes are created perpendicular to each other ('Orthogonal')
- 2) The actual images are added as dynamic 'Lambert' materials that change at every time frame.
- 3) The body and the wings of the given insect are created to form the static model.
- 4) Critically important keyframes are chosen to animate the body first.
- 5) Once the body is animated, the wings are attached to the body at one single point as a 6 degree of freedom joint.
- 6) The wings are then animated with the help of control points to give a full reconstruction.

As we may imagine, this would also be the outline to follow while attempting to track the dragonfly automatically. However, this is not true. The brain's processing capabilities to automatically complete hidden images and objects, to interpolate and extrapolate, and to automatically ignore noise cannot be underestimated. These are some of the assumptions that an auto-tracking code cannot take for granted. Manual tracking, however, is very laborious. A 'Full' reconstruction of a dragonfly that completes two wing-beats or strokes may take up to 4 days. This is also one of the factors that imply that an auto-tracking solution may prove to be very useful in the future.

Issues with raw input

At first, following the same strategies followed by a manual tracker, but putting it in terms of a computer code, may sound trivial. However, there are several issues that need to be addressed, some of which are:-

- 1) Experimental set up
- 2) Lighting and Shadows
- 3) Presence of other objects
- 4) Focus and Resolution
- 5) Noise

At the very basic level, any image processing tool works with intensities and intensities only. At least this is the case while dealing with gray images. This stresses on the fact that the input (raw images) must be of very high quality. The quality is not only determined by the resolution of the image, which in fact is very good in this case (1024 x

1024 pixels), but also by the lighting, composition and focus of this dynamically changing system.

1. **Focus** : The insect to be detected is defocused. Hence, however large the resolution is, the edges are still not well defined.
2. **Lighting** : The light intensity can be seen as fluctuations of intensity on the image. Although this noise can be corrected, it is impossible to eliminate completely.
3. **Multiple detections** : OpenCV can only identify objects based on color intensity. As the video input is in grayscale, the options are limited. Shadows of the objects are *also* treated as objects. This can lead to errors.
4. **Background** : There is a need to eliminate random objects such as (4) in the figure above as much as possible. There are some standard procedures used to eliminate noise and shadows, and to also isolate dark regions, like in this case, the body. But these procedures also cause a loss of some very important information. To understand how important lighting is, let us look at the following example.

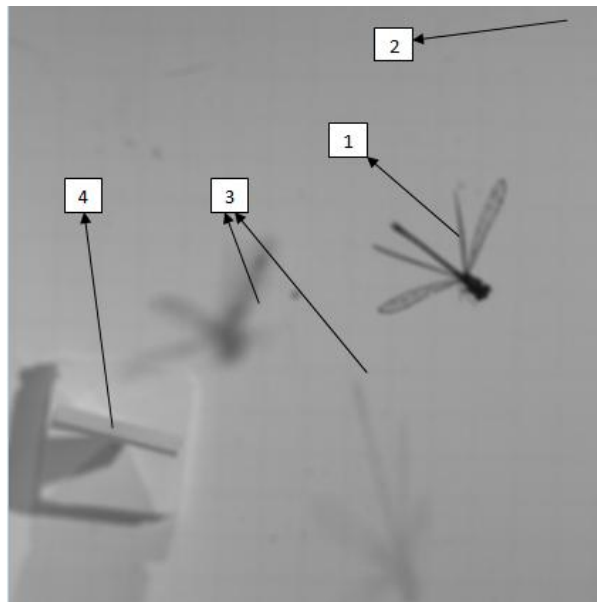


Figure 8 : Issues with raw input




Original set up	With background dimming	With Background dimming and Backlighting
		

Table 5: Effects of lighting

Shown in table 5 are effects of different lighting situations. The first image shows an old can pictured with only front lighting. At the same orientation, two other pictures are taken: one, by darkening the background and two, by adding some backlighting. It can be seen clearly that the third image with a dark background and backlighting gives the most amount of details. In addition to this, the light used is an A/C light, which means, the oscillations of some light pattern will be seen if the camera is fast enough. This is seen as

varying bands of intensities in the raw input. Fig. 9 shows the image at a threshold value of 88 to see the lines or bands caused by the oscillating light.

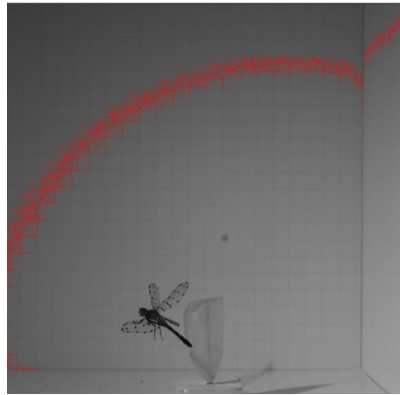


Figure 9 : Light 'Bands' seen in the video

Issues with Manual Tracking

Although manual tracking has been projected as a close-to-perfect method, there are several issues that go ignored. One of these issues is about the very method itself. As described earlier, the orthographic planes set to reconstruct the motion and deformation of an insect, are not accurate. This is because at any given frame, the insect is at a different distance from the three cameras. Thus, the magnification and focus change with every time-frame and with respect to every camera.

Perspective Modeling	Orthographic Modeling

Table 6 : Errors in orthographic modeling

Table 6 shows visually, the error that is present in measuring the location and orientation of the body when attempting a plain and simple orthogonal reconstruction. To delve deeper into this concept, the different views also never match simultaneously. This is mainly because the background pictures represent how the camera ‘sees’ the three planes, but the modeling procedure assumes that the planes are perfectly orthographic. Placing three cameras in the modeling software with identical camera matrices decrease manual tracking error dramatically. Seen in table 7 are examples of such frames in the process of tracking the body. Fig 7 shows the camera rig in the Autodesk Maya software. The green color is the mesh that represents the 3D body that is fit to the 2D plane. The three views, top, side and front along with the perspective views were generally made use of for modeling.

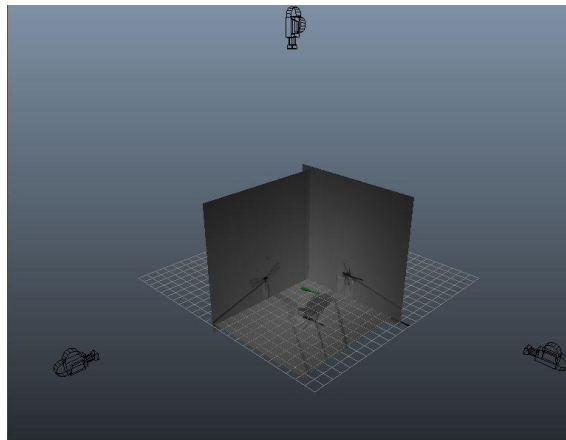


Figure 10: Artificial camera rig/set-up

This logic may be easily extended to a case where the modeling of wings as a whole, or control points in particular is involved. The basic idea is to simulate a back projection model that will retrace how the camera perceives a point.

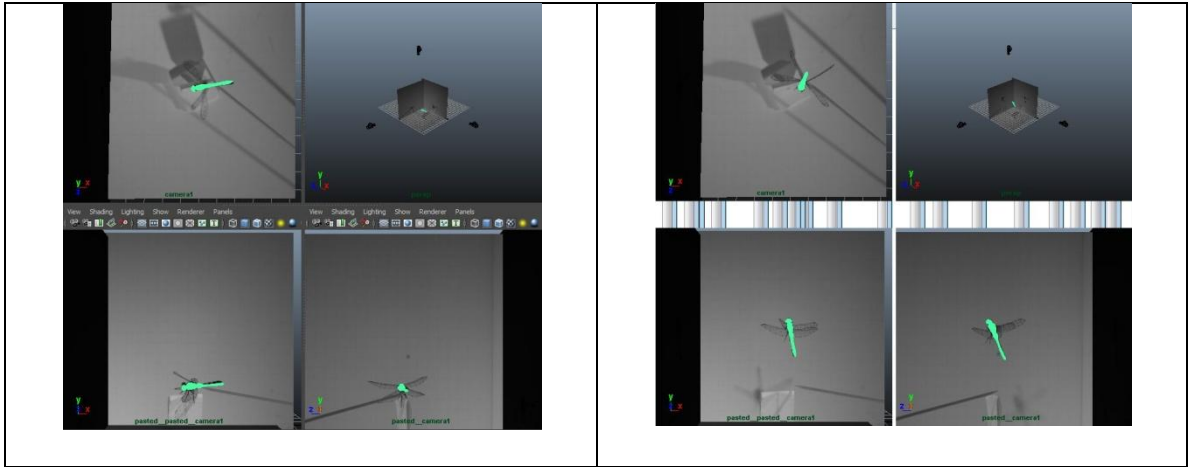


Table 7 : Using the artificial camera rig

Auto-Modeling via Image Processing

3D modeling is essentially locating the points on the solid mesh of the body accurately. Once all the points are known, a sensible relation may be derived for the interconnecting lines and therefore the surfaces (triangulated) involved. Here we will look at a tool that helps to construct the body and the wings of an insect automatically. This will be useful in the reconstruction process.

Body modeling

Most insect bodies are symmetric in one particular plane. Thus we would need information from one less orthogonal plane. Body modeling is done using a single photo of the insect taken in top view, along with a file called the ‘profile-descriptor’. This file contains information as to how the profile or side-view of the insect changes. The code processes the picture by first thresholding the picture. This is the process of separating

the darker portions of the picture from the lighter ones. Since the body of a dragonfly is opaque and the wings are transparent, this proves to be a good technique. The body may be then constructed as semi-ellipses. The major axis of each ellipse that forms one particular section of the body is along the top view. The minor axis is obtained from the two separate profile descriptors that describe the top profile and the bottom profile as seen below. The sections of semi ellipses form a fairly accurate representation of the dragonfly body.

The veins of the wing on the other hand are darker. The edges and veins may be directly obtained. The body and wing information together form the 3D model of the insect. The insect model is created in ten seconds or less. This is much faster than a manual attempt to model the body and wings in Autodesk Maya. We start off with an image as shown below that captures the details of the insect from the top view. This picture provides information about both the body and the wings. The image may be skewed or tilted. The program rotates the whole image so that the body is exactly vertical. The thresholded image gives mainly the body information. Then, there are several sections of the body that are automatically and accurately selected but some other parts are not. The parts that are not accurately selected are due to mainly the presence of appendages such as legs(tibia and femur, as shown below). The 'void' of information in this portion of the wing is filled in by interpolation. Now that the top view information is known, the program loads the profile descriptor files that were described earlier and builds the body. A canny edge detector is used to capture the wing points, that is, points that describe the wing veins and wing boundaries. Also, the stigma being darker in color may be easily detected.

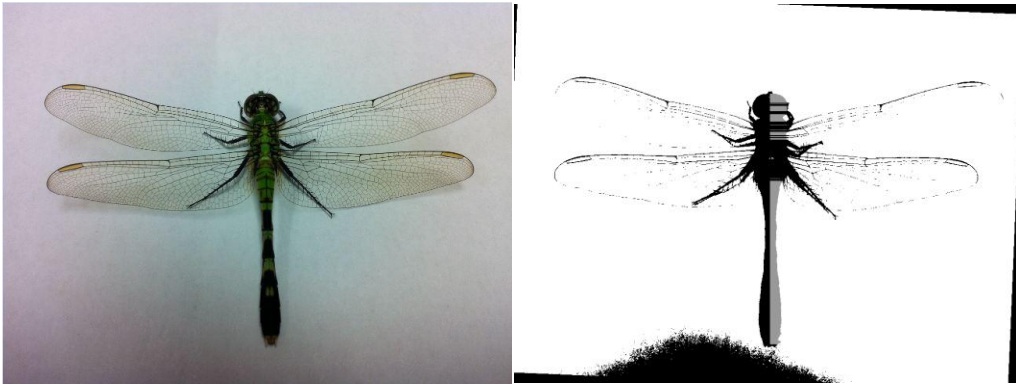


Figure 11: *Left*-Original image (top view); *Right*-Thresholded and corrected image along with detected segments

Figure 10 shows all the detected segments as gray horizontal line. The vertical resolution is one pixel, which means that one line is juxtaposed with the next. This makes it seem like one continuous gray region. This information is used along with the profile descriptor files that are obtained from a side view, as shown below in fig. 11. This is a specific file generated for one particular species and need not be generated repeatedly.



Figure 12 : Side view

The generated wireframe body is shown in fig. 12, and matches very closely with the actual dragonfly shape.

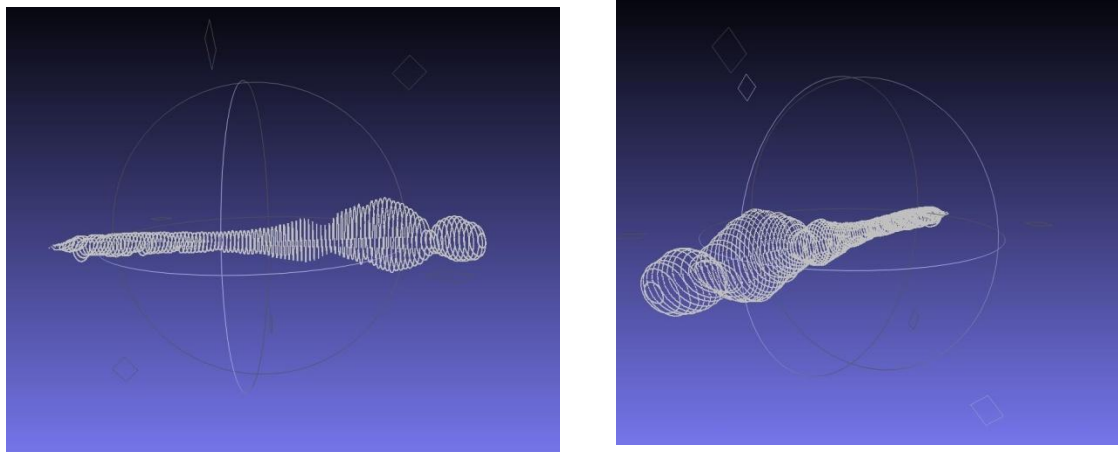


Figure 13 : 3D wire-frame rendering of the reconstructed body

As described earlier, the critical points on the wing such as the stigma and the veins are darker, and hence may be directly obtained from the picture. The table below shows different perspectives of the fully constructed insect. Again, this information is collected in less than ten seconds. Compared to a manual reconstruction, the body is as accurate if not more, and the wings have extra information such as the veins, critical parts like the stigma and the important edges. The table below shows the actual dragonfly, a picture of the manual reconstruction and another obtained from the code.

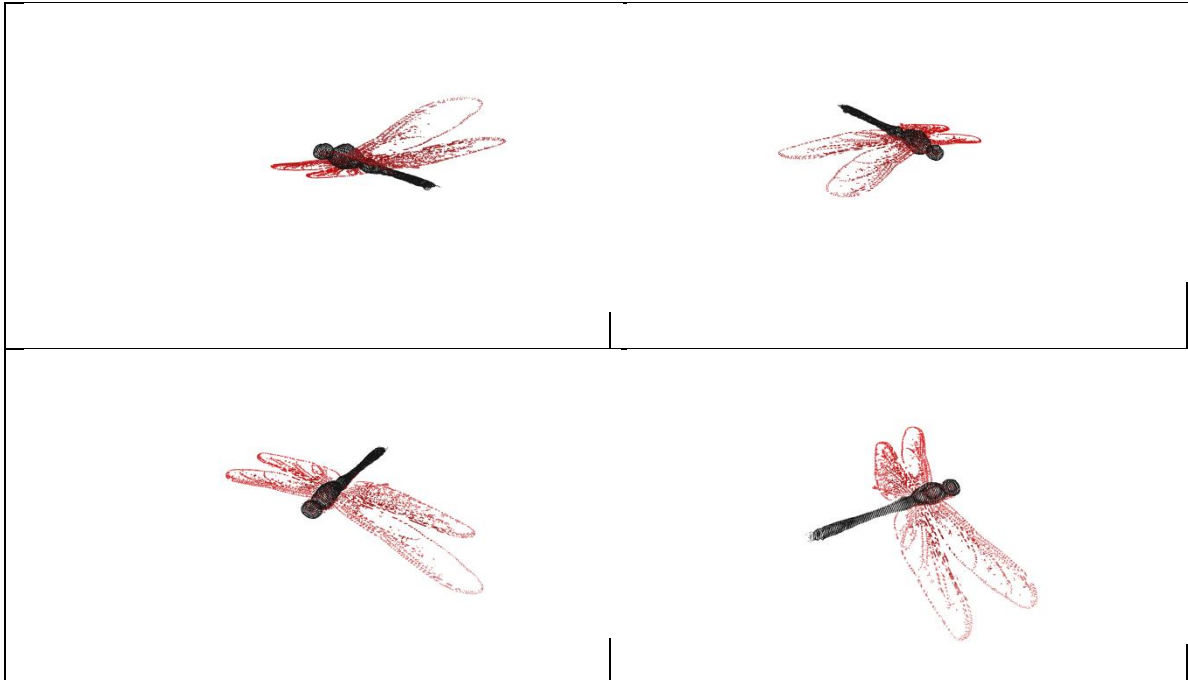


Table 8 : Different views of the fully constructed body

Table 9 tells us that the information obtained by a manual reconstruction or automatic reconstruction in the static case cannot be directly compared. Both methods give us the overall idea. Although the manually reconstructed model is visually appealing, it is only as accurate as the creator himself. The automatic method does not look for a 'better-looking' model but is scientifically correct and is accurate to one pixel dimension. The automatic method is faster and provides more information about the wing. The manual method is more accurate-looking, and automatically has catmul-clark surfaces defined between the edges formed by the points.

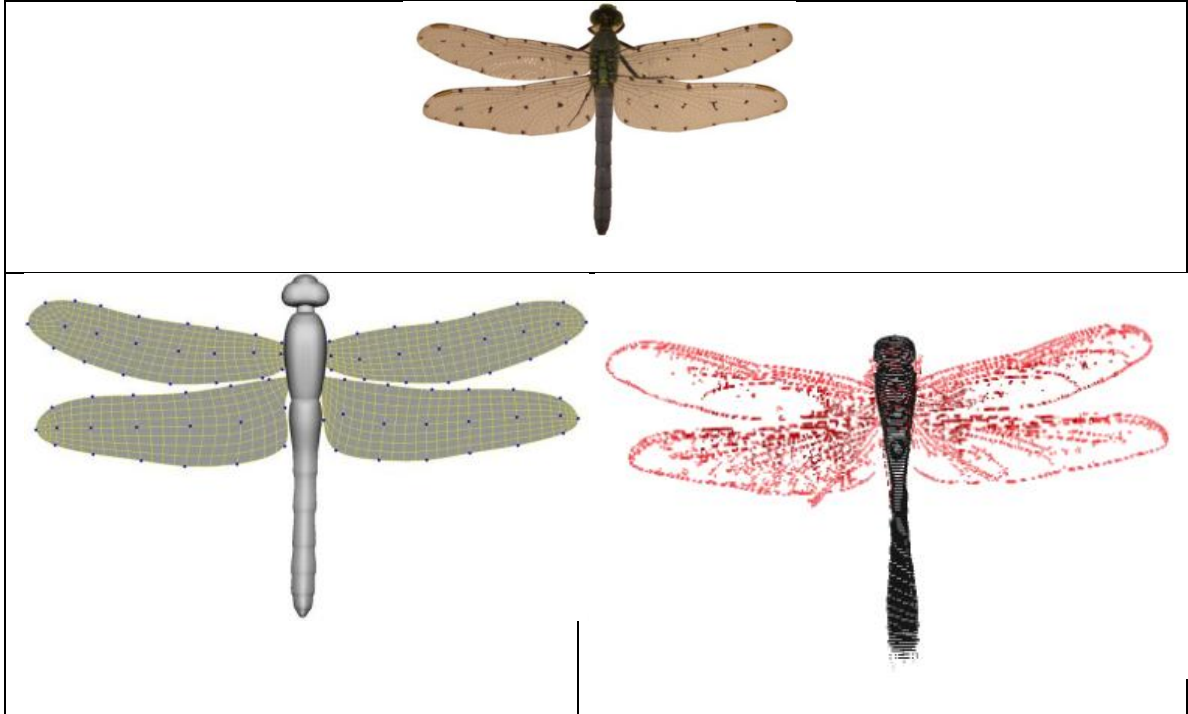


Table 9 : Comparison of static reconstruction results

Mathematical Modeling of the Wings

In the previous section, the body of the insect was modeled mathematically, even though this wasn't stated explicitly. This was done by creating semi-ellipses on either side of the body (Dorsal and Ventral) with the help of information from the profile descriptor files and the body segments. The wings however exist as random points and are not related in any way. Mathematical modeling might be necessary at some stage.

Each wing may be modeled as a separate 3D curve. There are several ways of representing a wing; two such methods are described below.

Modeling using b-splines

Most shapes are too complex to be represented by simple equations in the Cartesian or polar form. A spline curve or a Bezier curve is more flexible and can adapt to randomly varying control points very well. A spline curve is a series of curve segments that are joined in such a way that the result is a single continuous curve. Without going into the mathematics of it, curvature continuity and point-to-point continuity are required everywhere. Given m real values of t_i (called knots) with $t_0 \leq t_1 \leq \dots \leq t_{m-1}$, a b-spline of degree n is a parametric curve $\mathbf{S}: [t_n, t_{m-n-1}] \rightarrow \mathbf{R}^d$ composed of a linear combination of basis b-splines, $b_{i,n}$ of degree ‘ n ’

$$S(t) = \sum_{i=0}^{m-n-2} P_i b_{i,n}(t), t \in [t_n, t_{m-n-1}] \quad S(t) = \sum_{i=0}^{m-n-2} P_i b_{i,n}(t), t \in [t_n, t_{m-n-1}]$$

The points P (“control points” or “De-Boor Points”) here are the points on the outermost contour of the wing. More number of control points results in a ‘tighter’ b-spline. The points on the wing may be digitized by individual mouse clicks or obtained automatically by finding the points on the outermost contour. As an example, let us study the left fore wing. Now, we may recognize the outline of the wing directly. A random number of points may be digitized and used as the control points on the wing. Next, a closed b-spline is drawn using these control points. As seen below in fig. 13 , the b-spline approach gives an almost perfect fit using the control points.

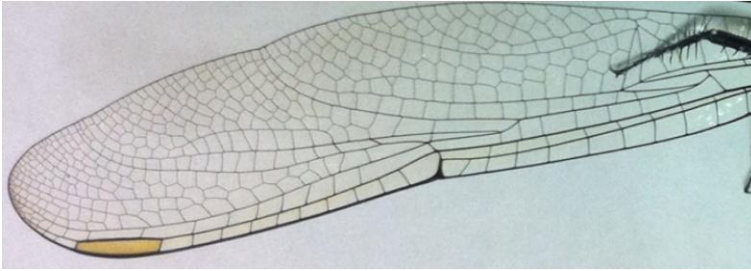


Figure 14 : Actual Left Fore-wing

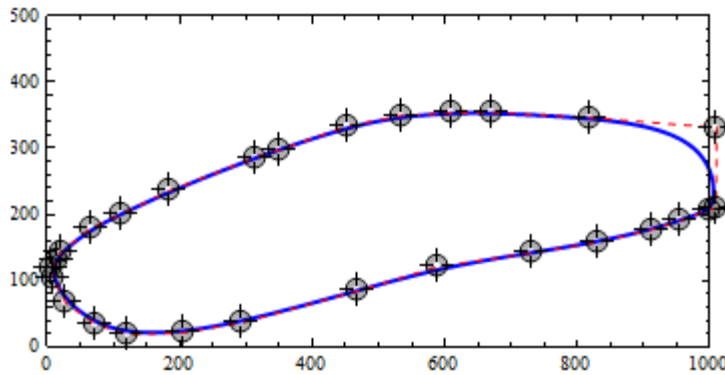


Figure 15 : Closed b-spline modelling the Left fore-wing

The digitized points are shown in fig. 13 as gray circles on the b-spline blue in color. Not only is the b-spline method accurate, it is also very flexible. Changing the location of any control point changes the shape of the whole curve or at least part of it, depending on the degree. However, there is no definite equation that we can relate the curve to. The parametric model solves this problem.

Modeling using Parametric Equations

The same left fore-wing may be parameterized using some variables to give a closed curve. Shown in fig. 14 is the left fore-wing (rotated) of the same test subject along with the points that were used to ‘fit’ a parametric equation to it.

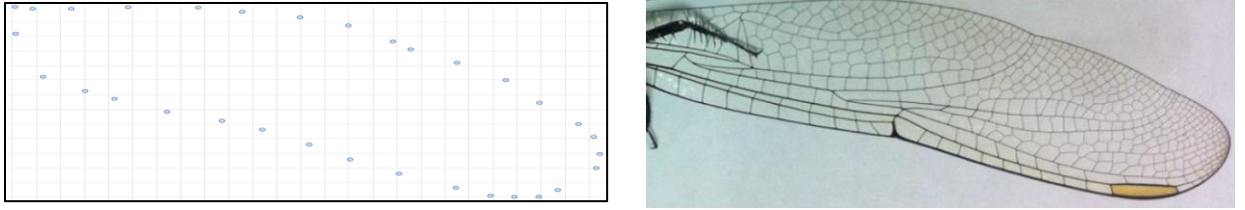
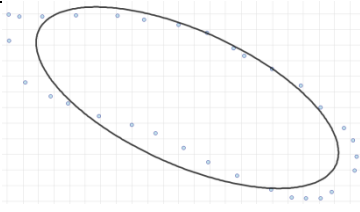
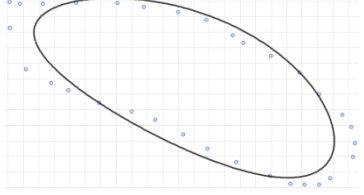
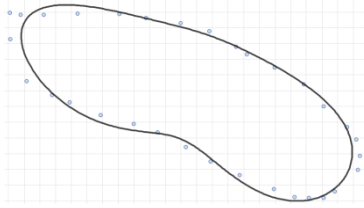


Figure 16 : Digitized points along with the original picture

The method of finding a perfect fit is using harmonics. For example, a standard approach is to use just one harmonic. The resulting curve is an ellipse that is a best fit to all the given points. We may add several harmonics to let the curve ‘fit’ the points better. The table below shows details about the same. As we can notice, the curve progressively becomes more accurate when we add more harmonics to represent the closed curve through the points. Both the methods, Bezier curves and parametric curves have their own advantages and disadvantages.

No. of Harmonics used	Parametric curve	Corresponding equations
1		$X[u] = 523.84535 + 30.157815 \sin(2\pi u) + 30.157815 \cos(2\pi u)$ $Y[u] = 196.07313 - 142.6678 \sin(2\pi u) - 142.6678 \cos(2\pi u)$
		$X[u] = 524.91486 + 32.240129 \sin(2\pi u) + 32.240129 \cos(2\pi u)$

2		$449.69821 \cdot \sin(4 \cdot \pi \cdot u)$ $449.69821 \cdot \cos(4 \cdot \pi \cdot u)$ $Y[u] = 198.28515$ $137.44528 \cdot \sin(2 \cdot \pi \cdot u)$ $137.44528 \cdot \cos(2 \cdot \pi \cdot u)$ $77.30617 \cdot \sin(4 \cdot \pi \cdot u)$ $77.30617 \cdot \cos(4 \cdot \pi \cdot u)$
4		$X[u] = 569.23315$ $+55.522314 \cdot \sin(2 \cdot \pi \cdot u)$ $+55.522314 \cdot \cos(2 \cdot \pi \cdot u)$ $458.27476 \cdot \sin(4 \cdot \pi \cdot u)$ $458.27476 \cdot \cos(4 \cdot \pi \cdot u)$ $+5.5309542 \cdot \sin(6 \cdot \pi \cdot u)$ $+5.5309542 \cdot \cos(6 \cdot \pi \cdot u)$ $9.0340476 \cdot \sin(8 \cdot \pi \cdot u)$ $9.0340476 \cdot \cos(8 \cdot \pi \cdot u)$ $Y[u] = 221.92512$ $157.50302 \cdot \sin(2 \cdot \pi \cdot u)$ $157.50302 \cdot \cos(2 \cdot \pi \cdot u)$ $+89.682758 \cdot \sin(4 \cdot \pi \cdot u)$ $+89.682758 \cdot \cos(4 \cdot \pi \cdot u)$ $+8.3766503 \cdot \sin(6 \cdot \pi \cdot u)$ $+8.3766503 \cdot \cos(6 \cdot \pi \cdot u)$ $5.0076607 \cdot \sin(8 \cdot \pi \cdot u)$ $5.0076607 \cdot \cos(8 \cdot \pi \cdot u)$

These equations may not be easily applied to the tracking solution. The purpose of this exercise was to show that mathematical expressions exist to model both, the wings and the body. The application of the wing modeling will not be applied in the first stage of wing kinematics estimation. At a more advanced stage of this research, the above information may prove to be useful.

III. Auto-Tracking Technology

Introduction

Of the several attempts made towards achieving an auto-tracking solution, the two most prominent research contributions are by Ristroph et al(21) and Ebraheem et al(4). Both researchers use the fruit fly (*Drosophila*) as a test subject for their auto tracking technology, and use custom software to achieve the same. However, there are several differences when compared to the methodology used at the FSRG lab.

1. The test subject is different. The FSRG lab uses a female dragonfly (*Ajax Junius*) whereas the above mentioned labs use the Fruit Fly (*Drosophila*).
2. Deformations of the wing and body are more pronounced; The rigid body application may not be useful here.
3. The viewing volumes are very small in both the cases. In other words, the volume of the imaginary cube formed by the intersection of the rays entering each camera is very small compared to the set up here.
4. Ristroph et al (21) use a Hull reconstruction method (HRMT) and Ebraheem et al(4) use a model based approach. A completely different method will be attempted here. The concept of constructing a cloud of voxels was however, inspired by the pictures in Ristroph et al's (21) paper.

5. The complexity of the problem increases as the Dragonfly has four pairs of wings. A combination of these two factors: One, the presence of only one pair of wings on the Drosophila, and Two, the small viewing volume as described above, makes the work of image processing simpler.
6. An added consequence of the small control volume is that shadows are completely negligible.
7. The fruit fly's body is symmetrical. This makes roll estimation difficult. The dragonfly has a definite shape when seen in the profile view. This will be used to estimate roll.
8. Both methods make use of a MATLAB implementation. The image processing here is done using an open source library 'OpenCV' and the code is written in C++.
9. Both codes require the user to manually track the first position of the body and wings. The algorithm then tracks the wings and the body at in subsequent frames.

Thus, it might not be fair to compare the three methods. However, the aim of the three methods is similar, and this will be used to connect them somehow. At this juncture it might be relevant to briefly describe the afore mentioned methods.

Ristroph et al(21) use Phantom CMOS digital cameras at a resolution of 512x512 to film the insect. Magnification is achieved by optical bellows. The cubic filming volume has a side equal to 1.5 cm. This is to eliminate perspective distortion. In my method, perspective distortion is eliminated by modeling the camera itself. Shadows are

eliminated by directing the light into the camera lens. A laser is used to trigger the three cameras when the fly is in the filming volume. Voxels are generated by a MATLAB code and the data is analyzed using Principle component analysis. A clustering algorithm identifies different parts of the body and the wing. Body as well as wing kinematics are generated. Finally the auto-tracking method is compared to a manual tracking method.

Ebraheem et al(4) film the fly at 6000 frames per second with a Photron Ultia camera, also at a resolution of 512 x 512. At each frame and for each camera, a 1D Gaussian mixture model is fit into the images and the fly pixels are segmented using an expectation-maximization approach. The pose of the fly is predicted dynamically using the previous pose (in the previous frame). A generative model is used instead of a triangle mesh model to reduce complexity. The images are segmented to body and appendages using a histogram of pixel intensities. About 500 samples are constructed manually in the training stage for both escape and voluntary take offs. The code then “learns” to predict the wing and body movement in frames. Errors and kinematics results are also presented.

The Input Image

For any image processing application, it is important to study the data-set in consideration first. Shown below in fig. is a sample image. The raw image has dimensions of 1024 x 1024 and is an 8 bit integer ‘uint’ image, which means that there is a maximum possibility of $256(2^8)$ gray shades that can be used here. In other words, the gray intensity values all range from 0 to 255, and are integers. A custom tool was developed in MATLAB to judge as to which set of images are suitable for studying. Suppose we have an input image as shown below. The histogram shows that most of the

pixels are clustered around the mid-tones region. The image is shown here in a ‘gray’ color map along with its histogram. The image is thus not balanced well.

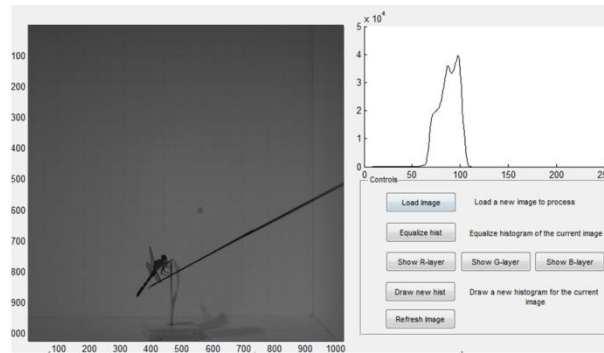


Figure 17 : The original image with its histogram

We may use the tool to draw a custom histogram to see how it affects the raw image. To include more details we may increase the number of pixels in the shadow region of the histogram and decrease the highlights in the far right end.

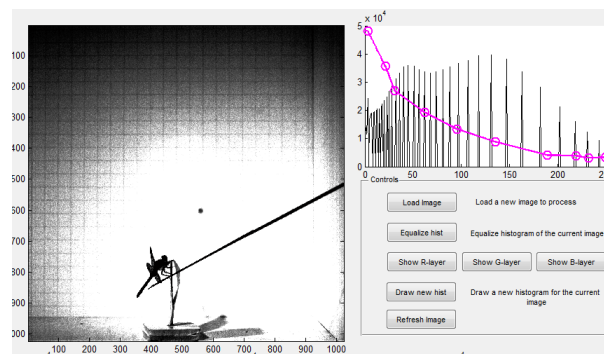


Figure 18 : Forcing the image to have a particular histogram

The non-uniform distribution of light is clearly seen in this image. All three spotlights are focused at the ‘origin’ which is the brightest region. The ‘forced’ histogram equalization in fig. 18 seems to be useful as the background is eliminated and the image is segmented

or separated from the rest of the image. But as we can see, only details of the body are seen as the lighter wings are flushed out too.

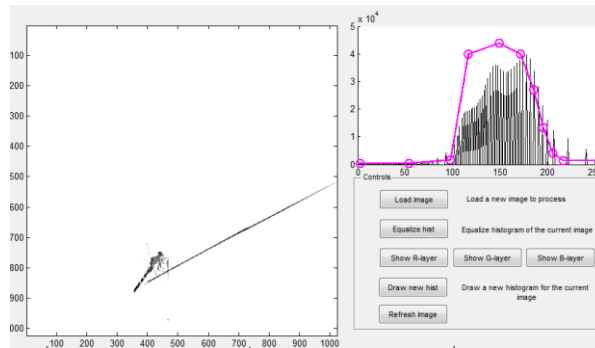


Figure 19 : Segmentation using histograms

Processing a Single Image

Though the dataset we have is actually a ‘video’ or a series of images from the high speed camera, we need to address the issue of processing single images first. Later, a correlation between consequent images may be made. The code involves several levels of user interaction, but that will not be discussed here. The steps of processing will be outlined below.

Smoothing

Smoothing, or *blurring* as it may be better known, is the process using a gaussian function to smudge the image. Mathematically, applying the gaussian function is the same as transforming the image with :

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

This is also known as a Weistrass transform in two dimensions. The effect obtained is that of viewing the image through a blurred translucent screen. This helps spread out the dark shadows and external objects that are present in the scene.

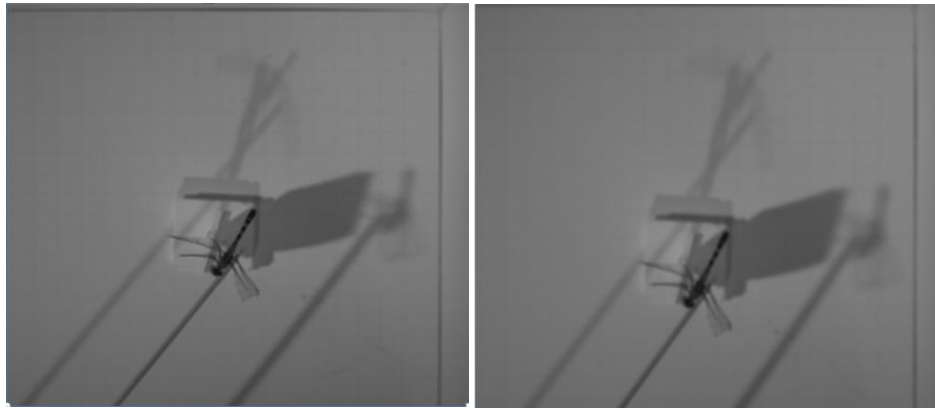


Figure 20 : Image after smoothing

The effects of smoothing the image, specifically by a 17 x 17 window of pixels cannot be appreciated at this stage directly. This helps the next stage of processing, which is thresholding.

Thresholding

Thresholding is a process that is applied to each point (or pixel) in the image. According to the values that are fed into the function, each pixel is marked as an “object” pixel if the value is above a threshold or in between a particular range, and is marked as a “background” pixel if the afore mentioned conditions are not satisfied. In this particular

implementation, a truncation method is used wherein all the pixels that aren't in the provided range or threshold are truncated to a particular value.



Figure 21 : Image after thresholding

As seen above, the combined effect of blurring followed by thresholding has successfully segmented the body from the background. The shadows and external objects have been eliminated. However, the downside is that information from the body has also been eliminated. Also, the body is not continuous in all regions as holes are seen. These defects will be corrected by morphological functions.

Equalizing the Histogram

To be able to use the image, we need to convert the image to a black and white image, or ones and zeros. Since the values around the dragonfly's body are all truncated to the same value, equalizing the histogram of this modified image simply lifts the values of all the pixels that have been truncated to one set value by the previous stage to the maximum value. As a result the following conversion is achieved.



Figure 22 : Image after histogram equalization

Morphological Operations

The two most common morphological operations are Dilating and Eroding. Both these are applied to binary (or black and white) images. The effect of this operation is to gradually enlarge the boundaries of regions of the foreground pixels. Thus the holes in an image become smaller. The erode operator works mainly on binary images but some versions of it also work on grayscale images. The effect of the function is to erode away boundaries of regions on the foreground. The holes therefore become larger.

The result of equalizing the histogram as seen is to create a hole (the darker body) with a white foreground. Thus to add more artificial detail, we need to use the erode operator. The result of this stage is shown below.



Figure 23 : Image after eroding

Edge Detection

The final stage of processing is detecting the edges of the image produced after the morphological operations. The method used is called Canny edge detection. The canny edge detector uses a multi stage algorithm to detect a wide range of edges. Without going into much detail, the canny edge detector uses the calculus of variations which optimizes a given functional(Canny Edge Detector 2012). As we can see in the image below, the outline of the object has been detected, thanks to all the previous stages.

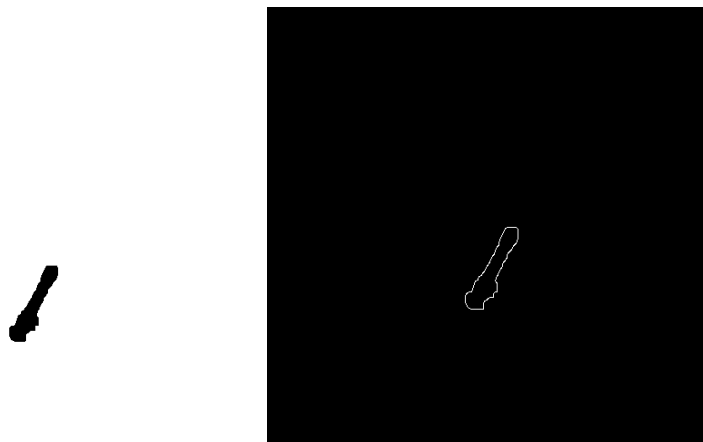


Figure 24 : Edge detection

Combining the Three Camera-Views

The discussion in the previous section gives us an idea about how one camera image is processed. To have a useful outcome, we will have to use information from all three camera views. It is a fact that two camera views may automatically give information about the third, but all the details might not surface. It is even safe to say that to an extent, two orthographic views may never be able to give enough information about the solid object. The work of Ebraheem et al(4) is not one of a kind, in a manner that the method of using *voxels* has been made use of in many research works. A voxel may be described as a discrete 3D pixel. As a 3D object may be projected on the three mutually perpendicular planes to obtain the so called ‘orthographic views’, we may imagine that a 3D model may be reconstructed from three such views.

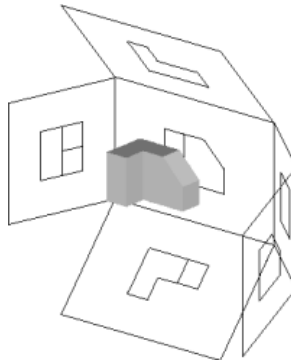


Figure 25 : Orthographic views of an object

As seen in fig. 23, we need to address the question of whether the solid gray body in between the expanded orthographic planes has enough information to reconstruct the body in between. To show that this is not possible, consider a simple object as shown

below. If we had to use only two images or two views to reconstruct it, clearly the result is not desirable as there is simply not enough information.

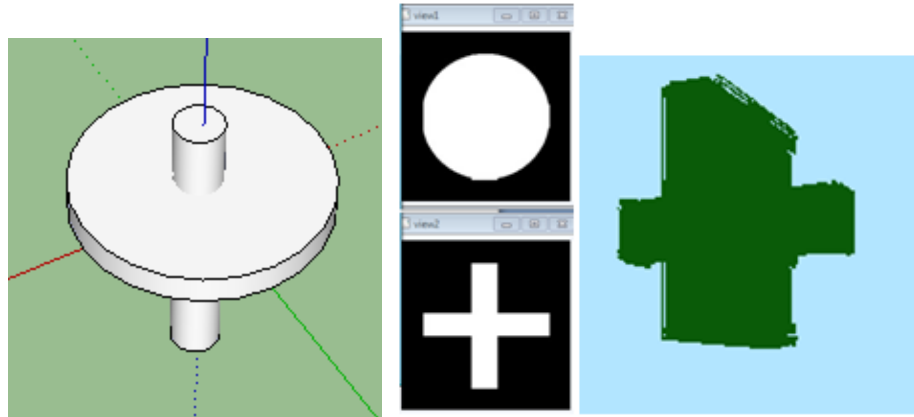


Figure 26 : Reconstruction errors while using only two views

Extending the processing of one camera view to the others is simple since the operations are the same and are done in the same order.

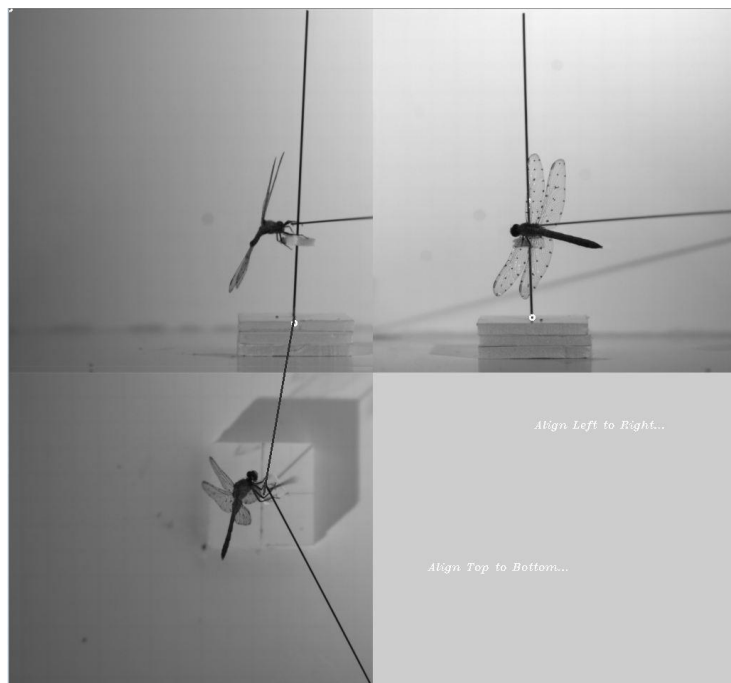


Figure 27 : Three views of the camera

Contour Information

The views are processed as mentioned before and combined together. The next step is simplifying the combined image into ‘quad’, ‘tri’ and ‘bi’ contours. These are various levels of contours that reveal important information. The edge detected dragonfly body in each view is fit to a contour. Now, at very high resolution, the contour closely resembles the edge that is detected. However, we may break down these contours to have lesser number of total line segments. A typical view is shown below. The green rectangle



around the body of the dragonfly is the ‘bounding box’. The yellow curve is the contour at high resolution, and closely matches the edge detected by the canny algorithm. The blue quadrilateral is the quad-contour and the red triangle is tri-contour. The bi-contour simply joins the head point to the tail point. The purple curve is an ellipse fit to the points on the contour. Because of the peculiar shape of the

dragonfly body, the location of the head point and the tail point is decided by analyzing as to which side this ellipse is biased towards. These contours help simplify the problem.

The only user input required is that of aligning the front view (Camera 1), top view (camera 2) and side view (camera 3). Once this is done, this set of contours and other information is displayed for all three views. Using the pixel density of the image, it is possible to identify with 100% accuracy, the head point, tail point and “roll” point as seen in figure 26. The largest white circle is the “head point”, the white point is the “tail point” and the gray, medium sized circle is the “roll point”. These three points form a plane. It is possible to find the pose of a triangle once a mathematical model of the dragonfly is

given. But this is not the approach that will be used here. Once all this information is stored in structures (a c++ object), we may proceed to the next frame.

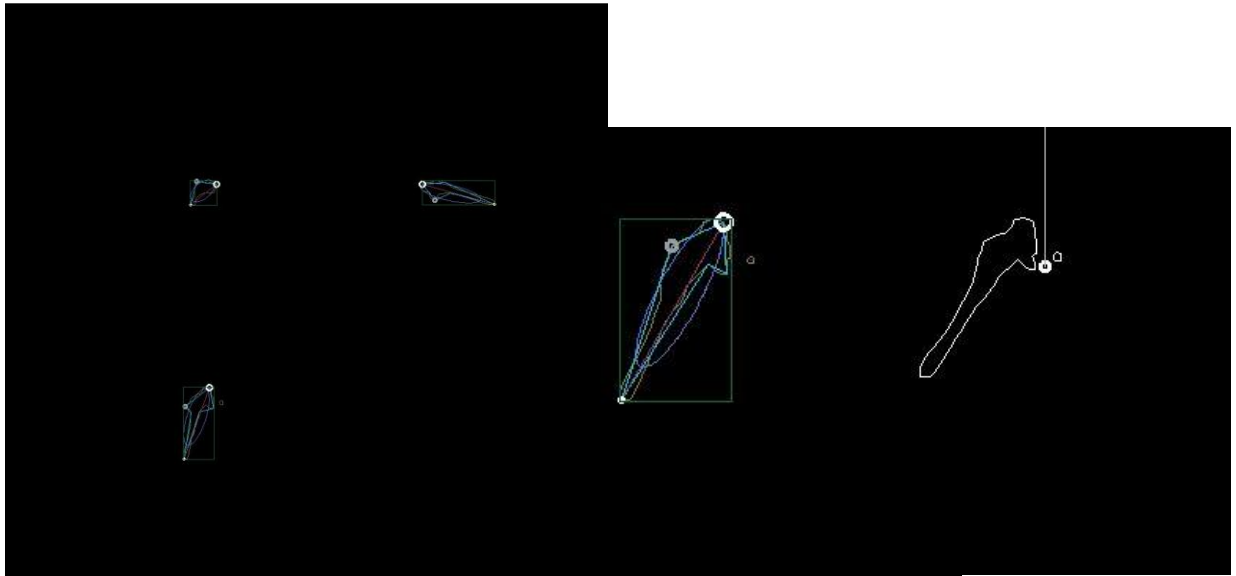


Figure 28 : Decomposing into sets of contours

Constructing the Voxel Model

Now that all the information has been collected from all views, we may construct the voxel representation of the body. The contour that closely follows the actual edge of the body is a closed curve. This curve is first filled so that all points inside the contour are ones, and all points outside the contour are zeros. Essentially this is like creating a surface from a closed curve. Next, the surface formed in each plane is extruded outwards to form a volume that has a cross section that is the same as the surface obtained from the curve in the first place. Thus, at this stage we have three extruded volumes from each

view. The bounding boxes (in green) that are calculated in the contour stage define the bounding ‘volume’ that will enclose the final body volume. This 3D array is dynamically created in the computer at every single frame since the dimensions of this box are different for different frames.

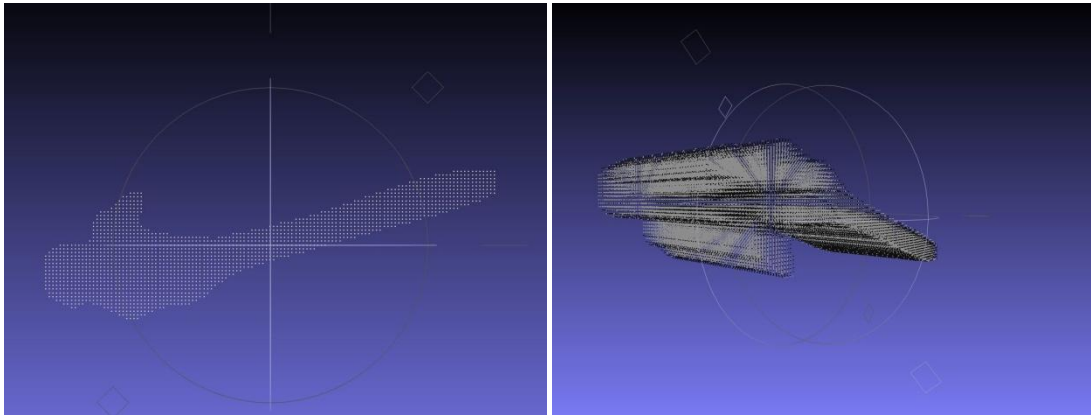


Figure 29 : Surface and extruded volume of one view

Fig. 27 shows an example of the 2D surface constructed from the edges and also the extruded volume. The intersection of such extruded volumes from all three views gives a 3D representation of the body in voxel format. Fig. 28 shows the resulting voxel formed using the information from one time frame(i.e. three cameras).

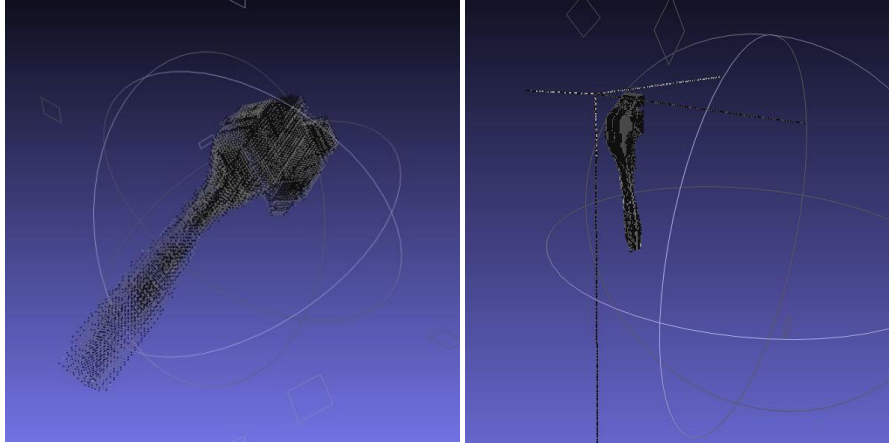


Figure 30 : Two views of the reconstructed body

One method of finding the orientation of the body is to fit the body that was constructed originally (discussed in Auto-modeling) to this set of voxels. The least square fit would determine the orientation. However this method was not very accurate and was hence discarded. The results are shown below. The left hand side image shows the axes and the voxel body whereas the right hand side shows the pre-constructed body oriented to match these voxels using least squares.

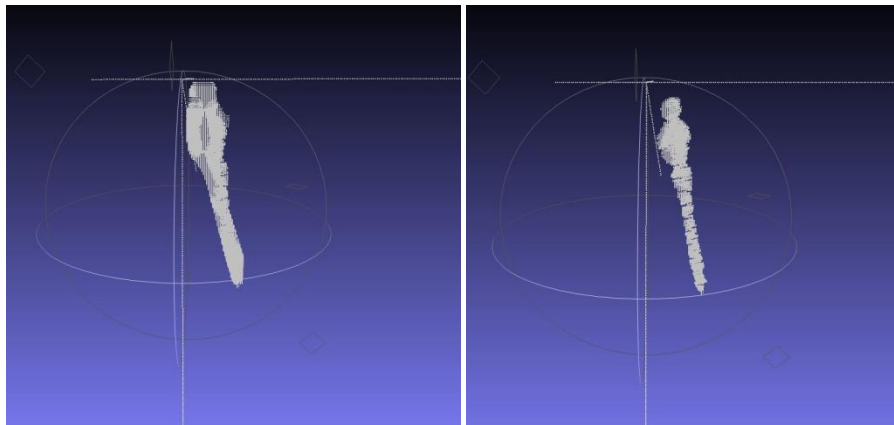


Figure 31 : Voxel and pre-constructed bodies

Analyzing Voxels

The generated voxels are stored in the dynamically created bounding volume that was described earlier. Recounting the shape of the dragonfly's body we see that the 'hump' on the back of the dragonfly may be used to our advantage. The shoulder and the thorax turn upwards near the head. The body is also symmetric about the longitudinal plane. Therefore, the orientation of the dragonfly is fully specified by this particular plane only. There is no other information required. We first slice the voxel body and find out the centroids of the each slice (or surface). To obtain the orientation of the plane, we define three points: A point on the tip of the head, a point on the shoulder and a point on the auricle or the portion where the tail segments start. We may also choose the tail point as the third point.

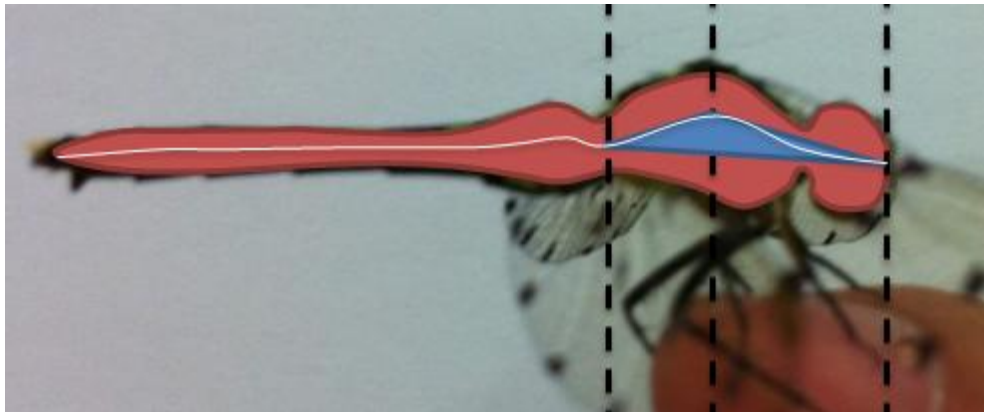


Figure 32 : The centroid line and orientation triangle

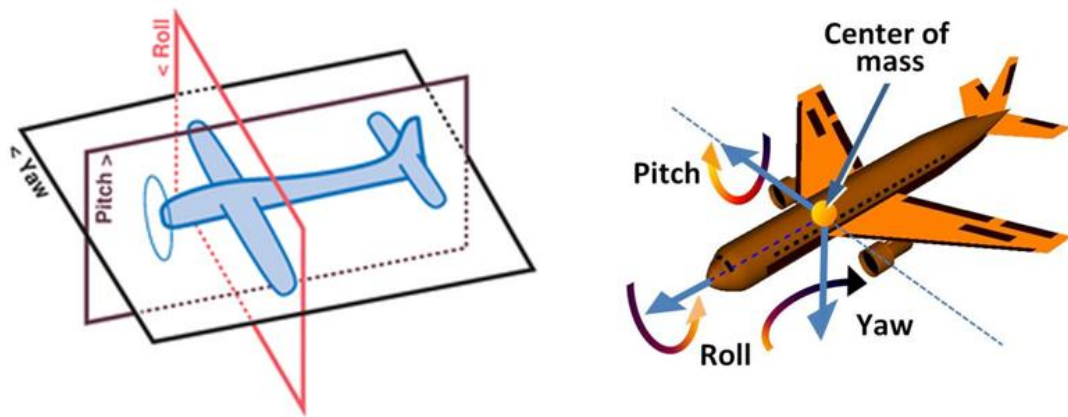
This as we will see later is troublesome as the tail moves erratically throughout any take off or maneuvers. All these points lie on the curve that joins the centroids 'line' which is shown in white above. As we can see below, the red shaded area is the section that cuts

the dragonfly into two longitudinal halves. The blue 'orientation triangle' is the plane formed by the three points mentioned earlier. The fact that these points are not collinear at any given point of time may be used to our advantage. These three points specify the orientation triangle plane in 3D space, which then gives us information about how the dragonfly is oriented (Pitch, Roll and Yaw).

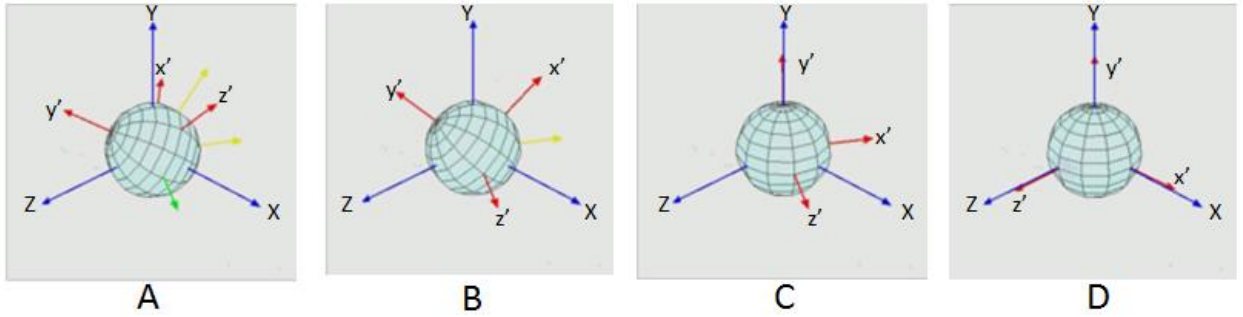
Although the orientation of this plane is continuously changing, the choice of the three points that form the orientation triangle is one that gives reliable information. If the third point was selected to be the tail point, the problem would yield more erratic results. This is because the dragonfly's tail segments are much more flexible. The body can thus be classified as the more rigid body-head portion and the more flexible tail portion. The body-head portion does not deform and so it makes sense to use the this triangle rather than the larger, more unstable triangle involving the tail. However, algorithm wise, involving the tail would be simpler as it would not require analyzing of voxels. All points may be located on the surface of the Dragonfly body itself.

Calculating Pitch, Roll and Yaw

Pitch, Roll and Yaw are the three critical flight dynamics parameters that describe the orientation of an aircraft about its centre of mass. This set of angles along with the translation of the centre of mass in 3D is the information we look to harvest. There are several ways to calculate the pitch, roll and yaw but the simplest way is to use the definition itself to calculate the successive three rotations required to transform the object coordinate system to the global (fixed) coordinate system.



The dragonfly is oriented randomly with respect to the global coordinate system. Using vector manipulation in an iterative procedure, we may find a reasonably accurate solution. To calculate the angles, the code iterates until it hits an appropriate plane, and stores that rotation in degrees. The order of calculating the angles are ‘Yaw’ first, then ‘Pitch’ and finally ‘Roll’. Some other authors may use different notations. To understand what is really happening, refer fig. . Initially the object (represented by a sphere) has two axes connected to it. As shown below, the red vectors are the body coordinate system after each rotational transform. The fixed blue vectors constitute the Global coordinate system. Initially the body is oriented at some random angles of pitch roll and yaw. First, we rotate the body about the y' (*Yaw*) axis until the red z' vector hits the X-Z plane. Next, we *Pitch* about the new z' axis until the x' vector reaches the X-Z plane. At this stage the y' axis is in line with the Y axis. Finally we *Roll* about the y' axis so that all three vectors match the corresponding vectors in the global coordinates system. This may also be done using matrices.

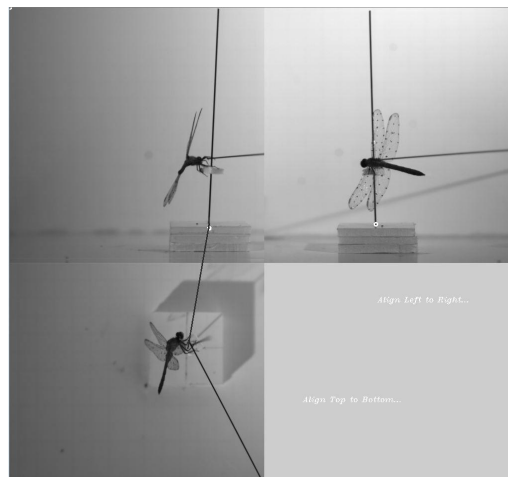


Processing the video

Processing a video is exactly the same as processing multiple images in succession. The processing that was discussed until this point will be applied one by one to all frames that describe the flight of the dragonfly and thus we will obtain the history of motion and kinematics of the body as a result. The steps involved in operating the software to enable processing of the entire video are outlined here.

STEP 1 – Initiation

The user first directs the program to the folder that has the required image sequence. Once the path is set and the code is compiled, the three views from the three cameras are seen on one pop-up window.



STEP 2 – Aligning

The user must then middle-click the image to let the program know that the initiation process has begun. Then the global origin of the imaginary 3-D space is set by dragging first, from the front view (top left) to the side view (top right), and then from the front view to the top view. After this point the continuous auto-tracking begins.

STEP 3 – Tweaking Parameters

The parameters that control the image processing may be tweaked using the trackbar window shown below. The user is allowed to change four parameters:

- **Threshold:** The default value of threshold is 54. It has a range of 1 to 100.
- **Erode:** The default value of the erode trackbar is 4. It ranges from zero (no erode) to 10 (10 iterations of eroding).
- **Smooth:** The smooth trackbar has a default value of 5. It is allowed to have only odd values due to constraints of the function as specified by the OpenCV library.
- **Canny:** The canny trackbar has a default value of 4. As this value increases the size of the segments used to form the closed edge around the dragonfly body decreases.

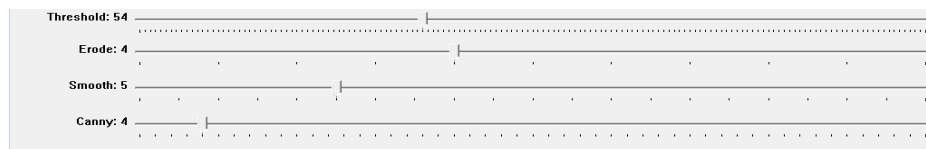


Figure 33 : Trackbar controls

The parameters may be changed at any point of time during the video processing.

STEP 4 – Displaying results

The values of Pitch, Roll and Yaw calculated by the techniques described before are immediately displayed on a separate ‘plot’ window. The red, blue and green lines represent the time history of the values of Pitch, Roll and Yaw respectively. Thus the kinematic parameters of the maneuver are being displayed as the required angle in real time. The range of values of all three angles are from -180° to $+180^{\circ}$.

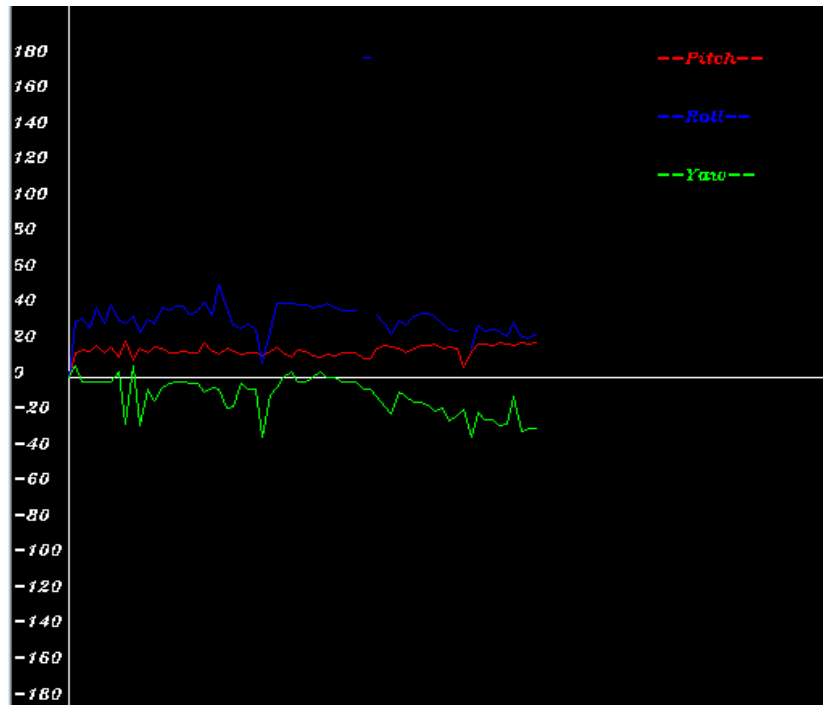


Figure 34 : Displaying Euler angles in real time

Graphical User Interface

There were several ‘codes’ referenced to up until now. These codes are reproduced in the appendix section of this document. For an end user who is just looking for results of each of these codes it is difficult to vary parameters and variable values as is. Therefore a GUI was created to access these codes in a systemized way. The GUI was written in c++ as a forms application. Some screenshots of the GUI are shown below:-

Initial screen

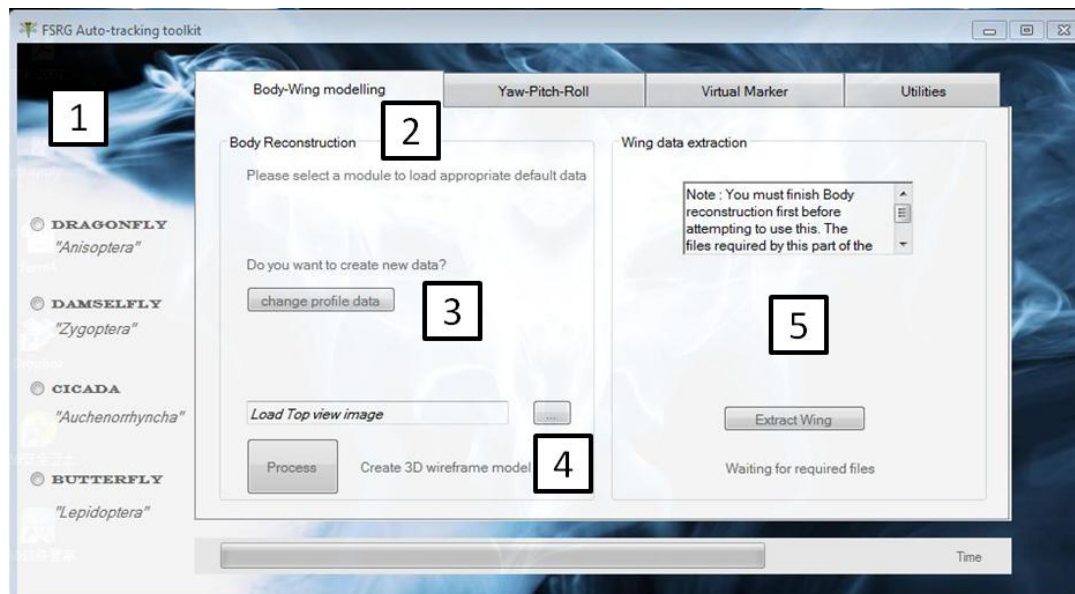


Figure 35 : Initial screen of the GUI Interface

Fig. 34 shows the initial screen of the GUI interface. The user first selects which insect is being studied in the region marked ‘1’.

Body-Wing modeling Tab

The region marked '2' is the *Body-Wing* modeling tab where the two entities of the insect, Body (comprising of the head, thorax and abdomen) are built. Once a choice of the insect is made through the radio buttons in region one, the appropriate 'profile descriptor' of that particular choice of insect is loaded. Note that only the 'Dragonfly' module has been completed. To manually digitize a profile descriptor, the user presses the 'change profile data' button in region '3'. The program for manually clicking the top and bottom profiles of the insect then pops up. The user then clicks the two endpoints to generate a grid in light gray. This grid helps the user to manually trace the upper and lower profiles. Once the user completes digitizing the points on both the profiles, the profile descriptor files for that specific species are written onto a file. A screen-shot of this particular utility is shown in fig. 35.

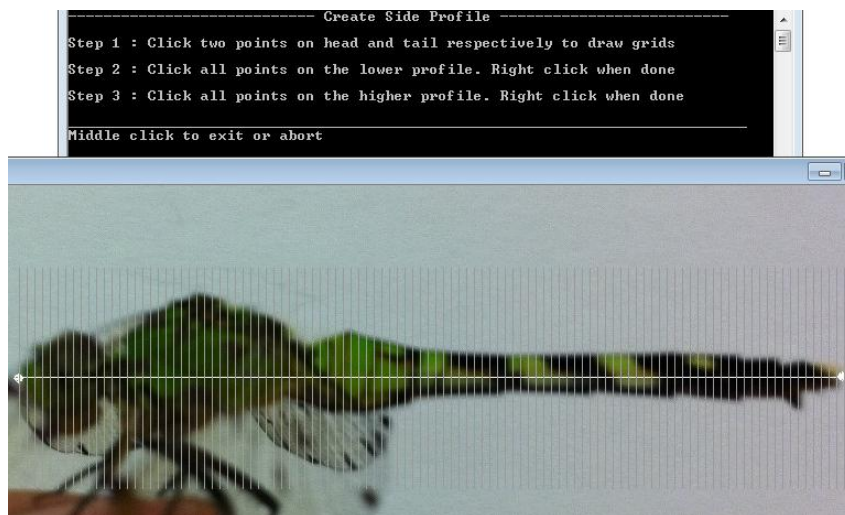


Figure 36 : The profile descriptor utility with instructions

Once the profile descriptor is loaded and the body is constructed, the wing extraction utility is enabled and the status-bar and the time are updated as shown below.

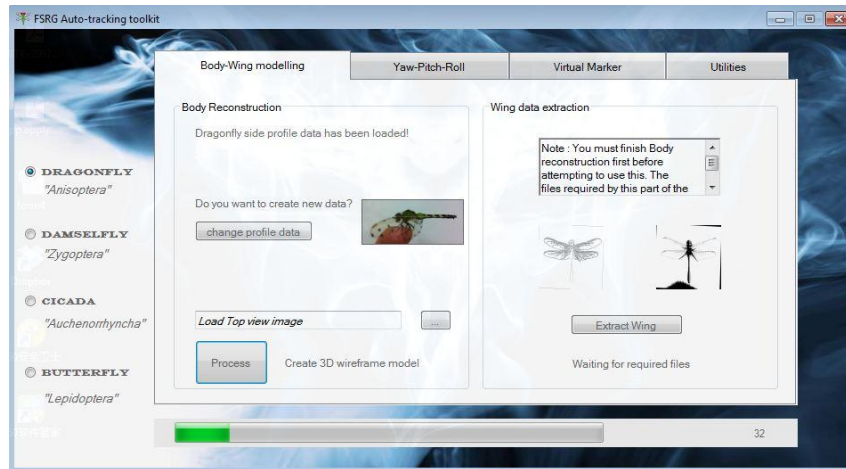


Figure 37 : Body-wing modeling complete

Yaw-Pitch-Roll tab

The second tab is for the Euler angles calculation utility that was described in the previous section. This tab is shown in fig. 35. The user must direct the GUI to the path that contains images from the three different camera views. The user may also choose to ‘Move Files’ to one single location. If the user has moved the files manually into the working directory, he may use the check-box to specify so. Once the three folders are recognized, the views appear on the right hand side of the second tab as shown below. The user specifies three numbers in the text boxes shown in the fourth quadrant of images in the second tab. The Tabs ‘From’, ‘To’ and ‘Skip’ let the user specify the range frames to process and the number of frames to skip in between one processing stage to the next. A ‘Skip’ value of 1 would process each and every frame in the specified range. The skip value may also be specified by the mini trackbar. Once all the parameters are supplied, the ‘Calculate’ button is used to start the Euler angle program.

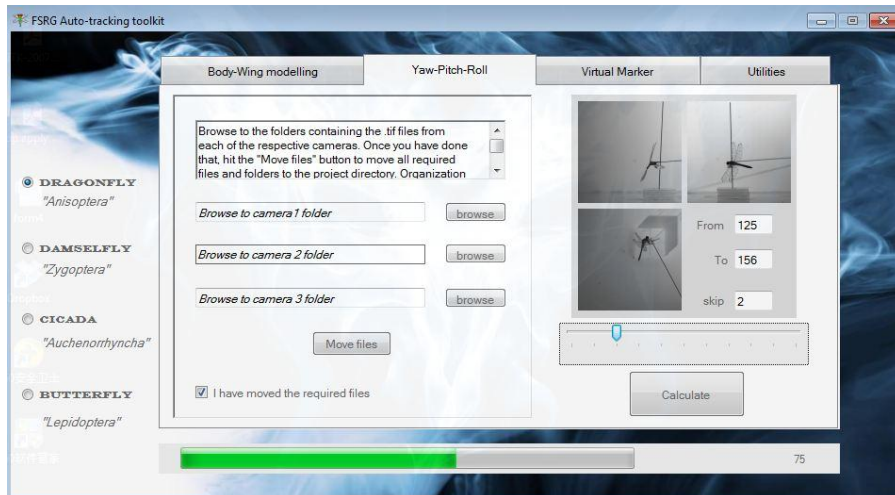


Figure 38 : Yaw - Pitch -Roll calculation tab

Other Tabs

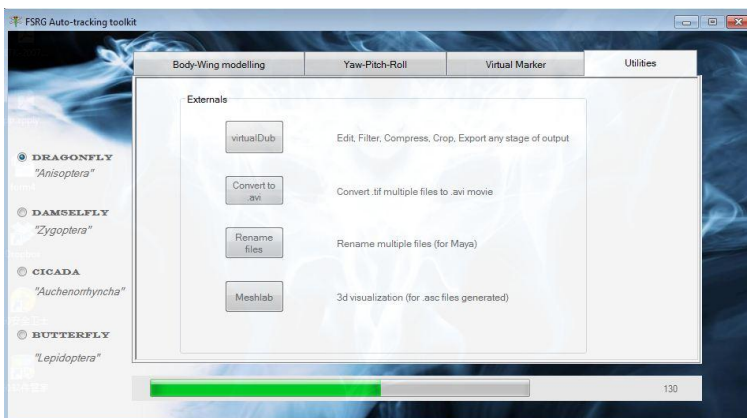
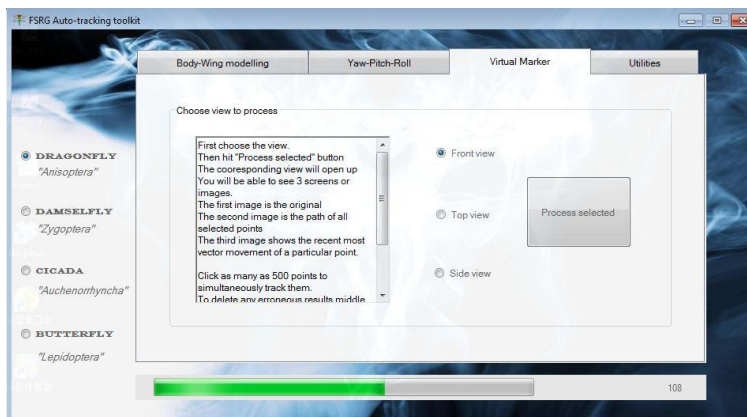


Figure 39 : Virtual marker and Utilities Tab

The third tab shows information about a ‘Virtual Marker technology’. This will be described in the future work section. The GUI also opens the user to some open source utilities, either available online or developed in house. ‘VirtualDub’ is a software that may be used to enhance or preprocess the image before getting into the actual processing stage. The

contrast and sharpness may be improved and the noise may be eliminated. The ‘ConvertAvi’ simply converts the input or the output .tif images into a movie. The ‘RenameFiles’ utility helps the user rename multiple files at once. This is used during the manual reconstruction stage before importing the image files into the software Autodesk Maya. Finally, ‘MeshLab’ is the software that is used to display the ‘.asc’ geometry files that are generated by the code.

Test Cases

In an attempt to further study the working of the Euler angle code, test cases will be discussed in detail. Several cases were studied out of which only some will be discussed in detail. The final results of the other codes will be discussed in brief to confirm the validity of such a software. All cases were studied by first reconstructing the body movements using Autodesk Maya by hand and then comparing it with the results obtained by the software. The Euler angles from the software are displayed or written automatically as discussed earlier. To calculate the Euler angles from the manual reconstruction, a customized code was used that was developed by another member of the FSRG group. This code has been used extensively and tested by the members for other applications.

Test Case 1

Shown alongside is the first case to be studied. The dragonfly is initially close to a wall in an almost vertical position (see the blue body). The color scale represents time. The hotter the color is (redder), the more towards the end of the maneuver the snapshot is.

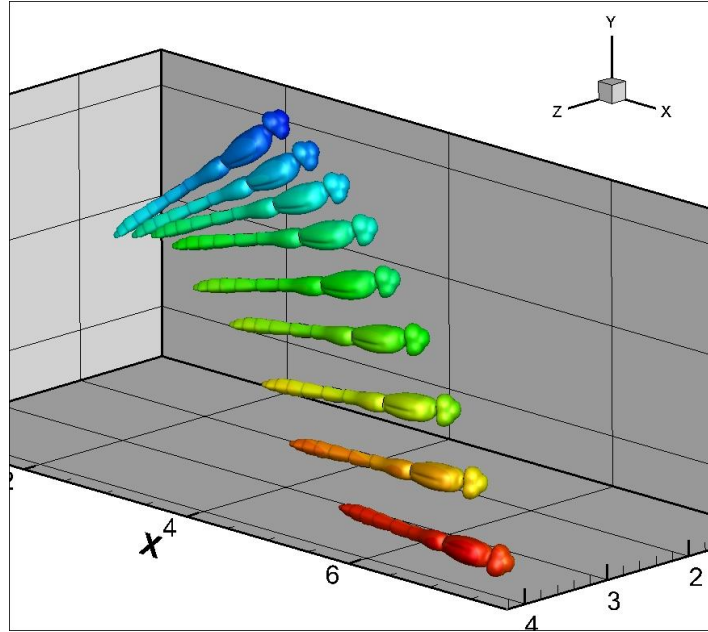


Figure 40 : Test case 1 in time snapshots

Verbally describing a maneuver is close to impossible, but simply said, we can see that the dragonfly is initially at a random orientation, after which it dives backward and rolls to regain a straight heading. There is data before the starting time and after the ending time shown here, but clearly this is the most important part of the maneuver. What can be seen directly is that the body rolls in a negative sense. It must also be clarified that though the centre of mass decreases, it does not mean we may get a good picture about whether the insect is pitching down or pitching up. The three kinematic parameters will be looked at hereafter. Shown below is a table describing three graphs. Each graph compares the results obtained from the manual reconstruction and the auto-reconstruction software. A dimensionless time scale simply implies that the time at which the maneuver starts is represented as '0' and the time at which the maneuver ends is represented as '1'.

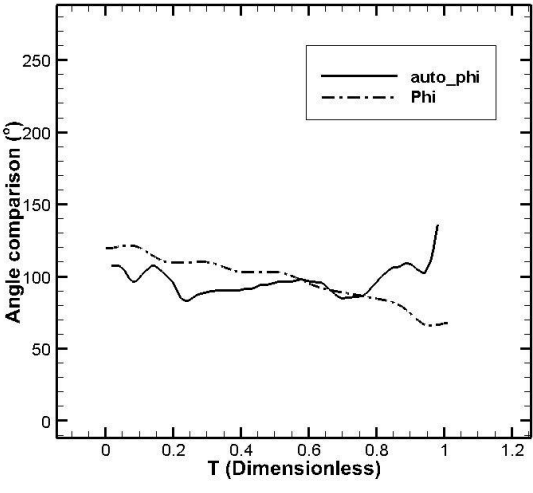
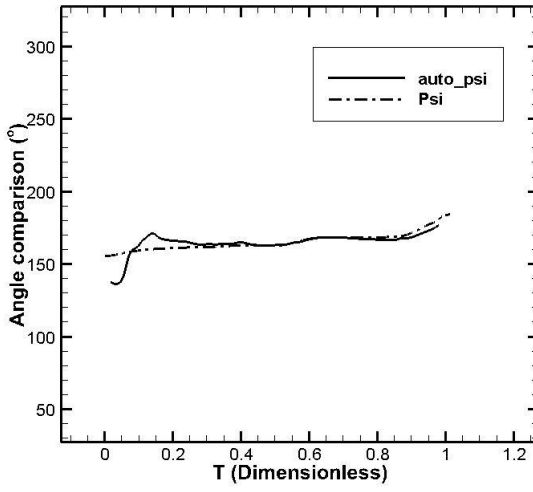
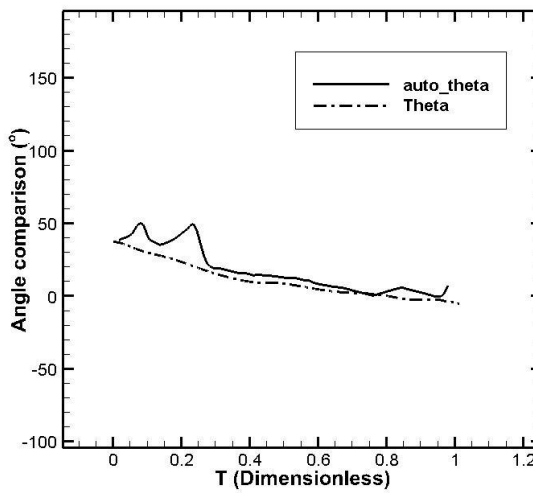
Plot	Explanation
	<p>$\phi(phi) = roll$</p> <p>First of all, the range of variation of both values does not differ by much. The auto-tracking results seem to fluctuate much more than the results of the manual tracking. The roll decreases smoothly from an initial value of 110 to a value as low as 50.</p>
	<p>$\varphi(psi) = yaw$</p> <p>The code generated points that were in very close agreement with the manual tracking code. The error seems to be very small. From $t=0$ to $t=0.2$ however there is a slight fluctuation. This range of peaks corresponds to fluctuation discussed in the roll.</p>
	<p>$\theta(theta) = pitch$</p> <p>We find that the pitch calculated is also in close agreement to the manual tracking results. The initial fluctuation corresponds to the that of the other two angles.</p>

Table 10 : Test case 1 - Plots and explanations

Test Case 2

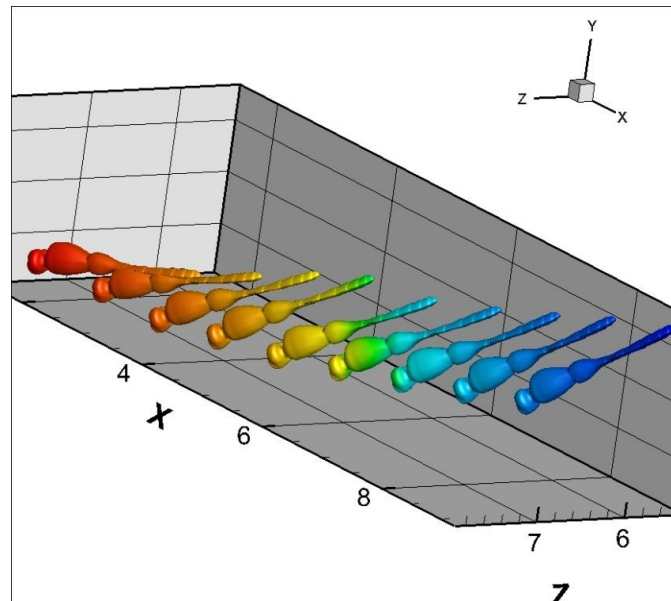


Figure 41 : Test case 2 in snapshots

The dragonfly in test case 2 is seen flying across the filming volume towards a wall. The dragonfly is initially pitched down; it continues in the same angle for a short period of time and then suddenly starts pitching up. Similar to the previous test case, we see that the bodies shown in fig 39 are color coded. Blue or colder colors represent starting of the maneuver and Redder colors represent the end. The body orientation also makes it obvious for us to understand the manner in which the dragonfly is moving. This is a very smooth pitch – up maneuver and does not have any sudden oscillations. Shortly after the portion of the filmed maneuver shown above, the dragonfly hits the wall. But this does not seem to affect the trajectory of the turn before the end, which is very smooth. By inspection we can see that the dragonfly pitches up slowly until the end. We cannot decipher yaw and roll from the video, which says that the variation of these parameters cannot be much.

Plot	Explanation
	<p>$\phi(phi) = roll$</p> <p>Most of the values of roll is in very close to the calculated roll except for the last portion, from time 0.85 onwards. As predicted there is not much of a change in roll angle throughout the maneuver time (T = 0 to 1).</p>
	<p>$\varphi(psi) = yaw$</p> <p>A very steep increase in yaw is seen in the values generated by the manual tracking. This is unrealistic as there is no such apparent jump in the yaw value. Thus in this case, the auto-tracking code generates more reliable results.</p>
	<p>$\theta(theta) = pitch$</p> <p>The pitch calculated by the manual tracking code also shows a similar fluctuation in the initial part of the maneuver, which is not reflected by the results of the autotracking code. Also, the manual code shows the value of pitch to go past the negative line into the positive region. By inspection we can see that the dragonfly body doesn't really reach a position that is parallel to the X-Z plane (see the last red body in fig. 39).</p>

Table 11 : Results of test case 2

As seen in the results, the dragonfly mainly pitches up. The quality of the video is good. Since the insect flies across a relatively small filming volume, the lighting distribution is even. It is not a take-off case and hence there are no external objects, be it light or dark. There are almost no shadows in the videos except for the top view, and this is acceptable as the shadow formed is light and disappears towards the end of the sequence. As all conditions are favorable and the maneuver is simple and smooth, one can expect the results to be in good agreement.

Test Case 3

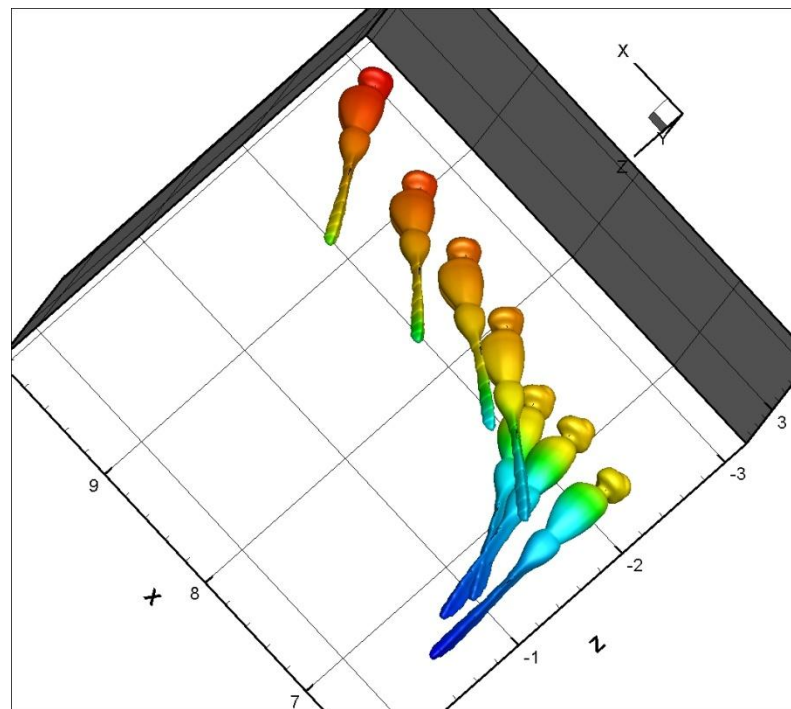


Figure 42 : Test case 2 in snapshots

Shown above is the second case studied. Again the color scheme indicates that the redder or hotter the color is the later the snapshot is taken. The view shown is close to a top

view, but slightly rotated about the y-axis. The dragonfly is initially at rest. It is rolled towards the left side and is ready to take off. Once it leaves the stand on which it was perched, it makes a very sharp left turn and climbs slightly. The insect rolls in unstable looking jerks, trying to stabilize itself. It seems as though the turn taken by the dragonfly is very sharp as it then senses this, and follows by stabilizes itself by pitching down too. It can be said that in the first portion of the maneuver the rotational and translational accelerations and velocities are low. That is, this portion is sluggish. After the quick turn, the dragonfly moves in the positive x direction considerably faster than in the first portion where it is seen taking off. It must be noted that the left turn is completed within two wingbeats. The typical wingbeat frequency is about 30 to 40 times per second. One of the purposes of this study is to find out how such a tight maneuver is possible in such a short amount of time.

The results comparing the three angles, ϕ (roll), ψ (yaw) and θ (pitch) are shown in the table below. As the lighting in this particular video was not of very high quality, noise is added to the images. Now, the presence of external objects in the video such as the stand and the stick used to force the insect to take-off also contribute to results of lower quality. This case shows that the code works sufficiently well in bad lighting and local environmental conditions.

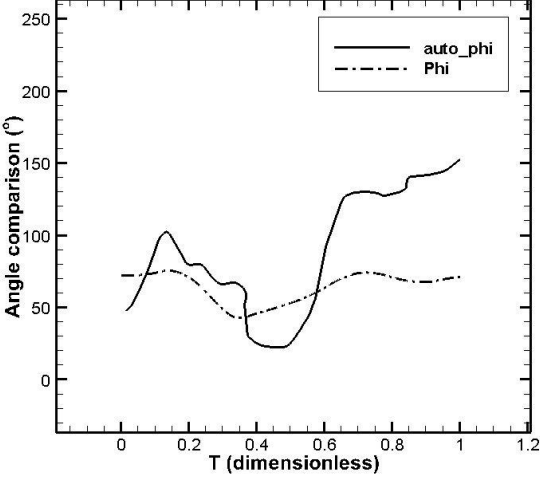
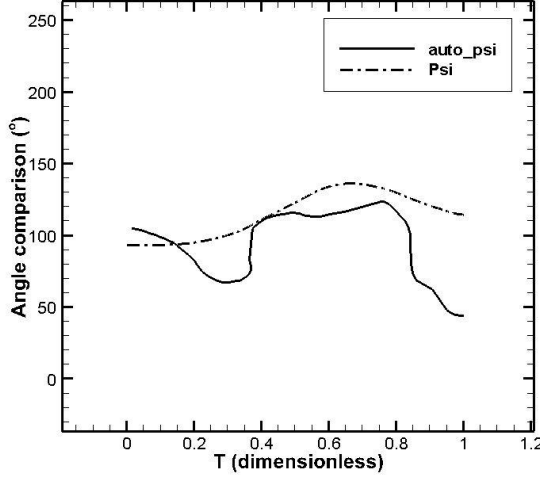
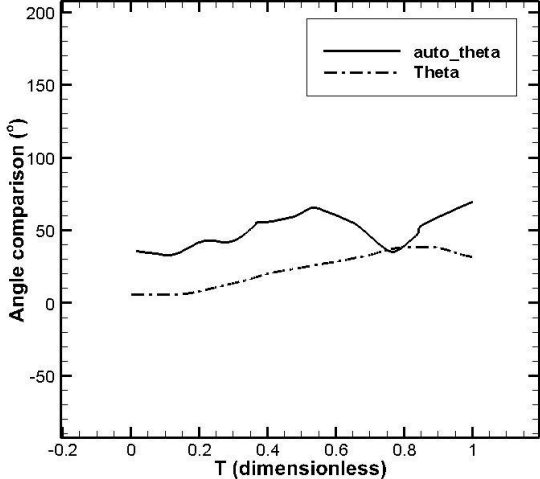
Plot	Explanation
	<p>$\phi(phi) = roll$</p> <p>The range of values does not match in the second portion of the maneuver. It seems like the code fails to capture the correct trend of the body rotations when the insect moves towards a dark object in lower lighting conditions. The initial portions of the graph however, do match well.</p>
	<p>$\varphi(psi) = yaw$</p> <p>Unlike roll, the yaw angle is mostly in the range of angles calculated by the manual reconstruction code. However towards the end, the angle of yaw, which is also related to the roll is influenced by the highly erroneous roll value.</p>
	<p>$\theta(theta) = pitch$</p> <p>The pitch angle calculated agrees with the trend of the manual results, much more than the other angles. In this particular case, the fluctuating roll and yaw angles do not cause any fluctuations in pitch in the same time frame. However, as we saw in the previous case, the angle seems to be shifted above the actual plot that is computed from the manual reconstruction.</p>

Table 12 : Results of test case 3

Other Cases (Plots only)

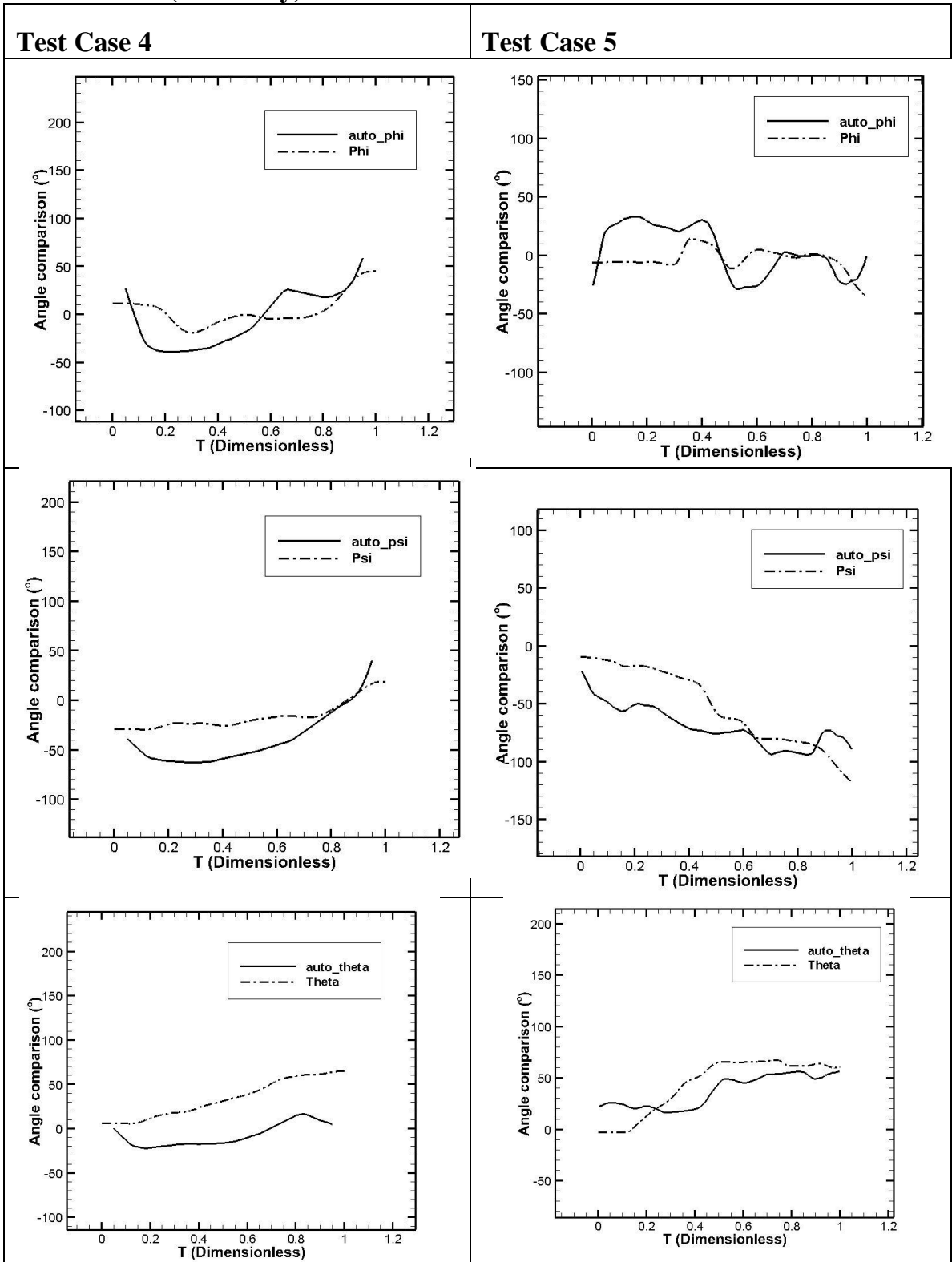


Table 13 : Test Cases 4 and 5

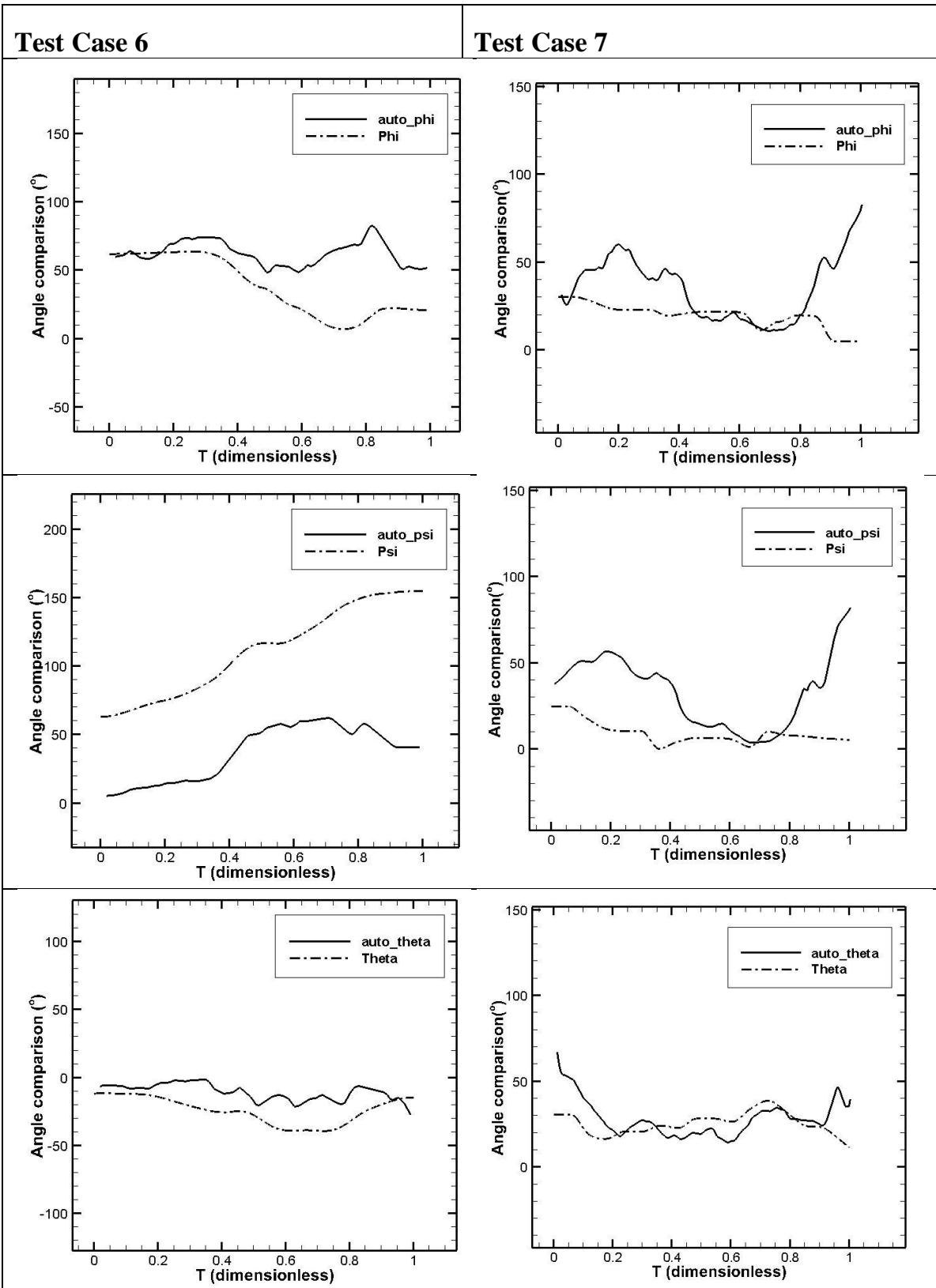


Table 14 : Test Cases 6 and 7

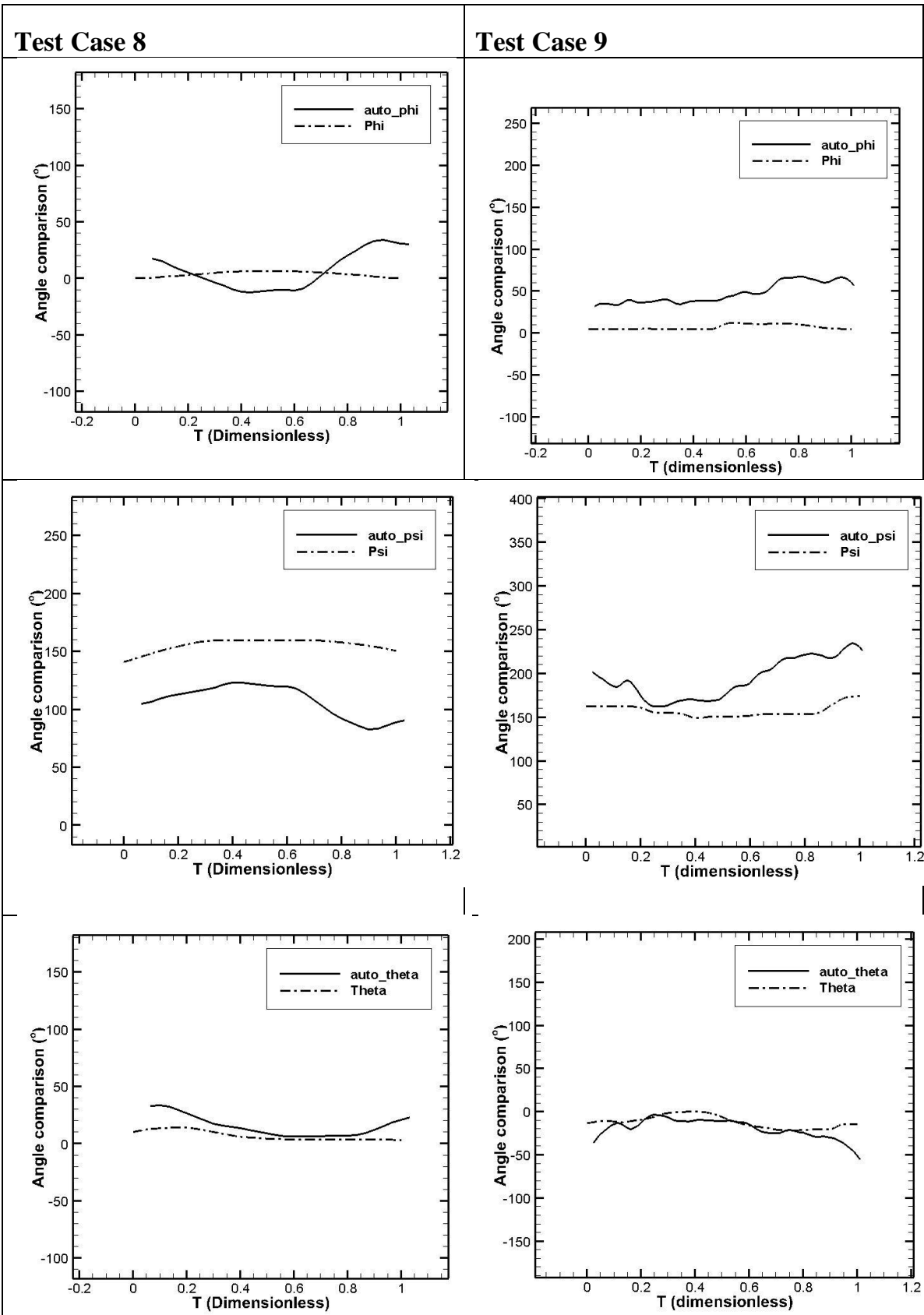


Table 15 : Test Cases 8 and 9

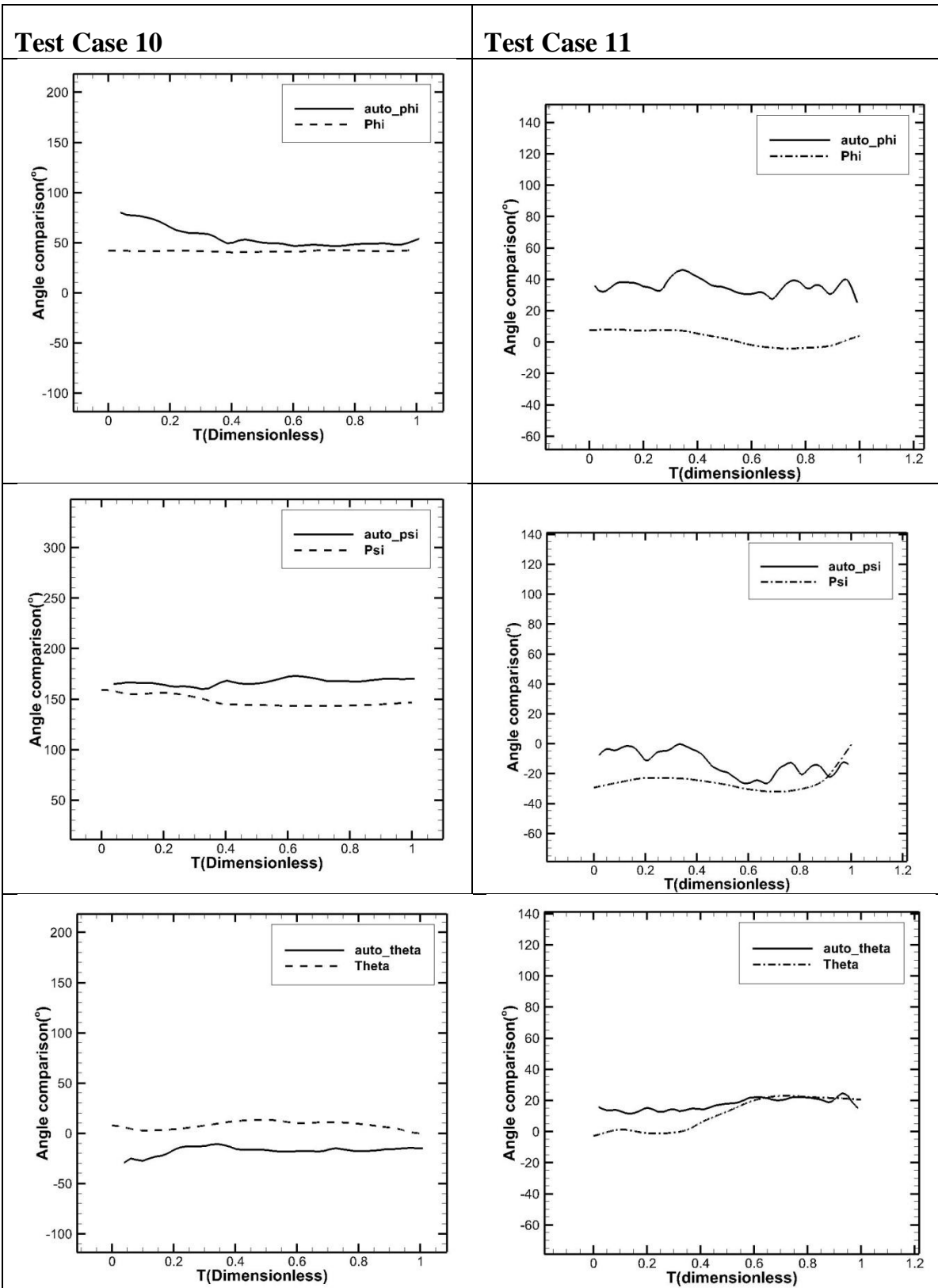


Table 16 : Test Cases 10 and 11

Discussion

- The corresponding outputs from the three cases are mostly in agreement with the values obtained from the manual reconstruction method.
- Three major differences exist between the maneuvers chosen to be discussed. All other cases are similar and may be classified as one of the following types:
 - **Type 1:** Test case 1 shows a maneuver where the lighting is and shadows are moderately dark. There is only one external object present. The auto-tracking code has small errors when compared to the outputs of the manual reconstruction.
 - **Type 2:** Test case 2 shows a case where the auto-tracking code seems to reflect the tendency of the maneuver more closely than the outputs of the manual code. The lighting is very good and there are almost no shadows or external objects.
 - **Type 3:** Test case 3 is a ‘worst-case-scenario’. The lighting is bad, there are multiple shadows of very high intensity and the maneuver is complicated. In addition, there are external objects such as the stand and the probe used to force the insect to take off.
- Results seem to agree with the trend almost always, but the lines are to be shifted in some cases.
- The roll angle measured is almost always inaccurate. This could mean that either of the methods is inaccurate. However, at the onset it seems like the output from the manual tracking code seems unrealistically smooth.

- Using varying n-skip values, which means skipping some frames before processing the next, smoothes the plot by varying amounts. A smoothing algorithm has not been incorporated into the code yet.
- The trend and limits from each method are similar.
- Fluctuations in the angle of roll always seem to influence the values of other angles recorded at the same range of time.
- It is tough to compare the values of roll obtained from both the methods as roll is the most difficult value to obtain by inspection. Although the position of the wing root or appendages gives an estimation of the body roll, it is based completely on the users guess, and nothing else.
- The code also gives sudden peaks in the values of angles. The point at a peak is generally 90, 180 or 270 degrees apart. This is because the code follows an iterative strategy and hence might skip a convergence plane. Of two planes that divide a 3D space into octants, the vector that we are rotating may converge or match with a corresponding plane in any octant. Calculation of the change in centre of gravity is also possible. Shown below is the variation of coordinates in the manual reconstruction method versus the auto reconstruction method.

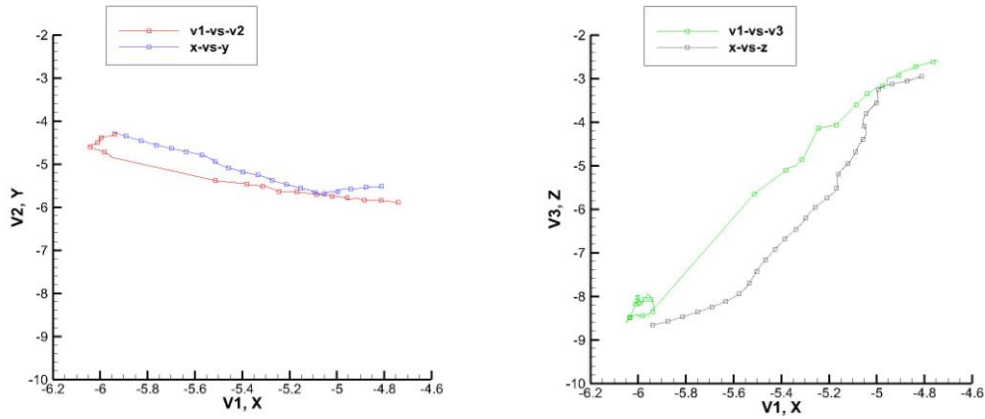


Figure 43 : Comparison of CG's : x-y-z vs v1-v2-v3

In this representation x-y-z is compared with the coordinates of the imaginary 3D space used by the auto-tracking logic which is V1-V2-V3. Two graphs are sufficient to compare all three variables and their change with respect to time. It can be seen that after a transformation, the values of translation also can be made to match the manual tracking output. It should be noted that unlike estimation of the kinematic angles Pitch, Roll and Yaw, CG translation in 3D is bound to be more accurate than the auto-tracking code output.

Similar to the fluctuations in angle, we may also have peaks in the translation values. These are shown in fig. 40. These can be smoothed easily using a plotting software or through a piece of code that involves a filter.

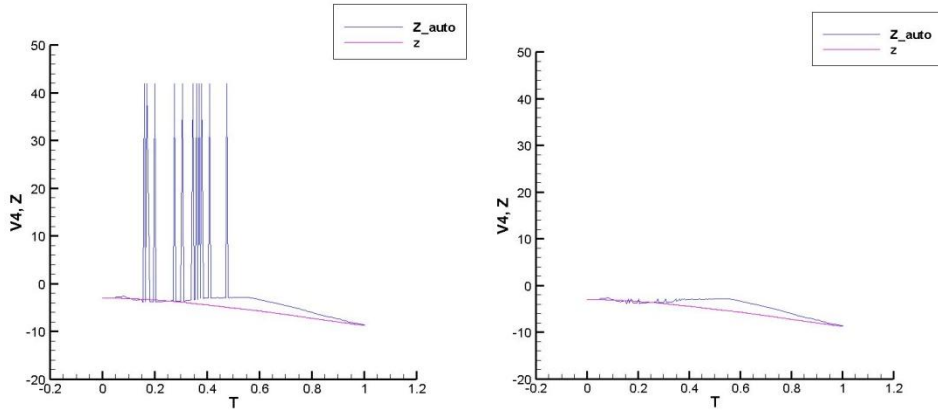


Figure 44 : Demonstration of oscillating or peaking output

- Some graphs though, may not be directly comparable. For instance the graph in fig. 41 compares the value of pitch for one particular case. Though the error is large, we can see that the overall trend has been captured properly. This includes the sudden pitch-up motion followed by a short plateau.

On the left hand side we see that the trend has been captured, however the plot seems to have shifted up by approximately 40°

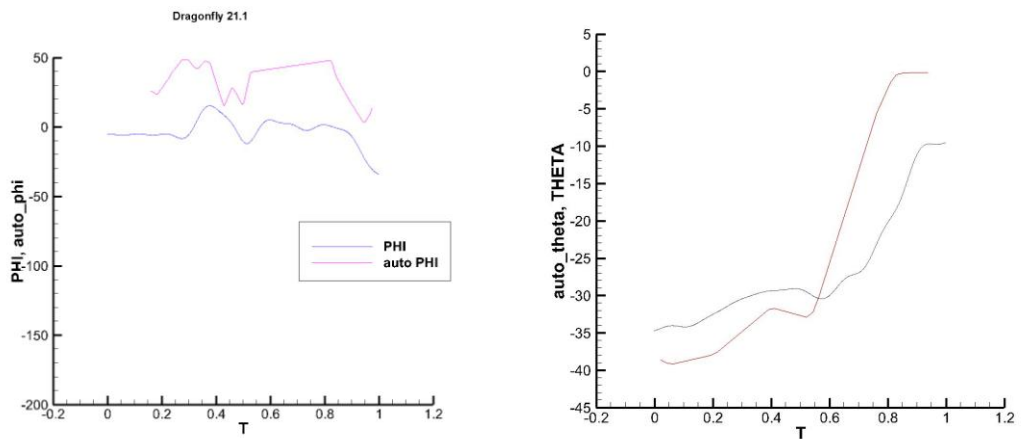


Figure 45 : Other types of errors seen in output

- Also, when lighting is very less, or the presence of shadows is affecting the output, one might see oscillations such as those shown in fig. 42. If the trend is agreeable, the right output may be obtained by using a smoothing filter or averaging.

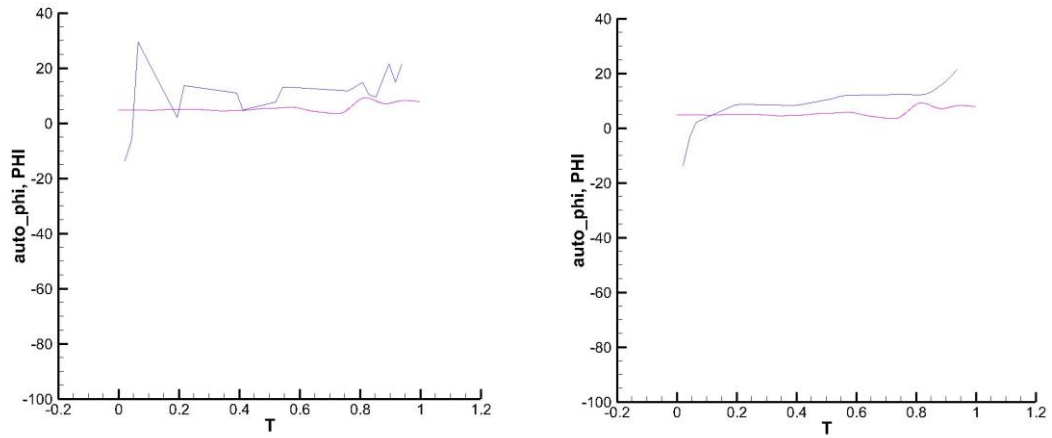


Figure 46 : Effects of smoothing on erroneous output

IV RESULTS

The test cases were selected to be random to show that the code was robust in any kind of maneuver – be it a take-off, turn, or a simply fly-by. However, to be able to use this tool, we need to select one particular kind of maneuver. Due to the existence of several videos of dragonflies roughly performing a left turn, this would be a natural choice to bring about some interesting observations to light.

Classification

Of the 35 available left turn videos, 30 videos were selected as experimental data to be analyzed. These videos may be further classified as follows:

1. Forced Take off – where the dragonfly is forced to take-off. This causes it to follow an unnatural trajectory as its main priority is to escape from the ‘predator’. Thus, this kind of take off is also known as ‘escape take-off’. Two (8) of the 30 videos are escape take-offs.
2. Natural Take off – Here the dragonfly is allowed to take-off on its own. Generally, the dragonfly flies towards one of the light sources. These 22 videos are the ones we are interested in.

There are several ways to classify the 22 natural take-off videos. One possible way is presented below in fig. 45. The numbers represent the ‘identification number of the dragonfly used in that video, as per laboratory format.



Figure 47 : Case availability

These are the cases available for us to analyze. Slow or fast describes roughly how fast the maneuver is completed. Medium, Low and High describes the height the dragonfly finally reaches. A forced take-off happens when the insect is prodded by the rod shown in the fig 46. A Natural take off is when no human initiation is necessary. The blue cases in fig. 45 are Natural whereas the Red cases are forced.

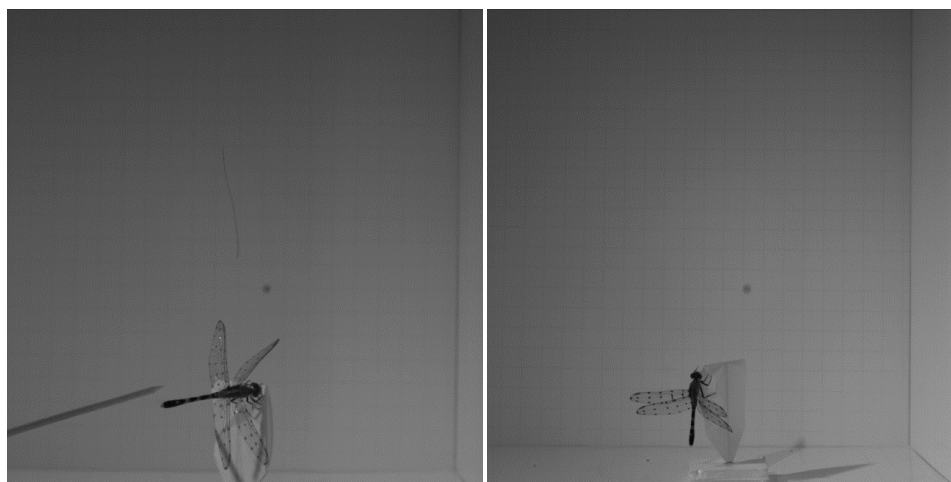


Figure 48 : Forced take-off (Left) vs. Natural take-off (Right)

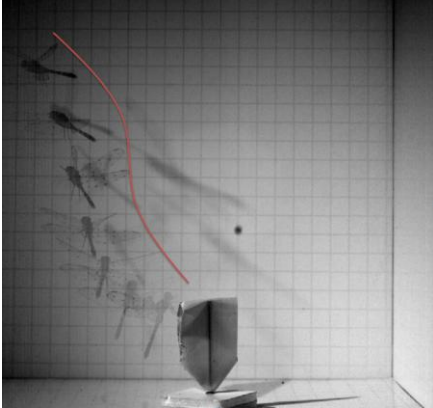
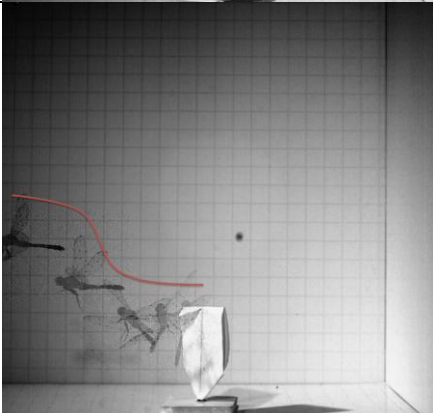
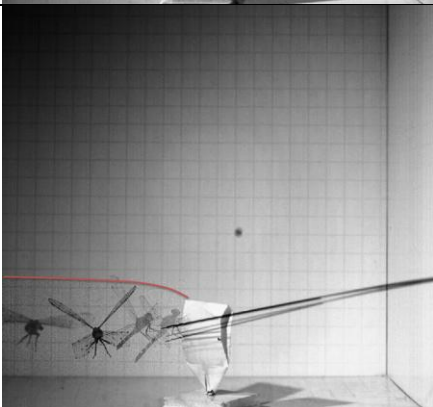
Case Bifurcation

Based on the available cases as discussed above, we may test several important features of this particular 'Left Take-off case' only. Here are the tests that will be carried out:-

1. Slow Take-off vs. Fast Take-off
2. High Take-off vs. Medium Take-off
3. High Take-off vs. Low Take-off
4. Medium Take-off vs. Low Take-off
5. Forced Take-off vs. Natural Take-off
6. Dragonfly 15 Left Take-off (4 cases)
7. Dragonfly 13 Left Take-off (7 cases)
8. Dragonfly 12 Left Take-off (5 cases)
9. Dragonfly 8 Left Take-off (3 cases)

The list above shows all the possible individual experiments that can be performed with the given dataset while aiming to produce some interesting results. The combinations involve the study of either one dragon-fly in several seemingly similar, but different left turn maneuvers, Forced versus Natural take-off, as well as combinations between high medium and low left turns. These appeared in the classification table also. The idea of classifying videos into high, medium and low take offs comes from the follow through observed **after** the dragonfly takes off. For example, a high take off means that generally, the insects gains a lot of altitude by the end of the video sequence. Similar explanations

may be given for low and medium left turns. Examples of these maneuvers are shown below in table 17.

Motion merged image	Description
	<p>High</p> <p>Characterized by very smooth movements with a steady follow through after take-off with slow gain in altitude.</p>
	<p>Medium</p> <p>Generally smooth movements are observed in this kind of take-off. The dragonfly however does not gain as much altitude as the 'high' case.</p>
	<p>Low</p> <p>These are really fast left turns since most of the effort goes into turning, and not gaining altitude.</p>

Classification of left turns pictorially

Validation cases

Some cases did not fit into any of the above ways of describing the take off, but were available for use nevertheless. Some may not have been used due to poor lighting or presence of a large external object in the scene. These cases were used for validation and are listed below.

Dragonfly number	Reason
14.4	Straight take-off
19.1_gust	External Object
19.2	Flyby
20.1	Flyby
20.2	Flyby

Table 17 : Validation cases

Pitch – Roll – Yaw plots

A list of several plots is presented below. These plots include information about the Pitch, Roll and Yaw angles of the body in that particular video sequence. A smoother is incorporated in the code so that the plots do not show unrealistic oscillations. All plots shown below are graphs of these variables for the body in left turn only.

It is easy to observe that yaw decreases and becomes more negative in each of the cases. This is expected as a negative yaw implies left turn, generally speaking. It may be premature to decide by inspection only, that the dragonfly uses one strategy and nothing else for a left turn. Further analysis is needed, and will be presented in a later section.

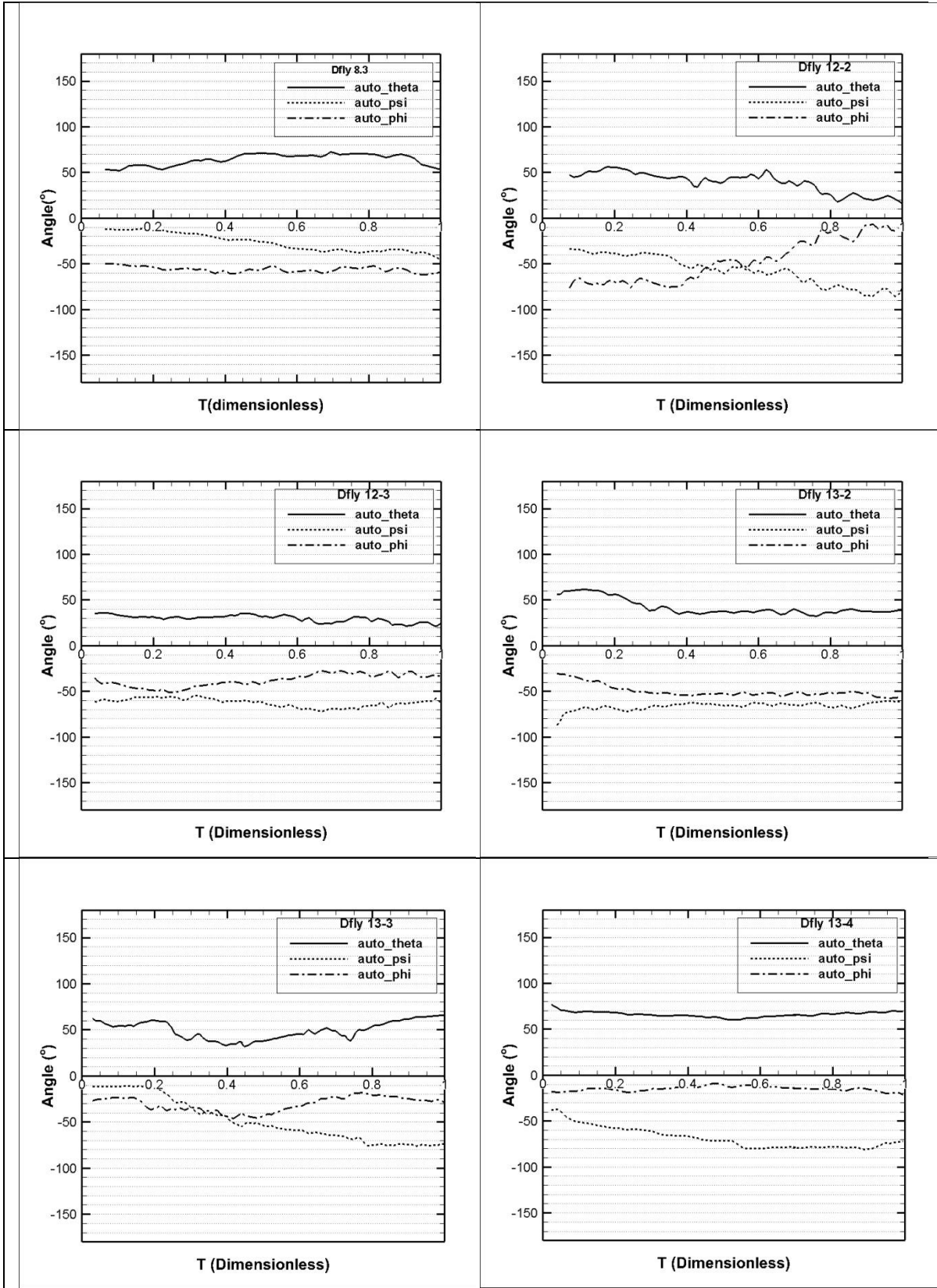
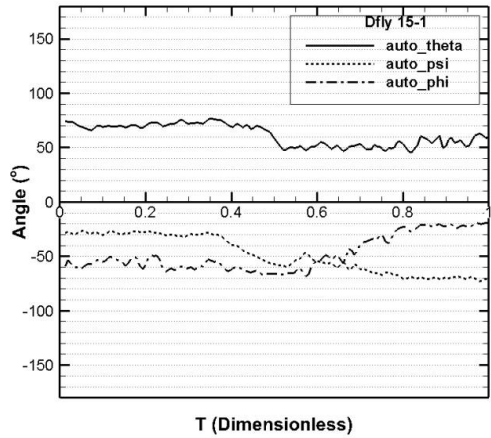
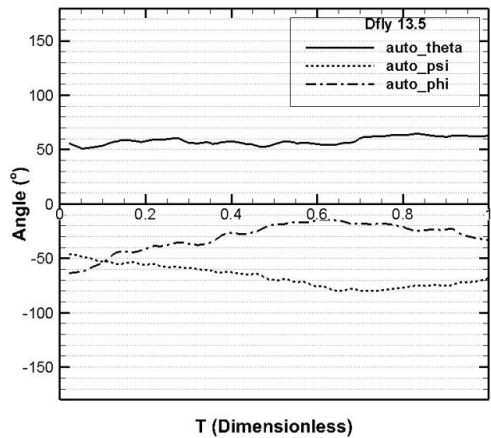


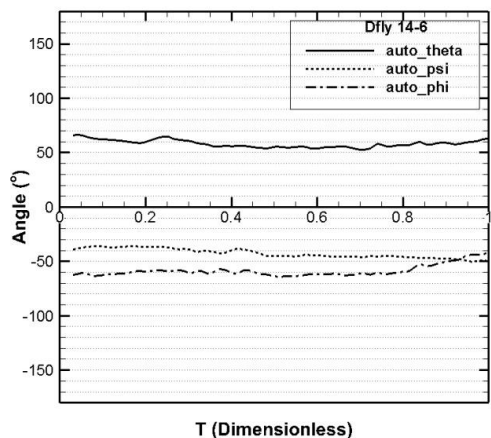
Table 18 : Slow and high, set 1



T (Dimensionless)



T (Dimensionless)



T (Dimensionless)

Table 19 : Slow and high, set 2

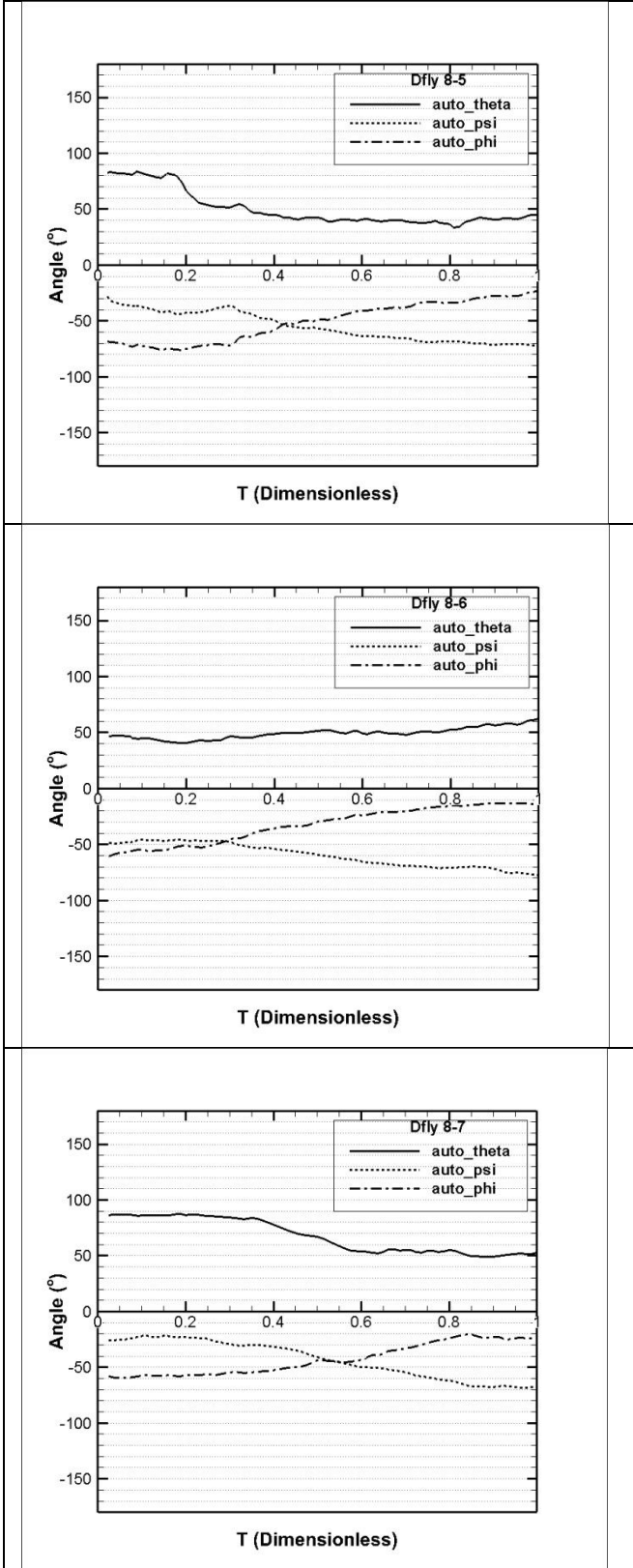


Table 20 : Fast and medium, set 1

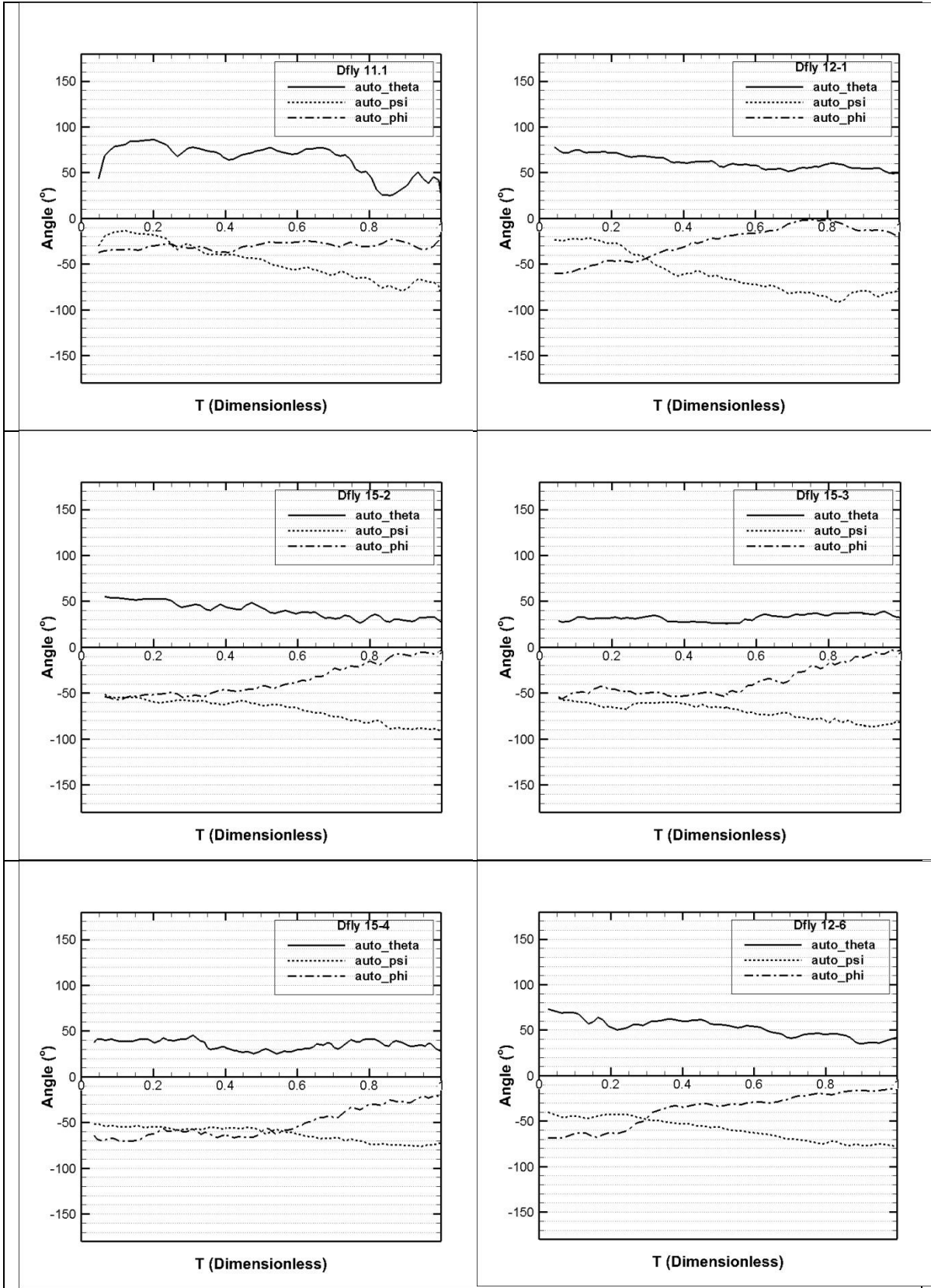


Table 21 : Fast and low, set 1

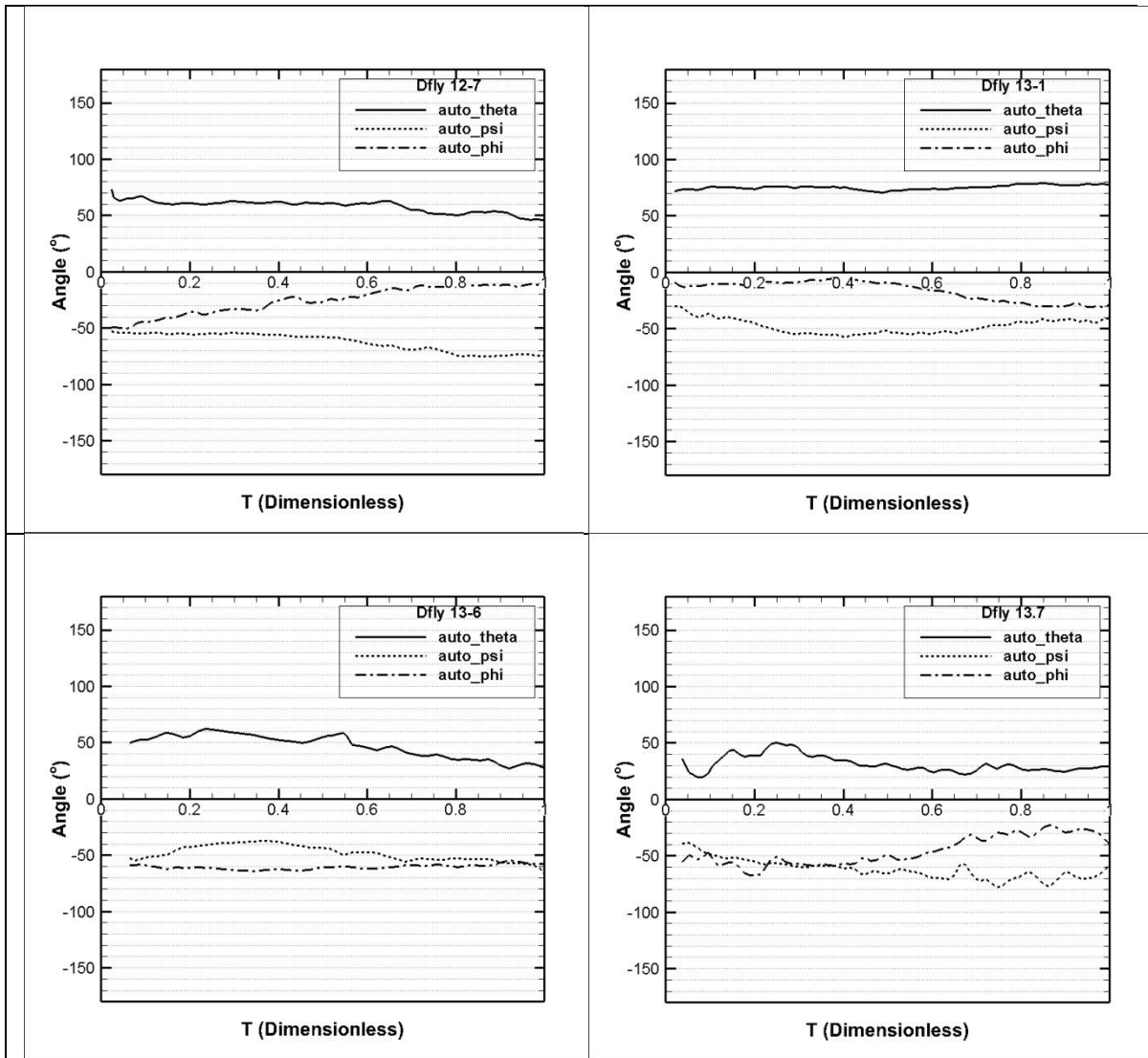


Table 22 : Fast and low, set 2

As a general observation, it is impossible to even surmise by inspection, the approximate value of Pitch, Roll and Yaw. Every turn is a result of a different strategy, or at least, it is safer to assume so. The dragonfly maneuvers showcased above have been provided enough follow through time while processing the video so that the insect is completely, or almost completely reoriented after the take-off.

There is now an abundance of data. However, completely decoding all the mysteries of insect flight is not our goal. It was suggested that any further analysis be focused to an extent that only a few variables are allowed to change. This is discussed in the following section.

Dragonfly 8 – an experiment

In addition to the pitch roll angle, there is also information on the body and tail tip vectors of the body. The flexible tail may prove to be very useful in maneuvers. Intuitively, the insect would attempt to make a turn with the least amount of effort. Extreme amounts of deflection in the tail show that this might be an important factor in the turn being executed. Several animals and fish can be seen using the posterior part of the body to an advantage, for balance, control or posture. It is only natural to think that after years of evolution, a long tail exists for a reason in the Odonata class. What remains to be done is the quantification of this idea. A simple hypothesis is made in this stage. The tail must naturally bend towards the center of the instantaneous axis of rotation, if not always, at least during the most important section of the turn, or during turn initiation.

We will be comparing 4 left turn maneuvers of the very same dragonfly, filmed on the same day in close succession. The dragonflies used in this test are 8.3, 8.5, 8.6 and 8.7. Two quantities are used to quantify the relationship between the tail-thorax angle and the instantaneous axis of rotation – ‘Instacross’ magnitude (defined as the magnitude of the unit vector obtained as a result of the cross product of two vectors v_1 and v_2) where V_1 is the cross product of the tail and the thorax to head vector and V_2 is the instantaneous rotation axis.

It is intuitive to think that the dragonfly must bend its tail towards the centre if a turn is being negotiated. It can also be argued that such a configuration decreases the moment of inertia about the v_1 axis and shifts the CG outside the body (and to the left) to assist in roll (Dong et al, AIAA compliant tail MAV). But how do we relate these quantities? The cross product of this quantity must be zero if v_1 is normal to v_2 and one if they are parallel. As mentioned before, we are not talking about ideal systems. Thus satisfaction of this idea only helps in a better understanding of how a turn is executed, and will bring us closer to understanding insect flight.

The figure below shows the vectors v_1 and v_2 as well as the body rotating about an imaginary axis (blue circle).

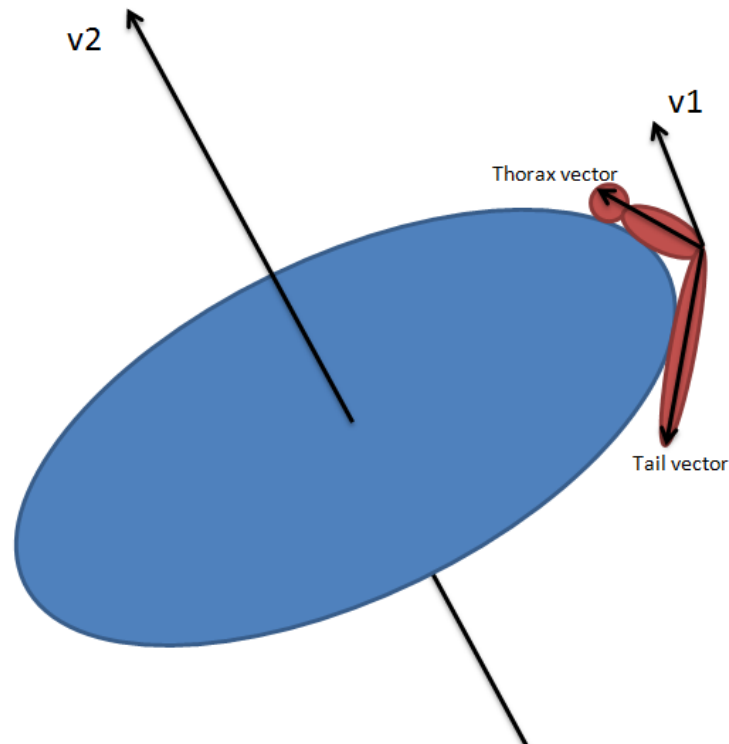


Figure 49: Instacross magnitude from the v_1 and v_2 vectors shown

Observations

The data from the additional piece of code that relates the trajectory of the insect with the tail vector is to be analyzed along with the Pitch, Roll and Yaw angles shown earlier. The afore mentioned Dragonfly cases are discussed below along with the relevant plots.

Dragonfly 8.5

DFLY 8.5 shows a very interesting take off. It first releases its hold from the stand and starts the actual turn during the downstroke of the second wing-beat. However, at this point of time, the instantaneous circle plane's normal is close to perpendicular to the magnitude vector. The second part of the maneuver is a simple pitch up where the tail seems to be kept at an upward angle. Thus we expect that the magnitude is close to one (1) in the beginning, and zero (0) towards the end.

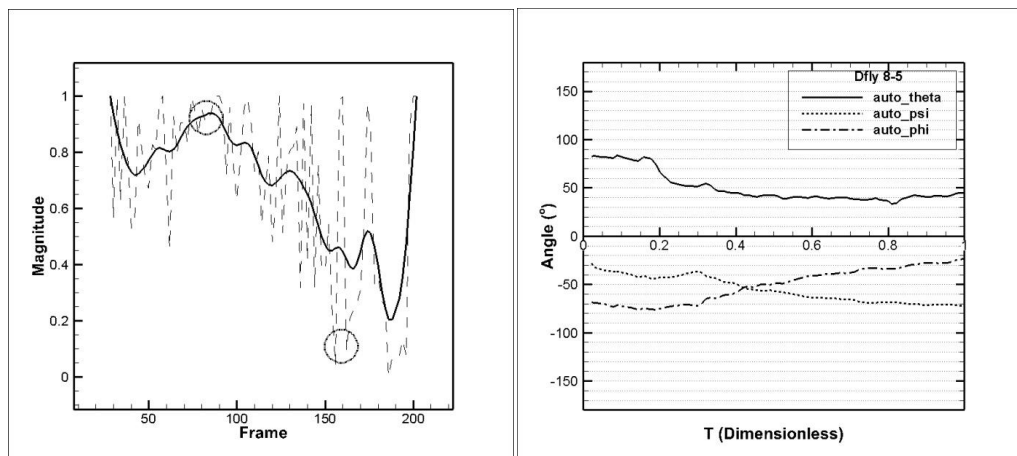


Figure 50 : Instacross plot along with P-R-Y, DFLY 8.5

Dragonfly 8.3

DFLY 8-3 Is relatively simpler to explain. The points marked with circles show clearly where the two vectors are parallel. The dragonfly takes off by a powerful downstroke while maintaining the initial orientation, rolls quickly clockwise just after the first upstroke. This clockwise roll is seen along with a clear leftward slipping. This is shown in point '1'. The point marked '2' is roughly where the dragonfly yaws to the left while keeping an intuitively obvious tail orientation towards the left. This is followed by a slow ascent while pitching up.

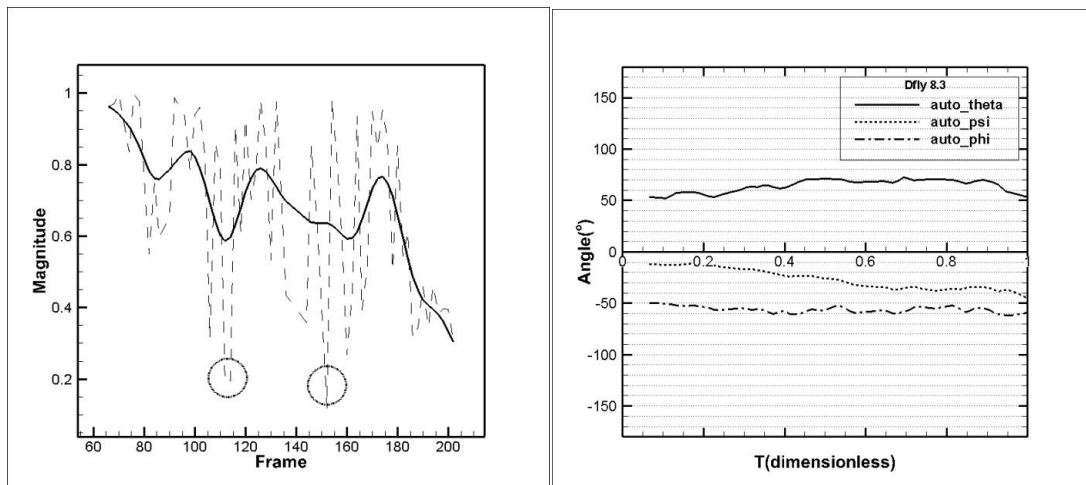


Figure 51 : Instacross magnitude with P-R-Y angles, DFLY 8.3

Dragonfly 8.6

DFLY 8.6 shows a different strategy as compared to DFLY 8.3 above. Here, the first downstroke is used to increase yaw first, while maintaining the initial orientation. During the second down stroke the dragonfly increases its yaw further. The clockwise roll to re orient itself is done during the second upstroke and third downstroke. The rest of the motion is a constant ascent. Thus most of the values must be close to zero.

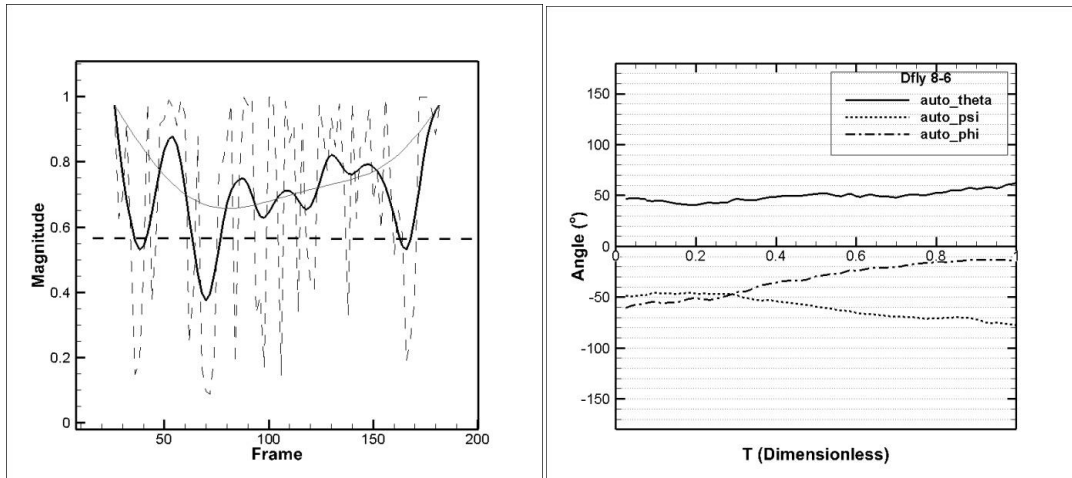


Figure 52 : Instacross magnitude with P-R-Y angles, DFLY 8.6

Dragonfly 8.7

The turn seen in DFLY 8.7 seems like a blended combination of the above turns. There are no clearly demarcated sections like that described in the above cases. The later part is only an ascent using the tail up configuration as usual. As seen from the r_o curve, the relatively straight path (very high value) gradually becomes a tight curve.

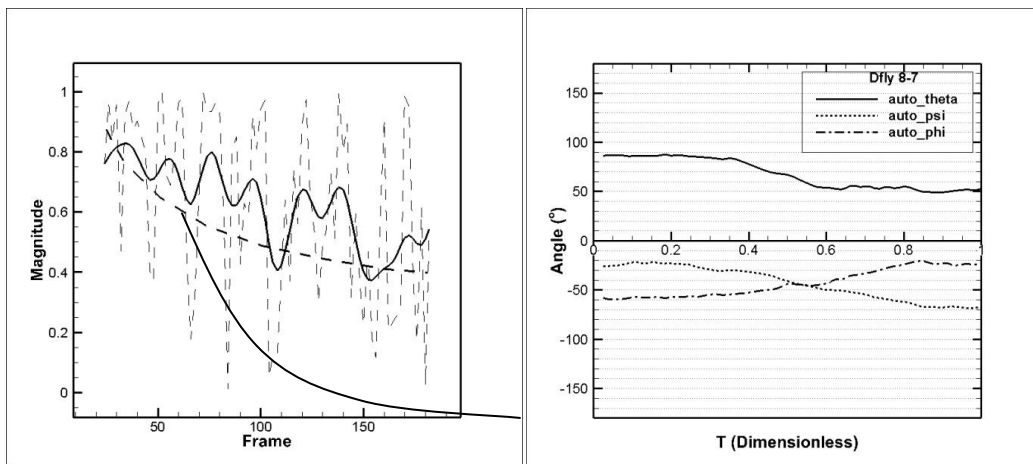


Figure 53 :Instacross magnitude with P-R-Y angles, DFLY 8.7

Other interesting maneuvers

Dragonfly 11.1

A very typical vertical initial pose is seen. Just after the dragonfly releases its hold from the take-off stand, its body falls significantly. To make up for this sudden loss in altitude, the dragonfly attempts to pitch up as well as turn leftwards. Thus, although the trajectory is a left turn, the tail is seen bobbing upwards at the end of each forewing downstroke and the instacross magnitude is expected to be close to 1. In addition there is a sudden and purposeful pitch action downwards towards the end of the video sequence. This is also seen in the Pitch-Roll-Yaw graph. The later part of the video shows a completely reoriented dragonfly that is trying to regain altitude, therefore the instacross magnitude is expected to be close to zero.

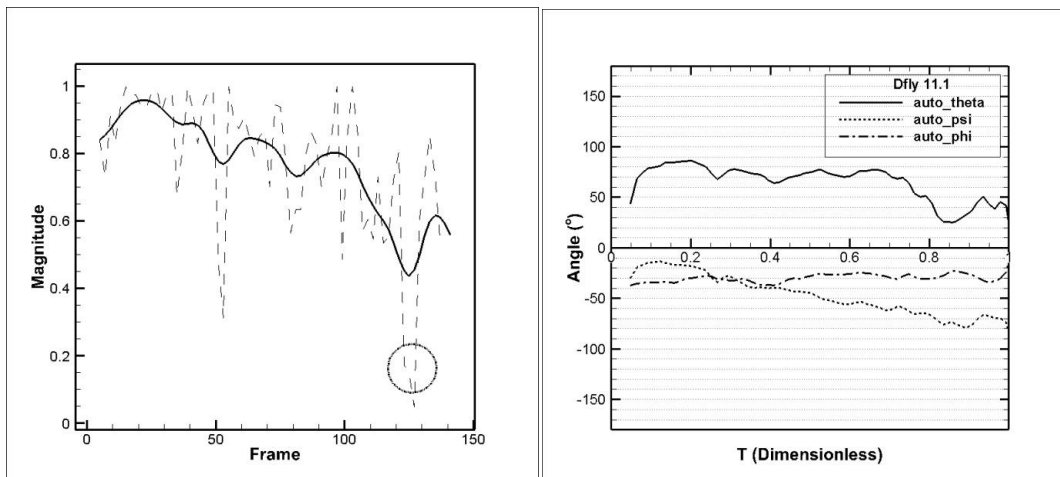


Figure 54 : Instacross magnitude with P-R-Y angles, DFLY 11.1

Dragonfly 12.6

This dragonfly exhibits a very unique left turn, and is also part of a separate wing damage experiment. The dragonfly directly yaws to the left from the moment it leaves the take-

off stand. Here we expect the magnitude to be close to zero as the tail bends towards the turn. For a very short amount of time, after the first downstroke, the maximum deflection of the tail towards the turn is seen. So, we expect a low value for a very short time, sometime after the downstroke (close to 50th frame). The dragonfly does not reorient itself completely as yet. At the end of the second downstroke (close to 100th frame) the dragonfly quickly rolls to the left and pitches up to continue the left turn trajectory. After this the dragonfly reorients itself completely.

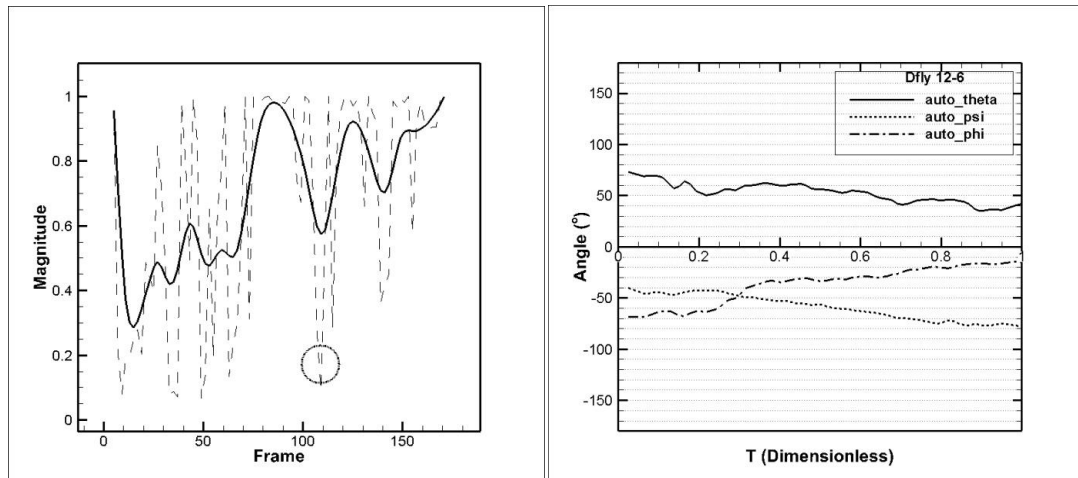


Figure 55 : Instacross magnitude with P-R-Y angles, DFLY 8.7

Conclusions

The same dragonfly, filmed on the same day with the same lighting conditions thus shows four different ways of executing a left turn. This left turn was quantified using the instantaneous vector and 'instacross' magnitude. Also it may not be concluded that the tail movement is categorically passive, or active. In some cases, an active turn may be followed by passive damping and so on.

A note on Longitudinal Static Stability

Comparing the dragonfly to a conventional aircraft is not at all possible, since they operate in different regimes and environments. However, similar explanations may be offered at this stage. The longitudinal stability of an aircraft refers to the aircraft's stability in the pitching plane(22 1975). In a stable aircraft, a small change in angle of attack will cause a pitching moment that restores the plane to equilibrium pitching angle. For an aircraft, the force and corresponding moment caused by the horizontal stabilizer is responsible, or at least plays a very important role in aircraft stability (in the longitudinal or pitching plane). A dragonfly does not have such a stabilizer (see fig. 52) that can make use of aerodynamic forces. Stability in the other directions (Yaw and Roll) may be attributed to forces transmitted to the body from the wings. However, since the wing roots, which are the only source of transfer of flapping flight-forces to the body (here, the fuselage), are located a small distance from each other (about 2.94 mm), the moment contributed in the longitudinal direction may be significant, only if the forces are very high. But this is not practical.

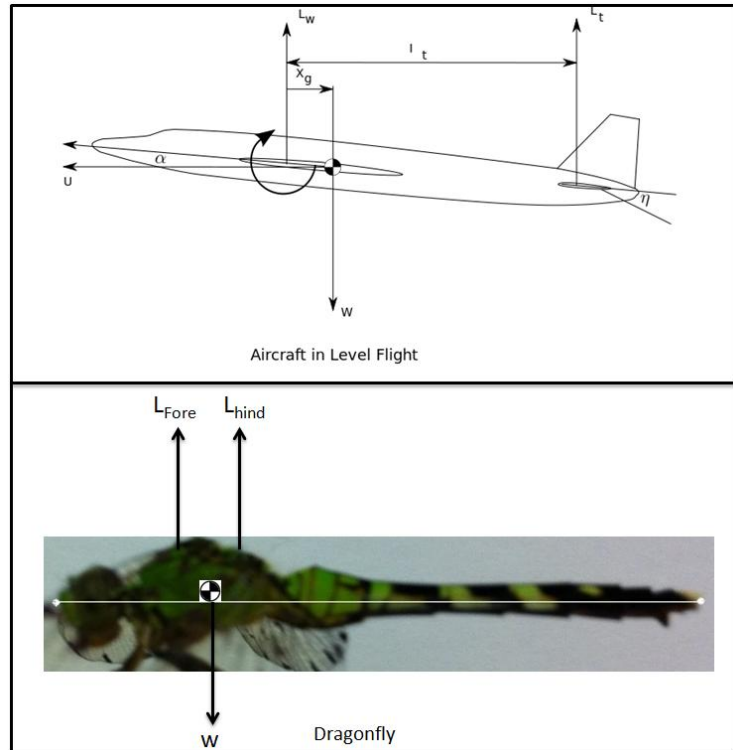


Figure 56 : Conventional aircraft vs a Dragonfly

In fig. 52, ‘ w ’ on both the dragonfly and the aircraft is where the gravity force acts. This varies erratically in the case of a dragonfly as shown in the ‘instacross’ magnitude plots. ‘ L_{fore} ’ and ‘ L_{Hind} ’ are forces transmitted from the center of pressure of the corresponding wing pair to the wing roots (marked as red circles in fig. 53). The resultant forces due to the aerodynamics of each flapping wing act at the center of pressure of that wing. This acts at a distance from the wing root, and hence has a corresponding moment. ‘Stroke amplitude’ may be defined as the angle swept by the wing from two extreme positions, say downstroke to upstroke, in a single wingbeat as seen in the projected top view. The dragonfly does not (at least in the videos observed) pull its wings far enough to get close to the tail, which is to say that the stroke amplitude much lesser than 180° when measured from the body axis (as opposed to that seen in butterfly flight). Thus, the moment caused

by these aerodynamic forces in the longitudinal direction is much lesser than the moments in the other two directions. A more detailed analysis is required to understand the complete picture.

It is logical then, to conclude that the tail must have a significant effect on the stability of the dragonfly during flight. Even though the tail is light, it is long enough to cause significant changes in the center of gravity. The centre of mass of the dragonfly is calculated by measuring the weight of separate sections such as the head, thorax and abdomen. Fig. 54 shows the effect of the angle that the tail makes (in any plane, i.e. about any axis) on the centre of gravity. For perfect static stability, the centre of pressure must coincide with the centre of gravity. In other words, the center of gravity must be as close as possible to the resultant of the forces transferred by the wings to the body (at the wing roots).

Dynamic stability may be inherent in long tailed insects (Dudley 2002), and external perturbations may be damped out by actively or passively holding the tail in certain positions.

Figure 53 also shows the data used in the calculation of center of mass. The head, thorax and tail were separated as three components that contributed to center of mass. The effect of angle of the tail on center of mass is shown in fig. 54. The dashed box represents the area between the wing roots.

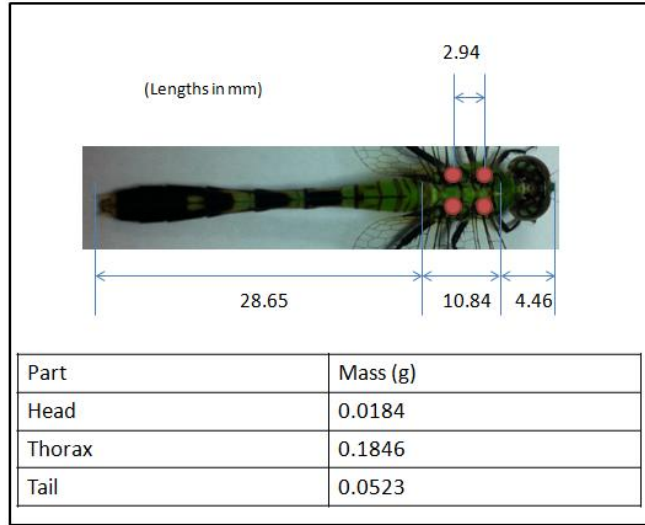


Figure 57 : Calculation of Centre of gravity and location of wing roots

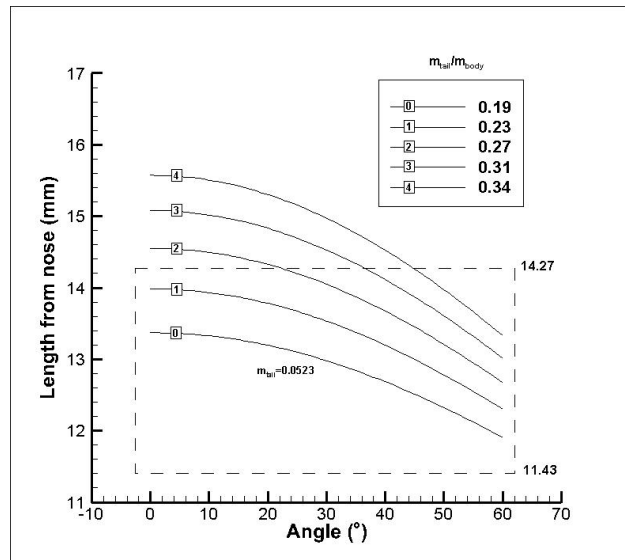


Figure 58 : Effect of flexible tail on Centre of Gravity

Summary

A robust image processing tool was developed to automatically track the Pitch, Roll and Yaw angles of a dragonfly in free flight. Additional information such as the trajectory of the center of mass, as well as the relationship of the continuously changing tail vector angle with the turning radius was presented. The videos recorded and analyzed were categorized into several types. One of these sub-categories was studied in detail (Dragonfly 8, Take-off followed by left turn).

Future Work

Even though there are several directions to explore with just information from the body, explanations in all these directions would be incomplete without involving kinematics of the four wings. A code was written to simultaneously track several points on the wing by using the Lucas Kanade Pyramidal search algorithm. The code in this preliminary stage was used to track features on a butterfly and a cicada in free flight. Shown below is the automatic, multiple point tracking algorithm at work. The code successfully tracks appendages such as antennae and legs, points on the steadily moving body, as well as points on the fast moving wings (like wing tips of fore- and hind-wings).

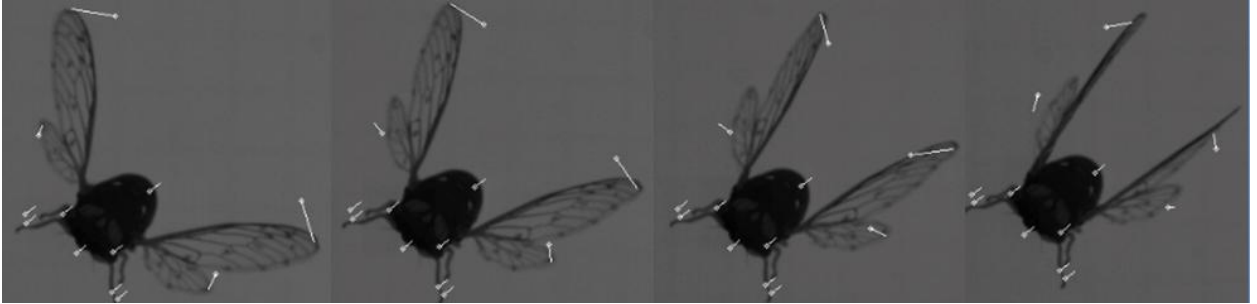


Figure 59 : Multiple point tracking, Cicada

Also, trajectory of a particular point may be tracked throughout the video sequence as shown below.



Figure 60 : Multiple point tracking, Butterfly

It is very important to obtain the wing kinematic parameters along with the body information for a complete understanding of a maneuver. Automatic body reconstruction only naturally leads to future work on wing kinematics study. Also, the quality of the images may be improved by changing the lighting set-up as explained earlier.

Bibliography

1. Non-Journal references

Canny Edge Detector. March 9, 2012. http://en.wikipedia.org/wiki/Canny_edge_detector.

Dudley, R. *The Biomechanics of Insect Flight: Form, Function, Evolution*. 2002.

http://si.edu/Encyclopedia_SI/nmnh/buginfo/insflight.htm.

2. Tyson L Hedrick. "Software Techniques For Two and Three Dimensional Kinematic Measurements of Biological and Biomimetic Systems." *Bioinspiration and Biomimetics*, 2008.

3. S.A. Combes & T.L Daniel. "Flexural stiffness in insect wings I. Scaling and the influence of wing venation." *Journal of Experimental Biology*, 2003: 2979-2987.

4. Ebraheem I Fontaine, Francisco Zabala, Michael H. Dickenson, Joel W. Burdick. "Wing and Body Motion During Flight Initiation in *Drosophila* Revealed By Automated Visual Tracking." *Journal of Experimental Biology*, 2009: 1307-1323.

5. Chauncey F. Graetzel, Steven N. Fry, and Bradley J. Nelson. "A 6000 Hz Computer Vision System for Real Time Wing Beat Analysis of *Drosophila*." *Neuroinformatics conference, Zurich*, 2003.

6. George V. Lauder, Peter G.A Madden. "Advances in Comparative Physiology from High Speed Imaging of Animal and Fluid Motion." *Annual review of Physiol*, 07 2008: 143-163.

7. Yanpeng Liu, Mao Sun. "Wing kinematics measurement and aerodynamics of hovering droneflies." *Journal of Experimental Biology*, 2008: 2014-2025.

8. Degiang Song, Hao Wang, Lijiang Zeng, Chunyong Yin. "Measuring the camber deformation of a dragonfly wing using projected comb fringe." *Review of Scientific Instruments*, 2011.

9. Shigeru Sunada, Deqiang Song, Xiannan Meng, Hao Wang, Lijiang Zeng, Keiji Kawachi. "Optical Measurement of the Deformation, Motion, and Generated Force of the Wings of a Moth, *Mythimna Separata*(Walker)." *JSME International Journal*, 2002: No.4 Vol. 2.

10. Simon M Walker, Adrian L.R Thomas, Graham K Taylor. " Photogrammetric reconstruction of high-resolution surface topographies and deformable wing kinematics of tethered locusts and free-flying hoverflies." *Journal of Royal Society Publishing*, 2008: 1098.

11. Simon M Walker, Adrian L.R Thomas and Graham K Taylor. "Deformable wing kinematics in the desert locust: how and why do camber, twist and topography vary through the stroke?" *Journal of Royal Society Interface*, 2008: 435.

12. Hao Wang, Lijiang Zeng, Hao Liu, Chunyong Yin. "Measuring wing kinematics, flight trajectory and body attitude during forward flight and turning maneuvers in dragonflies." *Journal of Experimental Biology*, 2002: 745 - 757.
13. Lijiang Zeng, Hirokazu Matsumoto, Keiji Kawachi. "A fringe shadow method for measuring flapping angle and torsional angle of a dragonfly wing." *Measurement Science and Technology*, 1996: 776-781.
14. Alexander P. Wilmott, Charles P. Ellington. "Measuring the Angle of Attack of Beating insect Wings : Robust Three Dimensional Reconstruction from Two Dimensional Images." *Journal of Experimental Biology*, 1997: 2693-2704.
15. M L Liu, K H Wong. *Pose Estimation Using Four Corresponding Points*. Hong Kong: Chinese University of Hong Kong, Department of Computer Science, 1998.
16. *Haar-like features*. March 30, 2012. http://en.wikipedia.org/wiki/Haar-like_features.
17. Daniel F Dementhon, Larry S Davis. *Model-Based object pose in 25 lines of code*. Computer vision laboratory, University of Maryland, 2003.
18. Viola, Jones. "Rapid Object Detection Using Boosted Cascade of Simple Features." *Pattern Recognition*, 2001.
19. Alexander Kuranov, Rainer Lienhart, Vadim Pisarevsky. *Emperical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection*. MRL, 2002.
20. Haibo Dong, Christopher Koehler, Zongxian Liang, Hui Wan, Zach Gaston. "An Integrated Analysis of a Dragonfly in Free Flight." *AIAA*, 2010: 4390.
21. Leif Ristroph, Gordon J. Berman, Attila J. Bergou, Z. Jane Wang and Itai Cohen. "Automated Hull Reconstruction Motion Tracking (HRMT) Applied to Sideways Maneuvers of Free-flying Insects." *Journal of Experimental Biology*, 2009: 1324-1335.
22. L.J. Clancy. *Aerodynamics, Chapter 16*. London: Pitman Publishing ltd., 1975.

Appendix 1 – Code

Autotracking

```
#include "stdafx.h"
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#define GLUT_DISABLE_ATEXIT_HACK
#ifdef __APPLE__
#include <OpenGL/OpenGL.h>
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#include <GL/freeglut.h>
#endif
using namespace std;
using namespace cv;
char basecam1[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.1_C001H001S0001\\Camera
No.1_C001H001S0001000";
char basecam3[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.2_C002H001S0001\\Camera
No.2_C002H001S0001000";
char basecam2[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.3_C003H001S0001\\Camera
No.3_C003H001S0001000";
char ext[10]="001.tif";
char num[10];
char temp10[10]="00";
char temp10bi[10]="00";
char temp100[10]="0";
char temp100bi[10]="0";
float GlobalBodyLength=0;
int Nmax=660;
double xforCOM=0,yforCOM=0,zforCOM=0;
int writeall=0;
//
CvMemStorage* g_storage=NULL;
CvMemStorage* g_storage1=NULL;
CvMemStorage* g_storage2=NULL;
CvMemStorage* g_storage3=NULL;
CvMemStorage* dp_storage=NULL;
CvSeq* contours=0;
CvSeq* contours1=0;
CvSeq* contours2=0;
CvSeq* contours3=0;
CvSeq* quad1=0;
CvSeq* tri1=0;
CvSeq* bi1=0;
CvSeq* quad2=0;
CvSeq* tri2=0;
CvSeq* bi2=0;
CvSeq* quad3=0;
CvSeq* tri3=0;
CvSeq* bi3=0;
CvSeq* dps=0;
CvPoint pdps;
```

```

CvPoint dummyhead;
CvPoint dummytail;
CvPoint dummyroll;
CvPoint exchange;
int choice=0;
int sub_choice=0;
CvPoint dummy1;
CvPoint dummy2;
CvPoint dummy3;
CvPoint dummy4;
CvPoint dummy5;
ofstream A1;
ofstream fkpc;
ofstream cross;
ofstream COM;
double testmag=0;
int idps=0;
int plot=0;
//
CvPoint plot_old_pitch=cvPoint(0,0);
CvPoint plot_new_pitch=cvPoint(0,0);
CvPoint plot_old_roll=cvPoint(0,0);
CvPoint plot_new_roll=cvPoint(0,0);
CvPoint plot_old_yaw=cvPoint(0,0);
CvPoint plot_new_yaw=cvPoint(0,0);
//
char base1cam1[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.1_C001H001S0001000";
char base1cam3[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.2_C002H001S0001000";
char base1cam2[190]="C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.3_C003H001S0001000";
CvRect bdirect = cvRect(0,0,0,0);
CvScalar red=CV_RGB(250,0,0);
CvScalar blue=CV_RGB(0,0,250);
CvScalar green=CV_RGB(0,250,0);
CvScalar yellow=CV_RGB(0,250,250);
CvScalar colour1=CV_RGB(0,200,250);
CvScalar colour2=CV_RGB(150,50,100);
CvScalar colour3=CV_RGB(100,100,200);
CvScalar colour4=CV_RGB(125,125,50);
IplImage* imgCam1=NULL;
IplImage* imgCam2=NULL;
IplImage* imgCam3=NULL;
IplImage* imgCam1small=NULL;
IplImage* imgCam2small=NULL;
IplImage* imgCam3small=NULL;
IplImage* imgCam1smallc=NULL;
IplImage* imgCam2smallc=NULL;
IplImage* imgCam3smallc=NULL;
IplImage* PiP=NULL;
IplImage* im1clone=NULL;
IplImage* im1dummy=NULL;
IplImage* ctrimg=NULL;
IplImage* trackbars=NULL;
IplImage* graphs=NULL;
CvRect cam1rect;
CvRect cam2rect;
CvRect cam3rect;
CvRect cam4rect;
CvRect drag1rect;
CvRect drag2rect;

```

```

CvRect border1;
CvRect border2;
int tripoints1=0;
int tripoints2=0;
int tripoints3=0;
int xdrag=0;
int ydrag=0;
int updiff=0;
int righdiff=0;
int align=0;
int xo1=0;
int yo1=0;
int xo2=0;
int yo2=0;
int xo3=0;
int yo3=0;
float zoomfact1=0;
float zoomfact2=0;
float zoomfact3=0;
double
xnot=0,ynot=0,znot=0,rnot=0,M11=0,M12=0,M13=0,M14=0,M15=0,a11=0,a22=0,a33=0,a44=0,m00,m01,m02,m03,
m10,m11,m12,m13,m20,m21,m22,m23,m30,m31,m32,m33;
double x4det(double m00,double m01,double m02,double m03,double m10,double m11,double m12,double
m13,double m20,double m21,double m22,double m23,double m30,double m31,double m32,double m33)
{
double det=0;

det = m03 * m12 * m21 * m30 - m02 * m13 * m21 * m30- m03 * m11 * m22 * m30+m01 * m13 * m22 * m30+ m02
* m11 * m23 * m30-m01 * m12 * m23 * m30- m03 * m12 * m20 * m31+m02 * m13 * m20 * m31+ m03 * m10 *
m22 * m31-m00 * m13 * m22 * m31- m02 * m10 * m23 * m31+m00 * m12 * m23 * m31+ m03 * m11 * m20 * m32-
m01 * m13 * m20 * m32- m03 * m10 * m21 * m32+m00 * m13 * m21 * m32+ m01 * m10 * m23 * m32-m00 * m11
* m23 * m32- m02 * m11 * m20 * m33+m01 * m12 * m20 * m33+ m02 * m10 * m21 * m33-m00 * m12 * m21 *
m33- m01 * m10 * m22 * m33+m00 * m11 * m22 * m33;
return(det);
}

float zoomfact1x=0;
float zoomfact2x=0;
float zoomfact3x=0;
float zoomfact1y=0;
float zoomfact2y=0;
float zoomfact3y=0;

float rollmax=0;
float areamax=0;
float per1,per2,per3;
//For ellipse _____
const double pi=3.1415926535;
int i=0;
int count=0;
CvPoint center;
CvSize size;
CvBox2D32f* box;
CvPoint* PointArray;
CvPoint2D32f* PointArray2D32f;
CvBox2D ellipse1;
CvSeq* c;

// _____
int n_smooth=5;
int n_threshold=54;
int n_erode=4;

```

```

int n_canny=4;
void on_trackbar(int)
{
if(n_smooth%2==0) n_smooth++;
}

CvRect maxrect=cvRect(0,0,0,0);
struct raw
{
float le;
float we;
CvRect bdbbox;
CvBox2D ebox;
CvPoint headpt;
CvPoint tailpt;
CvPoint rollpt;
float A;
float roll;
float angle;

}rawCam1[700],rawCam2[700],rawCam3[700];
float thresh=0.1;
float rollframe[700];
float rollaccum=0;
//_____
// | | |
// 1 | 3 |
//_____|_____
// | | |
// 2 | 4 |
//_____|_____

CvPoint T1;

//For solve pnp_____

Mat op;

vector<Point3f> modelPoints;

float m_L=1;
float m_Ldash;
float m_Lddash;
float m_Ltdash;
float m_Lddashtil;
float m_Lnot;
float m_angle=30*3.141532/180;
float m_BL;

double rot[9] = {0};
Vec3d eav;
Mat backPxIs;
GLuint textureID;

vector<double> rv(3), tv(3);
Mat rvec(rv),tvec(tv);
Mat camMatrix;

//_____

```

```

float roll,pitch,yaw;
CvPoint ecentroid;
float framelength=0;
void solve_euler(void);

//

void iterate_for_roll(int,int,int,int***);

struct vector1
{
    double x;
    double y;
    double z;
}axis,body,body_rot,roll_axis,centroid_plane,temp1,temp2,temp3,main1,main2,main3,tail,test1,test2,test2a,test2b,vect
_thorax,vect_tail,test1x2;
struct coord
{
    double x;
    double y;
    double z;
}a,am,amm;

struct coordinate_system
{
    vector1 v1;
    vector1 v2;
    vector1 v3;
}global,body_cood,initial,post_yaw,post_pitch,post_roll,initial_dummy,initial_dummy2;

float angle_pitch=0;
float angle_yaw=0;
float angle_roll=0;
void model_body(void)
{
    m_Ldash=m_L/cos(m_angle);
    m_Lddash=m_L*tan(m_angle);
    m_Lddashtil=m_L*(1-tan(m_angle));
    m_Ltdash=m_Lddashtil/sin(m_angle);
    m_BL=m_Ltdash+m_Ldash;
    m_Lnot=m_Lddashtil*cos(m_angle);

    modelPoints.push_back(Point3f(0.0f,0.0f,0.0f)); //thorax front
    modelPoints.push_back(Point3f(m_L*sin(m_angle),m_L*cos(m_angle),0.2f)); //thorax top
    modelPoints.push_back(Point3f(m_L*cos(m_angle),-m_Lddash*cos(m_angle),0.0f)); //thorax bottom
    modelPoints.push_back(Point3f(m_Ldash,0.0f,0.0f)); //tail start bottom
    modelPoints.push_back(Point3f(m_BL,0.0f,0.1f)); //tailtip
op = Mat(modelPoints);

op = op / 35;

rvec = Mat(rv);
double _d[9] = {1,0,0,
                0,-1,0,
                0,0,-1};

Rodrigues(Mat(3,3,CV_64FC1,_d),rvec);
tv[0]=0;tv[1]=0;tv[2]=0;
tvec = Mat(tv);
double _cm[9] = { 20, 0, 160,
                  0, 20, 120,
                  0, 0, 1 };

```

```

camMatrix = Mat(3,3,CV_64FC1,_cm);
}

void loadNext(CvPoint A,CvPoint B,CvPoint C,CvPoint D,CvPoint E)
{
vector<Point2f > imagePoints;
imagePoints.push_back(Point2f((float)A.x,(float)A.y));
imagePoints.push_back(Point2f((float)B.x,(float)B.y));
imagePoints.push_back(Point2f((float)C.x,(float)C.y));
imagePoints.push_back(Point2f((float)D.x,(float)D.y));
imagePoints.push_back(Point2f((float)E.x,(float)E.y));

Mat ip(imagePoints);
double _dc[] = {0,0,0,0};
solvePnP(op,ip,camMatrix,Mat(1,4,CV_64FC1,_dc),rvec,tvec,true);
Mat rotM(3,3,CV_64FC1,rot);
Rodrigues(rvec,rotM);
double* _r = rotM.ptr<double>();

Mat tmp,tmp1,tmp2,tmp3,tmp4,tmp5;

double _pm[12] = {_r[0],_r[1],_r[2],0,
                 _r[3],_r[4],_r[5],0,
                 _r[6],_r[7],_r[8],0};

decomposeProjectionMatrix(Mat(3,4,CV_64FC1,_pm),tmp,tmp1,tmp2,tmp3,tmp4,tmp5,eav);
cout<<eav[0]<<","<<eav[1]<<","<<eav[2]<<endl;
eav[0]=0;
eav[1]=0;
eav[2]=0;

}

void calculate_pose(void)
{

}

float distance_ptsf1(CvPoint P1, CvPoint P2)
{
return (fabs(float(sqrt( pow(double(P1.x-P2.x),2) + pow(double(P1.y-P2.y),2))))));
}

float distance_ptsf1(CvPoint P1,CvPoint2D32f P2)
{
return (fabs(float(sqrt( pow(double(P1.x-P2.x),2) + pow(double(P1.y-P2.y),2))))));
}

float distance_linept(CvPoint P1,CvPoint P2,CvPoint2D32f P3)
{
float a,b,c;

a=float(P2.y)-float(P1.y);
b=float(P2.x)-float(P1.x);
c=float(P1.y)*b - float(P1.x)*a;

return( (a*float(P3.x) + b*float(P3.y) +c)/(sqrt(pow(a,2)+pow(b,2))));
}

float distance_linept(CvPoint P1,CvPoint P2,CvPoint P3)
{

```



```

float a,b,c;

a=float(P2.y)-float(P1.y);
b=float(P2.x)-float(P1.x);
c=float(P1.y)*b - float(P1.x)*a;

return( (a*float(P3.x) + b*float(P3.y) +c)/(sqrt(pow(a,2)+pow(b,2))));
}

float trianglearea(CvPoint P1,CvPoint P2,CvPoint2D32f P3)
{
float A=float(P3.x*P1.y) + float(P1.x*P2.y) + float(P2.x*P3.y) - float(P2.x*P1.y) - float(P3.x*P2.y) -
float(P1.x*P3.y);
A=fabs(A)/2.0;
return(A);
}

float trianglearea(CvPoint P1,CvPoint P2,CvPoint P3)
{
float A=float(P3.x*P1.y) + float(P1.x*P2.y) + float(P2.x*P3.y) - float(P2.x*P1.y) - float(P3.x*P2.y) -
float(P1.x*P3.y);
A=fabs(A)/2.0;
return(A);
}

float sameside(CvPoint P1,CvPoint P2,CvPoint P3,CvPoint P4)
{
float a,b,c;

a=float(P2.y)-float(P1.y);
b=float(P2.x)-float(P1.x);
c=float(P1.y)*b - float(P1.x)*a;
float s1,s2;

s1=a*float(P3.x) + b*float(P3.y) +c;
s2=a*float(P4.x) + b*float(P4.y) +c;

return(s1*s2);
}

void drag_line( int event, int x, int y, int flags, void* param)
{
if(event!=CV_EVENT_MOUSEMOVE && event==CV_EVENT_LBUTTONDOWN)
{
xdrag=x;
ydrag=y;
xo1=xdrag;
yo1=ydrag;
}

if(event==CV_EVENT_MOUSEMOVE && flags==CV_EVENT_FLAG_LBUTTON)
{
cvLine(im1clone,cvPoint(xdrag,ydrag),cvPoint(x,y),cvScalar(50,50,50),2,8,0);
updiff=y-ydrag;
rightdiff=x-xdrag;
}
}

```

```

        if(rightdiff>updiff) {rightdiff=0; xo2=x;yo2=y;}
        else if(rightdiff<updiff) {updiff=0; xo3=x;yo3=y;}
        cvShowImage("PiP",im1clone);
        cvCopy(im1dummy,im1clone);
    }

    if(event==CV_EVENT_RBUTTONDOWN)
    {
        cvDestroyWindow("PiP");
        align++;
    }
}

void my_mouse( int event, int x, int y, int flags, void* param)
{
    if(event==CV_EVENT_MOUSEMOVE) {

        if(event==CV_EVENT_LBUTTONDOWN) {}

        if(event==CV_EVENT_MBUTTONDOWN && align<=1)
        {cvShowImage("PiP",PiP);cvSetMouseCallback("PiP", drag_line, (void*)PiP); cvWaitKey(0); }

        if(event==CV_EVENT_RBUTTONDOWN && align<=1) {cvDestroyWindow("PiP");}
    }

    int N_start=1;
    int N_end=199;
    int N=N_start;
    int N_skip=2;

    char * string1;
    char * string2;
    char * string3;

void calc_roll(float a,float b, float c)
{
    if(a<0 && b<0 && c>0) { rollframe[N]=(fabs(a)+fabs(b)+fabs(c))/3.00;} //clockwise positive guess
    else if(a>0 && b>0 && c<0) { rollframe[N]=-1*(fabs(a)+fabs(b)+fabs(c))/3.00;} //anti-clockwise
    positive guess
    else { rollframe[N]=max(fabs(a),max(fabs(b),fabs(c)))*0.5;} //average positive guess
}
void processmap(void);

int main(int argc, char** argv)
{
    // {N_start=atoi(argv[1]);N_end=atoi(argv[2]); N_skip=atoi(argv[3]);}
    N=N_start;
    cout<<"Processing frames "<<N_start<<" to "<<N_end<<" (every "<<N_skip<<)"<<endl;

    CvFont font;
    double hScale=0.35;
    double vScale=0.35;
    int lineWidth=1;
    int mainloopcall=0;
    A1.open("dfly22-3-Pitch-Roll-Yaw.dat");

```

```

COM.open("22-3-centreofmass_path.dat");
fkpc.open("kpc-22-3.dat");
cross.open("instacross-22-3.dat");
A1<<"VARIABLES=\\"T\\",\\"auto_theta\\",\\"auto_psi\\",\\"auto_phi\\"\\n";
fkpc<<"Variables = \\"i\\" \\"j\\" \\"k\\"\\n";
cross<<"Variables = \\"Frame\\" \\"Magnitude\\",\\"r_not\\"\\n";
//Define the model
model_body();
//_____

cvInitFont(&font,CV_FONT_HERSHEY_TRIPLEX|CV_FONT_ITALIC, hScale,vScale,0,lineWidth);

imgCam1=cvLoadImage("C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.1_C001H001S0001\\Camera
No.1_C001H001S0001000001.tif",0);
imgCam1small=cvCreateImage(cvSize(imgCam1->width/2,imgCam1->height/2),imgCam1-
>depth,imgCam1->nChannels);
imgCam2small=cvCreateImage(cvSize(imgCam1->width/2,imgCam1->height/2),imgCam1-
>depth,imgCam1->nChannels);
imgCam3small=cvCreateImage(cvSize(imgCam1->width/2,imgCam1->height/2),imgCam1-
>depth,imgCam1->nChannels);
graphs=cvCreateImage(cvSize(imgCam1->width/2,imgCam1->height/2),IPL_DEPTH_32F,3);

trackbars=cvCreateImage(cvSize(imgCam1->width,50),imgCam1->depth,imgCam1->nChannels);
PiP=cvCreateImage(cvSize(imgCam1->width,imgCam1->height),imgCam1->depth,imgCam1->nChannels);
im1clone=cvCreateImage(cvSize(imgCam1->width,imgCam1->height),imgCam1->depth,imgCam1->nChannels);
im1dummy=cvCreateImage(cvSize(imgCam1->width,imgCam1->height),imgCam1->depth,imgCam1->nChannels);

ctrimg=cvLoadImage("C:\\SUMMER VIDEOS\\summer2010_22.3\\Camera No.1_C001H001S0001\\Camera
No.1_C001H001S0001000001.tif",1);
imgCam1smallc=cvCreateImage(cvSize(ctrimg->width/2,ctrimg->height/2),ctrimg->depth,ctrimg->nChannels);
imgCam2smallc=cvCreateImage(cvSize(ctrimg->width/2,ctrimg->height/2),ctrimg->depth,ctrimg->nChannels);
imgCam3smallc=cvCreateImage(cvSize(ctrimg->width/2,ctrimg->height/2),ctrimg->depth,ctrimg->nChannels);

g_storage=cvCreateMemStorage(0);
g_storage1=cvCreateMemStorage(0);
g_storage2=cvCreateMemStorage(0);
g_storage3=cvCreateMemStorage(0);
dp_storage=cvCreateMemStorage(0);
//
plot_new_pitch=cvPoint(35,imgCam1smallc->width/2);
plot_old_pitch=cvPoint(35,imgCam1smallc->width/2);
plot_new_roll=cvPoint(35,imgCam1smallc->width/2);
plot_old_roll=cvPoint(35,imgCam1smallc->width/2);
plot_new_yaw=cvPoint(35,imgCam1smallc->width/2);
plot_old_yaw=cvPoint(35,imgCam1smallc->width/2);
//
cvZero(graphs);
//cvNot(graphs,graphs);
cvLine(graphs,plot_new_pitch,cvPoint(plot_new_pitch.y*2,plot_new_pitch.y),cvScalarAll(100),1,8,0);
cvLine(graphs,cvPoint(plot_new_pitch.x,0),cvPoint(plot_new_pitch.x,plot_new_pitch.y*2),cvScalarAll(100),1,8,0);
cvPutText (graphs,"--Pitch--",cvPoint(400,40), &font, cvScalar(0,0,155));
cvPutText (graphs,"--Roll--",cvPoint(400,80), &font, cvScalar(155,0,0));
cvPutText (graphs,"--Yaw--",cvPoint(400,120), &font, cvScalar(0,155,0));
//
for(int i=-180;i<=180;i=i+20)
{
itoa(i,num,10);
cvPutText (graphs,num,cvPoint(0,int(35.00+float((180-i)*float(graphs->width-70)/float(360))), &font,
cvScalarAll(55));
}

```

```

        while(N<=N_end)//earlier Nmax
        {
//cout<<"                                \r";
        cvSetZero(ctrimg);
        cvSetZero(imgCam1small);
        cvSetZero(imgCam2small);
        cvSetZero(imgCam3small);

        cam1rect=cvRect(rightdiff,updiff,imgCam1small->width,imgCam1small->height);
        cam2rect=cvRect(0,imgCam1small->height,imgCam2small->width,imgCam2small->height);
        cam3rect=cvRect(imgCam1small->width,0,imgCam3small->width,imgCam3small->height);
        cam4rect=cvRect(imgCam1small->width,imgCam1small->height,imgCam3small->width,imgCam3small-
>height);
        if(align>=2) cout<<N<<"/"<<N_end<<" || ";
        itoa(N,num,10);

        if(N<10)                                {                                strcat(temp10,num);
strcat(temp10,".tif");strcat(basecam1,temp10);strcat(basecam2,temp10);strcat(basecam3,temp10);strcpy(temp10,temp1
0bi);}
        else if(N>=10                                &&                                N<100)                                {strcat(temp100,num);
strcat(temp100,".tif");strcat(basecam1,temp100);strcat(basecam2,temp100);strcat(basecam3,temp100);strcpy(temp100,
temp100bi);}
        else if(N>=100 && N<1000) {strcat(num,".tif");
        strcat(basecam1,num);
        strcat(basecam2,num);
        strcat(basecam3,num);}
        if(align>=2){N=N+N_skip;}
        imgCam1=cvLoadImage(basecam1,0);
        imgCam2=cvLoadImage(basecam2,0);
        imgCam3=cvLoadImage(basecam3,0);

        if(!imgCam1 && N!=N_end)
        {
                cout<<"Could not open \""<<basecam1<< "\"!.Exiting...";getchar();exit(0);
        }

        if(!imgCam2 && N!=N_end)
        {
                cout<<"Could not open \""<<basecam2<< "\"!.Exiting...";getchar();exit(0);
        }

        if(!imgCam3 && N!=N_end)
        {
                cout<<"Could not open \""<<basecam3<< "\"!.Exiting...";getchar();exit(0);
        }

        cvFlip(imgCam2,imgCam2,-1);//if required
        //cvFlip(imgCam2,imgCam2,1);//if required
        cvResize(imgCam1,imgCam1small);
        cvResize(imgCam2,imgCam2small);
        cvResize(imgCam3,imgCam3small);

        strcpy(basecam1,base1cam1);
        strcpy(basecam2,base1cam2);
        strcpy(basecam3,base1cam3);

        cvSetImageROI(PiP,cam1rect);
        cvResize(imgCam1small,PiP,1);
        cvResetImageROI(PiP);

        cvSetImageROI(PiP,cam2rect);

```

```

        cvResize(imgCam2small,PiP,1);
        cvResetImageROI(PiP);

        cvSetImageROI(PiP,cam3rect);
        cvResize(imgCam3small,PiP,1);
        cvResetImageROI(PiP);
        if(align<2){rawCam1[N].A=0;rawCam1[N].roll=0;                rawCam2[N].A=0;rawCam2[N].roll=0;
rawCam3[N].A=0;rawCam3[N].roll=0;}
        if(align>=2){
            cvSmooth(PiP,PiP,CV_GAUSSIAN,n_smooth,n_smooth);//17,17 for test1,
            cvWaitKey(20);
            cvThreshold( PiP,PiP,n_threshold, 100, CV_THRESH_TRUNC );
            cvEqualizeHist( PiP,PiP );
            cvErode(PiP,PiP,0,n_erode);
            cvCanny( PiP, PiP,n_canny, 100,3);
            if(updiff<0) drag1rect=cvRect(0,imgCam1small->height+updiff-4,imgCam2small->width+4,-updiff+8);
            if(rightdiff<0) drag2rect=cvRect(imgCam1small->width+rightdiff-8,0,-rightdiff+12,imgCam2small->height+4);
            if(updiff>0)drag1rect=cvRect(0,0,imgCam2small->width+4,updiff+8);
            if(rightdiff>0)drag2rect=cvRect(0,0,rightdiff+8,imgCam2small->height+4);
            border1=cvRect(0,imgCam1small->height-8,imgCam1->width,16);
            border2=cvRect(imgCam1small->width-8,0,16,imgCam1->height);
            cvSetImageROI(PiP,cam4rect);
            cvSetZero(PiP);
            cvResetImageROI(PiP);
            //
            cvSetImageROI(PiP,drag1rect);
            cvSetZero(PiP);
            cvResetImageROI(PiP);
            //
            cvSetImageROI(PiP,drag2rect);
            cvSetZero(PiP);
            cvResetImageROI(PiP);

            cvSetImageROI(PiP,border1);
            cvSetZero(PiP);
            cvResetImageROI(PiP);

            cvSetImageROI(PiP,border2);
            cvSetZero(PiP);
            cvResetImageROI(PiP);

        }
        if(align>=2) {xo1=xo3;yol=yo2;

//Split into images again for processing

        cvSetImageROI(PiP,cam1rect);
        cvResize(PiP,imgCam1small,1);
        cvResetImageROI(PiP);

        cvSetImageROI(PiP,cam2rect);
        cvResize(PiP,imgCam2small,1);
        cvResetImageROI(PiP);

        cvSetImageROI(PiP,cam3rect);
        cvResize(PiP,imgCam3small,1);
        cvResetImageROI(PiP);

//-----

        cvFindContours(PiP,g_storage,&contours,sizeof(CvContour),CV_RETR_EXTERNAL,CV_CHAIN_APPR
OX_SIMPLE);

```

```

        cvFindContours(imgCam1small,g_storage1,&contours1,sizeof(CvContour),CV_RETR_EXTERNAL,CV_C
HAIN_APPROX_NONE);
        cvFindContours(imgCam2small,g_storage2,&contours2,sizeof(CvContour),CV_RETR_EXTERNAL,CV_C
HAIN_APPROX_NONE);
        cvFindContours(imgCam3small,g_storage3,&contours3,sizeof(CvContour),CV_RETR_EXTERNAL,CV_C
HAIN_APPROX_NONE);
        cvSetZero(imgCam1small);
        cvSetZero(imgCam2small);
        cvSetZero(imgCam3small);

//if(contours) cvDrawContours(ctrimg,contours,colour4,colour4,2);
        if(contours1) cvDrawContours(imgCam1smallc,contours1,colour4,colour4,2);
        if(contours2) cvDrawContours(imgCam2smallc,contours2,colour4,colour4,2);
        if(contours3) cvDrawContours(imgCam3smallc,contours3,colour4,colour4,2);
        if(contours) cvDrawContours(PiP,contours,cvScalarAll(255),cvScalarAll(255),2);
quad1=contours1;
quad2=contours2;
quad3=contours3;
maxrect=cvRect(0,0,0,0);
for(c=contours1;c!=0;c=c->h_next)
{
    if(        cvBoundingRect(c,0).width*cvBoundingRect(c,0).height > maxrect.height*maxrect.width)
        maxrect=cvBoundingRect(c,0);
}
rawCam1[N].bdblbox=maxrect;

maxrect=cvRect(0,0,0,0);
for(c=contours2;c!=0;c=c->h_next)
{
    if(        cvBoundingRect(c,0).width*cvBoundingRect(c,0).height > maxrect.height*maxrect.width)
        maxrect=cvBoundingRect(c,0);
}
rawCam2[N].bdblbox=maxrect;

maxrect=cvRect(0,0,0,0);
for(c=contours3;c!=0;c=c->h_next)
{
    if(        cvBoundingRect(c,0).width*cvBoundingRect(c,0).height > maxrect.height*maxrect.width)
        maxrect=cvBoundingRect(c,0);
}
rawCam3[N].bdblbox=maxrect;

//-----
//imgcam1small contours
for(c=contours1;c!=0;c=c->h_next)
{

    if(c->total >=6 && fabs(cvContourArea(c))>500.0)
    {
        bdblrect = cvBoundingRect(c,0);
        if(rawCam1[N].bdblbox.height==bdblrect.height && rawCam1[N].bdblbox.width==bdblrect.width)
        {
            ellipse1=cvFitEllipse2(c);
            ellipse1.angle=-ellipse1.angle;
            cvEllipseBox(imgCam1smallc,ellipse1,colour3);

            cvRectangle(imgCam1smallc,cvPoint(bdblrect.x,bdblrect.y),cvPoint(bdblrect.x+bdblrect.width,bdblrect.y+bdblrect.heig
ht),colour4, 1);
            rawCam1[N].ebox=ellipse1;
        }
    }
}

```

```

}
//imgcam2small contours
for(c=contours2;c!=0;c=c->h_next)
{
    if(c->total >=6 && fabs(cvContourArea(c))>500.0)
    {
        bdirect = cvBoundingRect(c,0);
        if(rawCam2[N].bdbbox.height==bdirect.height && rawCam2[N].bdbbox.width==bdirect.width)
        {
            ellipse1=cvFitEllipse2(c);
            ellipse1.angle=-ellipse1.angle;
            cvEllipseBox(imgCam2smallc,ellipse1,colour3);

            cvRectangle(imgCam2smallc,cvPoint(bdirect.x,bdirect.y),cvPoint(bdirect.x+bdirect.width,bdirect.y+bdirect.height),colour4, 1);
            rawCam2[N].ebox=ellipse1;
        }
    }
}
//imgcam3small contours
for(c=contours3;c!=0;c=c->h_next)
{
    if(c->total >=6 && fabs(cvContourArea(c))>500.0)
    {
        bdirect = cvBoundingRect(c,0);
        if(rawCam3[N].bdbbox.height==bdirect.height && rawCam3[N].bdbbox.width==bdirect.width)
        {
            ellipse1=cvFitEllipse2(c);
            ellipse1.angle=-ellipse1.angle;
            cvEllipseBox(imgCam3smallc,ellipse1,colour3);

            cvRectangle(imgCam3smallc,cvPoint(bdirect.x,bdirect.y),cvPoint(bdirect.x+bdirect.width,bdirect.y+bdirect.height),colour4, 1);
            rawCam3[N].ebox=ellipse1;
        }
    }
}
//-----

contours1 = cvApproxPoly( contours1, sizeof(CvContour), g_storage1, CV_POLY_APPROX_DP,9, 1 );
if(contours1) cvDrawContours(imgCam1smallc,contours1,colour1,colour2,2);
tri1=contours1;
contours2 = cvApproxPoly( contours2, sizeof(CvContour), g_storage2, CV_POLY_APPROX_DP,9, 1 );
if(contours2) cvDrawContours(imgCam2smallc,contours2,colour1,colour2,2);
tri2=contours2;
contours3 = cvApproxPoly( contours3, sizeof(CvContour), g_storage3, CV_POLY_APPROX_DP,9, 1 );
if(contours3) cvDrawContours(imgCam3smallc,contours3,colour1,colour2,2);
tri3=contours3;

contours1 = cvApproxPoly( contours1, sizeof(CvContour), g_storage1, CV_POLY_APPROX_DP,37, 1 );
if(contours1) cvDrawContours(imgCam1smallc,contours1,colour2,colour1,2);
bi1=contours1;
contours2 = cvApproxPoly( contours2, sizeof(CvContour), g_storage2, CV_POLY_APPROX_DP,37, 1 );

```

```

    if(contours2) cvDrawContours(imgCam2smallc,contours2,colour2,colour1,2);
    bi2=contours2;
    contours3 = cvApproxPoly( contours3, sizeof(CvContour), g_storage3, CV_POLY_APPROX_DP,37, 1 );
    if(contours3) cvDrawContours(imgCam3smallc,contours3,colour2,colour1,2);
    bi3=contours3;

//-----

// Processing bi and tri contours for each camera...
//Cam1
//cout<<"\ncam1 :";
for(c=contours1;c!=0;c=c->h_next)
{

CvSeqReader reader1;
cvStartReadSeq( c, &reader1, 0 );
if(fabs(cvContourPerimeter(c))>50.0 && c->total==2)
{
//cout<<c->total<<" ";
    for ( int i = 0; i < 2; i++ )
    {
        CV_READ_SEQ_ELEM (T1, reader1 );
        if(i==0) dummyhead=T1;
        if(i==1) dummytail=T1;
        //cout<<"<<T1.x<<","<<T1.y<<");
    }
    //cout<<"Cam1[("<<dummyhead.x<<","<<dummyhead.y<<"),("<<dummytail.x<<","<<dummytail.y<<")]";
}

if(dummyhead.x>=rawCam1[N].bobox.x && dummyhead.x<=rawCam1[N].bobox.x + rawCam1[N].bobox.width &&
dummyhead.y>=rawCam1[N].bobox.y && dummyhead.y<=rawCam1[N].bobox.y + rawCam1[N].bobox.height &&
dummytail.x>=rawCam1[N].bobox.x && dummytail.x<=rawCam1[N].bobox.x + rawCam1[N].bobox.width &&
dummytail.y>=rawCam1[N].bobox.y && dummytail.y<=rawCam1[N].bobox.y + rawCam1[N].bobox.height)
{if(distance_ptsf1(dummyhead,rawCam1[N].ebox.center)>distance_ptsf1(dummytail,rawCam1[N].ebox.center))
{exchange=dummyhead;          dummyhead=dummytail;          dummytail=exchange;
rawCam1[N].le=distance_ptsf1(rawCam1[N].headpt,rawCam1[N].ebox.center);
rawCam1[N].we=fabs(trianglearea(rawCam1[N].headpt,rawCam1[N].tailpt,rawCam1[N].ebox.center)*2.00/distance_p
tsf1(rawCam1[N].headpt,rawCam1[N].tailpt));}
}
}
cvCircle(imgCam1smallc,dummyhead,5,cvScalarAll(255),2,8,0);
cvCircle(imgCam1smallc,dummytail,1,cvScalarAll(255),2,8,0);
rawCam1[N].headpt=dummyhead;rawCam1[N].tailpt=dummytail;
areamax=0;
for(c=tri1;c!=0;c=c->h_next)
{
    CvSeqReader reader1;

    cvStartReadSeq( c, &reader1, 0 );
    tripoints1=c->total;
    //cout<<tripsoints1<<"*";
    for ( int i = 0; i < c->total; i++ )
    {
        CV_READ_SEQ_ELEM ( dummyroll, reader1 );
        if(tripsoints1==5)
        {
            if(i==1) dummy1=dummyroll;
            if(i==2) dummy2=dummyroll;
            if(i==3) dummy3=dummyroll;

```



```

        if(i==4) dummy4=dummyroll;
        if(i==5) dummy5=dummyroll;
    }
    if(dummyroll.x>=rawCam1[N].bobox.x && dummyroll.x<=rawCam1[N].bobox.x + rawCam1[N].bobox.width &&
    dummyroll.y>=rawCam1[N].bobox.y && dummyroll.y<=rawCam1[N].bobox.y + rawCam1[N].bobox.height)
    { //cout<<"!";
        if(areamax<trianglearea(rawCam1[N].headpt,rawCam1[N].tailpt,dummyroll))
        {areamax=trianglearea(rawCam1[N].headpt,rawCam1[N].tailpt,dummyroll);rawCam1[N].rollpt=dummyroll;}
    }
    }
    //if(tripoints1==5)cout<<"\nCam1 gives :"; loadNext(dummy1,dummy2,dummy3,dummy4,dummy5);
    tripoints1=0;
    //cout<<"\n";

    //Cam2
    //cout<<"ncam3 :";
    for(c=contours2;c!=0;c=c->h_next)
    {

    CvSeqReader reader2;
    cvStartReadSeq( c, &reader2, 0 );
    if(fabs(cvContourPerimeter(c))>50.0 && c->total==2)
    { //cout<<c->total<<" ";
        for ( int i = 0; i < 2; i++ )
        {
            CV_READ_SEQ_ELEM (T1, reader2 );
            if(i==0) dummyhead=T1;
            if(i==1) dummytail=T1;
            //cout<<"("<<T1.x<<","<<T1.y<<")";
        }
        //cout<<"Cam2[("<<dummyhead.x<<","<<dummyhead.y<<"),("<<dummytail.x<<","<<dummytail.y<<")]);
    }
    if(dummyhead.x>=rawCam2[N].bobox.x && dummyhead.x<=rawCam2[N].bobox.x + rawCam2[N].bobox.width &&
    dummyhead.y>=rawCam2[N].bobox.y && dummyhead.y<=rawCam2[N].bobox.y + rawCam2[N].bobox.height &&
    dummytail.x>=rawCam2[N].bobox.x && dummytail.x<=rawCam2[N].bobox.x + rawCam2[N].bobox.width &&
    dummytail.y>=rawCam2[N].bobox.y && dummytail.y<=rawCam2[N].bobox.y + rawCam2[N].bobox.height)
    { if(distance_ptsf1(dummyhead,rawCam2[N].ebox.center)>distance_ptsf1(dummytail,rawCam2[N].ebox.center))
        {exchange=dummyhead; dummyhead=dummytail; dummytail=exchange;
        rawCam2[N].le=distance_ptsf1(rawCam2[N].headpt,rawCam2[N].ebox.center);
        rawCam2[N].we=fabs(distance_linept(rawCam2[N].headpt,rawCam2[N].tailpt,rawCam2[N].ebox.center)); }
    }
    cvCircle(imgCam2smallc,dummyhead,5,cvScalarAll(255),2,8,0);
    cvCircle(imgCam2smallc,dummytail,1,cvScalarAll(255),2,8,0);
    rawCam2[N].headpt=dummyhead;rawCam2[N].tailpt=dummytail;
    areamax=0;
    for(c=tri2;c!=0;c=c->h_next)
    {
        CvSeqReader reader2;

        cvStartReadSeq( c, &reader2, 0 );
        tripoints2=c->total;
        //cout<<tripoints2<<"*";
        for ( int i = 0; i < c->total; i++ )
        {
            CV_READ_SEQ_ELEM ( dummyroll, reader2 );
            if(tripoints2==5)
            {

```

```

        if(i==1) dummy1=dummyroll;
        if(i==2) dummy2=dummyroll;
        if(i==3) dummy3=dummyroll;
        if(i==4) dummy4=dummyroll;
        if(i==5) dummy5=dummyroll;
    }
    if(dummyroll.x>=rawCam2[N].bobox.x && dummyroll.x<=rawCam2[N].bobox.x + rawCam2[N].bobox.width &&
    dummyroll.y>=rawCam2[N].bobox.y && dummyroll.y<=rawCam2[N].bobox.y + rawCam2[N].bobox.height)
    {
        //cout<<"!";
        if(areamax<trianglearea(rawCam2[N].headpt,rawCam2[N].tailpt,dummyroll))
        {areamax=trianglearea(rawCam2[N].headpt,rawCam2[N].tailpt,dummyroll);rawCam2[N].rollpt=dummyroll;}
    }
}
}
//if(tripoints2==5)cout<<"\nCam2 gives :"; loadNext(dummy1,dummy2,dummy3,dummy4,dummy5);
tripoints2=0;

//cout<<"\n";
rawCam2[N].A=trianglearea(rawCam2[N].headpt,rawCam2[N].tailpt,rawCam2[N].rollpt);
if(rawCam2[N-1].A!=0)
{
    if(rawCam2[N].A<=rawCam2[N-1].A) rawCam2[N].roll=acos(float(rawCam2[N].A)/float(rawCam2[N-1].A));
    else if(rawCam2[N].A>rawCam2[N-1].A) rawCam2[N].roll=-1*acos(float(rawCam2[N-1].A)/float(rawCam2[N].A));
}
//Cam3
//cout<<"\ncam1 :";
for(c=contours3;c!=0;c=c->h_next)
{

CvSeqReader reader3;
cvStartReadSeq( c, &reader3, 0 );
if(fabs(cvContourPerimeter(c))>50.0 && c->total==2)
{
//cout<<c->total<<"\n";
    for ( int i = 0; i < 2; i++ )
    {
        CV_READ_SEQ_ELEM (T1, reader3 );
        if(i==0) dummyhead=T1;
        if(i==1) dummytail=T1;
        //cout<<("<<T1.x<<","<<T1.y<<");
    }
//cout<<"Cam3[("<<dummyhead.x<<","<<dummyhead.y<<"),("<<dummytail.x<<","<<dummytail.y<<")];
}
if(dummyhead.x>=rawCam3[N].bobox.x && dummyhead.x<=rawCam3[N].bobox.x + rawCam3[N].bobox.width &&
dummyhead.y>=rawCam3[N].bobox.y && dummyhead.y<=rawCam3[N].bobox.y + rawCam3[N].bobox.height &&
dummytail.x>=rawCam3[N].bobox.x && dummytail.x<=rawCam3[N].bobox.x + rawCam3[N].bobox.width &&
dummytail.y>=rawCam3[N].bobox.y && dummytail.y<=rawCam3[N].bobox.y + rawCam3[N].bobox.height)
{if(distance_ptsf1(dummyhead,rawCam3[N].ebox.center)>distance_ptsf1(dummytail,rawCam3[N].ebox.center))
{exchange=dummyhead; dummyhead=dummytail; dummytail=exchange;
rawCam3[N].le=distance_ptsf1(rawCam3[N].headpt,rawCam3[N].ebox.center);
rawCam3[N].we=fabs(distance_linept(rawCam3[N].headpt,rawCam1[N].tailpt,rawCam3[N].ebox.center)); }
}
}
cvCircle(imgCam3smallc,dummyhead,5,cvScalarAll(255),2,8,0);
cvCircle(imgCam3smallc,dummytail,1,cvScalarAll(255),2,8,0);
rawCam3[N].headpt=dummyhead;rawCam3[N].tailpt=dummytail;

```

```

areamax=0;
for(c=tri3;c!=0;c=c->h_next)
{
    CvSeqReader reader3;

    cvStartReadSeq( c, &reader3, 0 );
    tripoints3=c->total;
    //cout<<trips3<<"*";
    for ( int i = 0; i < c->total; i++ )
    {
        CV_READ_SEQ_ELEM ( dummyroll, reader3 );
        if(trips3==5)
        {
            if(i==1) dummy1=dummyroll;
            if(i==2) dummy2=dummyroll;
            if(i==3) dummy3=dummyroll;
            if(i==4) dummy4=dummyroll;
            if(i==5) dummy5=dummyroll;

        }

        if(dummyroll.x>=rawCam3[N].bdsbox.x && dummyroll.x<=rawCam3[N].bdsbox.x + rawCam3[N].bdsbox.width &&
        dummyroll.y>=rawCam3[N].bdsbox.y && dummyroll.y<=rawCam3[N].bdsbox.y + rawCam3[N].bdsbox.height)
        { //cout<<"!";
            if(areamax<trianglearea(rawCam3[N].headpt,rawCam3[N].tailpt,dummyroll))
            {areamax=trianglearea(rawCam3[N].headpt,rawCam3[N].tailpt,dummyroll);rawCam3[N].rollpt=dummyroll;}
        }
    }
}
//if(trips3==5)cout<<"\nCam2 gives :"; loadNext(dummy1,dummy2,dummy3,dummy4,dummy5);
trips3=0;
//cout<<"\n";
rawCam3[N].A=trianglearea(rawCam3[N].headpt,rawCam3[N].tailpt,rawCam3[N].rollpt);
if(rawCam3[N-1].A!=0)
{
    if(rawCam3[N].A<=rawCam3[N-1].A) rawCam3[N].roll=acos(float(rawCam3[N].A)/float(rawCam3[N-1].A));
    else if(rawCam3[N].A>rawCam3[N-1].A) rawCam3[N].roll=-1*acos(float(rawCam3[N-1].A)/float(rawCam3[N].A));
}
//-----
//Angle calculation

//cam1

if(rawCam1[N].headpt.x > rawCam1[N].tailpt.x) // 3 or 4 in camera sense
{
    if(rawCam1[N].headpt.y < rawCam1[N].tailpt.y) rawCam1[N].angle=atan(float(-rawCam1[N].headpt.y+rawCam1[N].tailpt.y)/float(rawCam1[N].headpt.x-rawCam1[N].tailpt.x));
    else if(rawCam1[N].headpt.y > rawCam1[N].tailpt.y) rawCam1[N].angle=1.5*pi+atan(float(rawCam1[N].headpt.y-rawCam1[N].tailpt.y)/float(rawCam1[N].headpt.x-rawCam1[N].tailpt.x));
}
else if(rawCam1[N].headpt.x < rawCam1[N].tailpt.x) // 1 or 2 in camera sense
{
    if(rawCam1[N].headpt.y < rawCam1[N].tailpt.y) rawCam1[N].angle=pi+atan(float(-rawCam1[N].headpt.y+rawCam1[N].tailpt.y)/float(-rawCam1[N].headpt.x+rawCam1[N].tailpt.x));
    else if(rawCam1[N].headpt.y > rawCam1[N].tailpt.y) rawCam1[N].angle=0.5*pi+atan(float(rawCam1[N].headpt.y-rawCam1[N].tailpt.y)/float(-rawCam1[N].headpt.x-rawCam1[N].tailpt.x));
}
}

```

```

rawCam1[N].A=trianglearea(rawCam1[N].headpt,rawCam1[N].tailpt,rawCam1[N].rollpt);

if(align>=2 && sameside(rawCam1[N].headpt,rawCam1[N].tailpt,rawCam1[N].rollpt,rawCam1[N-1].rollpt)>0)
{
    if(rawCam1[N-1].A!=0)
    {

        if(rawCam1[N].A<=rawCam1[N-1].A) rawCam1[N].roll=acos(float(rawCam1[N].A)/float(rawCam1[N-1].A));
        else if(rawCam1[N].A>rawCam1[N-1].A) rawCam1[N].roll=-1*acos(float(rawCam1[N-1].A)/float(rawCam1[N].A));

    }

}

per1=fabs((rawCam1[N].A-rawCam1[N-1].A)/rawCam1[N].A);
if(per1 < thresh || per1>thresh+0.04) rawCam1[N].roll=0;
zoomfact1=distance_ptsf1(cvPoint(xo1,yo1),rawCam1[N].ebox.center)/imgCam1small->width;
zoomfact1x=fabs(float(xo1-rawCam1[N].headpt.x))/imgCam1small->width;
zoomfact1y=fabs(float(yo1-rawCam1[N].headpt.y))/imgCam1small->width;

//cam2

if(rawCam2[N].headpt.x > rawCam2[N].tailpt.x) // 3 or 4 in camera sense
{
    if(rawCam2[N].headpt.y < rawCam2[N].tailpt.y) rawCam2[N].angle=atan(float(-rawCam2[N].headpt.y+rawCam2[N].tailpt.y)/float(rawCam2[N].headpt.x-rawCam2[N].tailpt.x));
    else if(rawCam2[N].headpt.y > rawCam2[N].tailpt.y) rawCam2[N].angle=1.5*pi+atan(float(rawCam2[N].headpt.y-rawCam2[N].tailpt.y)/float(rawCam2[N].headpt.x-rawCam2[N].tailpt.x));
}
else if(rawCam2[N].headpt.x < rawCam2[N].tailpt.x) // 1 or 2 in camera sense
{
    if(rawCam2[N].headpt.y < rawCam2[N].tailpt.y) rawCam2[N].angle=pi+atan(float(-rawCam2[N].headpt.y+rawCam2[N].tailpt.y)/float(-rawCam2[N].headpt.x+rawCam2[N].tailpt.x));
    else if(rawCam2[N].headpt.y > rawCam2[N].tailpt.y) rawCam2[N].angle=0.5*pi+atan(float(rawCam2[N].headpt.y-rawCam2[N].tailpt.y)/float(-rawCam2[N].headpt.x-rawCam2[N].tailpt.x));
}

if(align>=2 && sameside(rawCam2[N].headpt,rawCam2[N].tailpt,rawCam2[N].rollpt,rawCam2[N-1].rollpt)>0)
{
    if(rawCam2[N-1].A!=0)
    {

        if(rawCam2[N].A<=rawCam2[N-1].A) rawCam2[N].roll=acos(float(rawCam2[N].A)/float(rawCam2[N-1].A));
        else if(rawCam2[N].A>rawCam2[N-1].A) rawCam2[N].roll=-1*acos(float(rawCam2[N-1].A)/float(rawCam2[N].A));

    }

}

per2=fabs((rawCam2[N].A-rawCam2[N-1].A)/rawCam2[N].A);
if(per2 < thresh || per2>thresh+0.04) rawCam2[N].roll=0;
zoomfact2=distance_ptsf1(cvPoint(xo2,yo2),rawCam2[N].ebox.center)/imgCam1small->width;
zoomfact2x=fabs(float(xo2-rawCam2[N].headpt.x))/imgCam1small->width;
zoomfact2y=fabs(float(yo2-rawCam2[N].headpt.y))/imgCam1small->width;

//cam3
//note the change

```

```

if(rawCam3[N].headpt.x < rawCam3[N].tailpt.x) // 3 or 4 in camera sense
{
    if(rawCam3[N].headpt.y < rawCam3[N].tailpt.y) rawCam3[N].angle=atan(float(-
rawCam3[N].headpt.y+rawCam3[N].tailpt.y)/float(rawCam3[N].headpt.x-rawCam3[N].tailpt.x));
    else if(rawCam3[N].headpt.y > rawCam3[N].tailpt.y) rawCam3[N].angle=1.5*pi+atan(float(rawCam3[N].headpt.y-rawCam3[N].tailpt.y)/float(rawCam3[N].headpt.x-
rawCam3[N].tailpt.x));
}
else if(rawCam3[N].headpt.x > rawCam3[N].tailpt.x) // 1 or 2 in camera sense
{
    if(rawCam3[N].headpt.y < rawCam3[N].tailpt.y) rawCam3[N].angle=pi+atan(float(-
rawCam3[N].headpt.y+rawCam3[N].tailpt.y)/float(-rawCam3[N].headpt.x+rawCam3[N].tailpt.x));
    else if(rawCam3[N].headpt.y > rawCam3[N].tailpt.y) rawCam3[N].angle=0.5*pi+atan(float(rawCam3[N].headpt.y-rawCam3[N].tailpt.y)/float(-rawCam3[N].headpt.x-
rawCam3[N].tailpt.x));
}
if(align>=2 && sameside(rawCam3[N].headpt,rawCam3[N].tailpt,rawCam3[N].rollpt,rawCam3[N-1].rollpt)>0)
{
    if(rawCam3[N-1].A!=0)
    {
        if(rawCam3[N].A<=rawCam3[N-1].A) rawCam3[N].roll=acos(float(rawCam3[N].A)/float(rawCam3[N-
1].A));
        else if(rawCam3[N].A>rawCam3[N-1].A) rawCam3[N].roll=-1*acos(float(rawCam3[N-
1].A)/float(rawCam3[N].A));
    }
}
per3=fabs((rawCam3[N].A-rawCam3[N-1].A)/rawCam3[N].A);
if( per3 < thresh || per3>thresh+0.04) rawCam3[N].roll=0;
zoomfact3=distance_ptsf1(cvPoint(xo3,yo3),rawCam3[N].ebox.center)/imgCam1small->width;
zoomfact3x=fabs(float(xo3-rawCam3[N].headpt.x)/imgCam1small->width;
zoomfact3y=fabs(float(yo3-rawCam3[N].headpt.y)/imgCam1small->width;
cvCircle(imgCam1smallc,rawCam1[N].rollpt,3,cvScalarAll(155),2,8,0);
cvCircle(imgCam2smallc,rawCam2[N].rollpt,3,cvScalarAll(155),2,8,0);
cvCircle(imgCam3smallc,rawCam3[N].rollpt,3,cvScalarAll(155),2,8,0);
cvRectangle(imgCam1smallc,cvPoint(rawCam1[N].bdbox.x,rawCam1[N].bdbox.y),cvPoint(rawCam1[N].bdbox.x+ra
wCam1[N].bdbox.width,rawCam1[N].bdbox.y+rawCam1[N].bdbox.height),CV_RGB(10,100,55), 1);
cvRectangle(imgCam2smallc,cvPoint(rawCam2[N].bdbox.x,rawCam2[N].bdbox.y),cvPoint(rawCam2[N].bdbox.x+ra
wCam2[N].bdbox.width,rawCam2[N].bdbox.y+rawCam2[N].bdbox.height),CV_RGB(10,100,55), 1);
cvRectangle(imgCam3smallc,cvPoint(rawCam3[N].bdbox.x,rawCam3[N].bdbox.y),cvPoint(rawCam3[N].bdbox.x+ra
wCam3[N].bdbox.width,rawCam3[N].bdbox.y+rawCam3[N].bdbox.height),CV_RGB(10,100,55), 1);

cvSetImageROI(ctrimg,cam1rect);
cvResize(imgCam1smallc,ctrimg,1);
cvResetImageROI(ctrimg);

cvSetImageROI(ctrimg,cam2rect);
cvResize(imgCam2smallc,ctrimg,1);
cvResetImageROI(ctrimg);

cvSetImageROI(ctrimg,cam3rect);
cvResize(imgCam3smallc,ctrimg,1);
cvResetImageROI(ctrimg);

cvWaitKey(10);
cvShowImage("Contours",ctrimg);

//if(align>=2 && mainloopcall == 0) { loadNext(rawCam1[N].headpt,rawCam1[N].rollpt,rawCam1[N].tailpt);
start_opengl_with_stereo(argc,argv); mainloopcall++;}

```

```

}
cvCircle(PiP,cvPoint(xo1,yo1),3,cvScalarAll(255),2,8,0);
cvCircle(PiP,cvPoint(xo2,yo2),3,cvScalarAll(255),2,8,0);
cvCircle(PiP,cvPoint(xo3,yo3),3,cvScalarAll(255),2,8,0);

if(N%5==0)
{
    if(xo2!=0)cvLine(PiP,cvPoint(xo1,yo1),cvPoint(xo2,yo2),cvScalarAll(180),1,8,0);
    if(xo3!=0)cvLine(PiP,cvPoint(xo1,yo1),cvPoint(xo3,yo3),cvScalarAll(180),1,8,0);
}
//cvShowImage("1",imgCam1);
//cvShowImage("2",imgCam2);
//cvShowImage("3",imgCam3);

if(align==0) cvPutText (PiP,"Align Left to Right...",cvPoint(700+40,550+40), &font, cvScalar(255,255,0));
if(align==1) cvPutText (PiP,"Align Top to Bottom...",cvPoint(550+40,750+40), &font, cvScalar(255,255,0));

if(align>=2){
cvPutText (PiP,"_____ ",cvPoint(700+40,750+40), &font, cvScalar(255,255,0));
cvPutText (PiP," | " ,cvPoint(699+40,770+40), &font, cvScalar(255,255,0));
cvPutText (PiP," 1 | 3 " ,cvPoint(696+40,790+40), &font, cvScalar(255,255,0));
cvPutText (PiP,"_____ ",cvPoint(700+40,810+40), &font, cvScalar(255,255,0));
cvPutText (PiP," | " ,cvPoint(700+40,830+40), &font, cvScalar(255,255,0));
cvPutText (PiP," 2 | 4 " ,cvPoint(697+40,850+40), &font, cvScalar(255,255,0));
cvPutText (PiP,"_____ ",cvPoint(700+40,870+40), &font, cvScalar(255,255,0));

//loadNext(rawCam1[N].headpt,rawCam1[N].rollpt,rawCam1[N].tailpt);
    processmap();
    //*****HELLLOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO*****
    //*****
}

// solve_euler();
cvShowImage("PiP",PiP);
if(align>=2)cvShowImage("trackbars",trackbars);
cvCreateTrackbar( "Threshold","trackbars", &n_threshold, 150, on_trackbar );
cvCreateTrackbar( "Erode","trackbars", &n_erode, 10, on_trackbar );
cvCreateTrackbar( "Smooth","trackbars",&n_smooth,20, on_trackbar );
cvCreateTrackbar( "Canny","trackbars",&n_canny,50, on_trackbar );

on_trackbar(0);
//cvCreateTrackbar( "Static_Ignore","PiP",&static_switch, 1, on_trackbar );
cvResize(PiP,im1clone);
cvResize(PiP,im1dummy);

cvSetMouseCallback( "PiP", my_mouse, (void*)PiP);
cvWaitKey(20);
}
A1.close();
return 0;
}

float GlobalTriArea=0;

void solve_euler(void)
{

```

```

    int choice=0; // Chooses which is the defective camera
    if(N<3) GlobalBodyLength = sqrt(float( pow(distance_ptsf1(rawCam1[N].headpt,rawCam1[N].tailpt),2) +
pow(distance_ptsf1(rawCam2[N].headpt,rawCam2[N].tailpt),2) +
pow(distance_ptsf1(rawCam3[N].headpt,rawCam3[N].tailpt),2)));
    if(N<3) GlobalTriArea = sqrt(float( pow( rawCam1[N].A,2) + pow( rawCam2[N].A,2) + pow(
rawCam3[N].A,2)));

float CurrentBodyLength=sqrt(float( pow(distance_ptsf1(rawCam1[N].headpt,rawCam1[N].tailpt),2) +
pow(distance_ptsf1(rawCam2[N].headpt,rawCam2[N].tailpt),2) +
pow(distance_ptsf1(rawCam3[N].headpt,rawCam3[N].tailpt),2)));
float CurrentTriArea= sqrt(float( pow( rawCam1[N].A,2) + pow( rawCam2[N].A,2) + pow( rawCam3[N].A,2) ));

    if(quad1->total > 3 && quad2->total>3) choice=3;
    else if(quad1->total > 3 && quad3->total>3) choice=2;
    else if(quad2->total > 3 && quad2->total>3) choice=1;
    else cout<<"\nCannot calculate for this frame!\n";
    //choice=2;
float actualscaling=0;
float apparentscaling=CurrentBodyLength/GlobalBodyLength;
if( apparentscaling>1 ) apparentscaling=1/apparentscaling;
float distance=0;
rawCam1[N].we=fabs(trianglearea(rawCam1[N].headpt,rawCam1[N].tailpt,rawCam1[N].ebox.center)*2.00/distance_p
tsf1(rawCam1[N].headpt,rawCam1[N].tailpt));
rawCam2[N].we=fabs(trianglearea(rawCam2[N].headpt,rawCam2[N].tailpt,rawCam2[N].ebox.center)*2.00/distance_p
tsf1(rawCam2[N].headpt,rawCam2[N].tailpt));
rawCam3[N].we=fabs(trianglearea(rawCam3[N].headpt,rawCam3[N].tailpt,rawCam3[N].ebox.center)*2.00/distance_p
tsf1(rawCam3[N].headpt,rawCam3[N].tailpt));
//cout<<rawCam1[N].we<<","<<rawCam2[N].we<<","<<rawCam3[N].we<<endl;

if(choice==3)
{
    distance=rawCam3[N].ebox.center.x-float(xo3);
    if(distance<0) actualscaling=1/(1+distance/200);
    else if(distance>0) actualscaling=1+(distance/200);
    yaw=actualscaling/apparentscaling;
    pitch=rawCam1[N].angle;
    roll=acos(GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling));
    //cout<<"\n*"<<GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling);
    //cout<<"Choice 3 : "<<roll*180/pi<<","<<pitch*180/pi<<","<<yaw*180/pi<<endl;
}

else if(choice==1)
{
    distance=rawCam3[N].ebox.center.x-float(xo1);
    if(distance<0) actualscaling=1/(1+distance/200);
    else if(distance>0) actualscaling=1+(distance/200);
    yaw=actualscaling/apparentscaling;
    pitch=rawCam1[N].angle;
    roll=acos(GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling));
    //cout<<"\n*"<<GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling);
    //cout<<"Choice 1 : "<<roll*180/pi<<","<<pitch*180/pi<<","<<yaw*180/pi<<endl;
}

else if(choice==2)
{
    distance=rawCam3[N].ebox.center.y-float(yo2);
    if(distance<0) actualscaling=1/(1+distance/200);
    else if(distance>0) actualscaling=1+(distance/200);
    yaw=actualscaling/apparentscaling;

```

```

        pitch=rawCam1[N].angle;
        roll=acos(GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling));
        //cout<<"\n*"<<GlobalTriArea/(CurrentTriArea *actualscaling *apparentscaling);
        //cout<<"Choice 2 : "<<roll*180/pi<<" , "<<pitch*180/pi<<" , "<<yaw*180/pi<<endl;

    }

else cout<<"\nCannot calculate for this frame!\n";
}

void processmap()
{

int L=0,B=0,H=0;

struct sizes
{
    int XX;
    int YY;
    int missing; //1 - L, 2 - B, 3 - H
}surf1,surf2,surf3;

//int offset1x,offset1y,offset2x,offset2y,offset3x,offset3y;

if(rawCam1[N].bbox.height*rawCam1[N].bbox.width > rawCam2[N].bbox.height*rawCam2[N].bbox.width &&
rawCam1[N].bbox.height*rawCam1[N].bbox.width > rawCam3[N].bbox.height*rawCam3[N].bbox.width)
choice=1;
else if(rawCam2[N].bbox.height*rawCam2[N].bbox.width > rawCam1[N].bbox.height*rawCam1[N].bbox.width
&& rawCam2[N].bbox.height*rawCam2[N].bbox.width > rawCam3[N].bbox.height*rawCam3[N].bbox.width)
choice=2;
else choice=3;
float scalep=0;
float scalepp=0;
//choice=2;
choice=4;
//cout<<choice<<" .";
CvSeq* c;
CvPoint *alongquad1;
CvPoint dummyquad1=cvPoint(0,0);
int **quadsurf1=NULL;
CvPoint * A1=NULL;
int *done_checking_index1=NULL;
int count_quadsurf1=0;
int count_alongquad1=0;

CvPoint *alongquad2;
CvPoint dummyquad2=cvPoint(0,0);
int **quadsurf2=NULL;
CvPoint * A2=NULL;
int *done_checking_index2=NULL;
int count_quadsurf2=0;
int count_alongquad2=0;

CvPoint *alongquad3;
CvPoint dummyquad3=cvPoint(0,0);
int **quadsurf3=NULL;

```



```

CvPoint * A3=NULL;
int *done_checking_index3=NULL;
int count_quadsurf3=0;
int count_alongquad3=0;

int A11=0,A22=0,B11=0,B22=0,C11=0,C22=0;
int ***volex=NULL;
int ***volex1=NULL;
int ***volex2=NULL;
int ***volex3=NULL;

alongquad1=new CvPoint[500];
alongquad2=new CvPoint[500];
alongquad3=new CvPoint[500];

for(int i=0;i<500;i++)
{
    alongquad1[i].x=0;
    alongquad1[i].y=0;

    alongquad2[i].x=0;
    alongquad2[i].y=0;

    alongquad3[i].x=0;
    alongquad3[i].y=0;
}

choice=4;
if(rawCam1[N].bbox.width>rawCam2[N].bbox.width)L=rawCam1[N].bbox.width;
else L=rawCam2[N].bbox.width;
//
if(rawCam1[N].bbox.height>rawCam3[N].bbox.height)B=rawCam1[N].bbox.height;
else B=rawCam3[N].bbox.height;
//
if(rawCam2[N].bbox.height>rawCam3[N].bbox.width)H=rawCam2[N].bbox.height;
else H=rawCam3[N].bbox.width;
//
sub_choice=4;

scalep=float(rawCam3[N].bbox.width)/float(B); //cam3
scalepp=float(rawCam1[N].bbox.height)/float(L); //cam1

surf1.XX=L;surf1.YY=B;surf1.missing=3;
surf2.XX=L;surf2.YY=H;surf2.missing=1;
surf3.XX=H;surf3.YY=B;surf3.missing=2;

L++;
B++;
H++;
quadsurf1=new int*[L];
for(int i=0;i<L;i++)
{
    quadsurf1[i]=new int[B];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        quadsurf1[i][j]=0;
    }
}

```

```

}

quadsurf2=new int*[L];
for(int i=0;i<L;i++)
{
    quadsurf2[i]=new int[H];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<H;j++)
    {
        quadsurf2[i][j]=0;
    }
}

quadsurf3=new int*[H];
for(int i=0;i<H;i++)
{
    quadsurf3[i]=new int[B];
}
for(int i=0;i<H;i++)
{
    for(int j=0;j<B;j++)
    {
        quadsurf3[i][j]=0;
    }
}

A11=rawCam1[N].bdfbox.x+L;
A22=rawCam1[N].bdfbox.y+B;

B11=rawCam2[N].bdfbox.x+L;
B22=rawCam2[N].bdfbox.y+H;

C11=rawCam3[N].bdfbox.x+H;
C22=rawCam3[N].bdfbox.y+B;

//}

volex=new int**[L];
for(int i=0;i<L;i++)
{
    volex[i]=new int*[B];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        volex[i][j]=new int[H];
    }
}

//-----
//initialize
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        for(int k=0;k<H;k++)
        {
            volex[i][j][k]=0;

```

```

    }
}

for(c=quad1;c!=0;c=c->h_next)
{
    CvSeqReader reader1;
    cvStartReadSeq( c, &reader1, 0 );

    for ( int i = 0; i < c->total; i++ )
    {
        CV_READ_SEQ_ELEM (dummyquad1, reader1);
        if(dummyquad1.x>rawCam1[N].bdbox.x    &&    dummyquad1.y>rawCam1[N].bdbox.y    &&
dummyquad1.x<A11 && dummyquad1.y<A22)
        {
            alongquad1[count_alongquad1].x=dummyquad1.x;
            alongquad1[count_alongquad1++].y=dummyquad1.y;
        }
    }

    int t=0;
    int x=0;
    int y=0;
    c=NULL;
    while(t<count_alongquad1)
    {
        // y const scanning
        //use bounding box and mark 1 for a projection of a vorex point in the bounding box coordinates
        if( (alongquad1[t].x - rawCam1[N].bdbox.x)<L && (alongquad1[t].y-rawCam1[N].bdbox.y)<B){
            quadsurf1[alongquad1[t].x - rawCam1[N].bdbox.x][alongquad1[t].y-rawCam1[N].bdbox.y]=1;}
        t++;
    }

    for(c=quad2;c!=0;c=c->h_next)
    {
        CvSeqReader reader2;
        cvStartReadSeq( c, &reader2, 0 );

        for ( int i = 0; i < c->total; i++ )
        {
            CV_READ_SEQ_ELEM (dummyquad2, reader2);
            if(dummyquad2.x>rawCam2[N].bdbox.x    &&    dummyquad2.y>rawCam2[N].bdbox.y    &&
dummyquad2.x<B11 && dummyquad2.y<B22 )
            {
                alongquad2[count_alongquad2].x=dummyquad2.x;
                alongquad2[count_alongquad2++].y=dummyquad2.y;
            }
        }

        t=0;
        x=0;
        y=0;
        c=NULL;
        while(t<count_alongquad2)
        {
            // y const scanning
            //use bounding box and mark 1 for a projection of a vorex point in the bounding box coordinates
            if((alongquad2[t].x - rawCam2[N].bdbox.x)<L && (alongquad2[t].y-rawCam2[N].bdbox.y)<H){
                quadsurf2[alongquad2[t].x - rawCam2[N].bdbox.x][alongquad2[t].y-rawCam2[N].bdbox.y]=1;}
        }
    }
}

```

```

        t++;
    }

for(c=quad3;c!=0;c=c->h_next)
{
    CvSeqReader reader3;
    cvStartReadSeq( c, &reader3, 0 );

    for ( int i = 0; i < c->total; i++ )
    {
        CV_READ_SEQ_ELEM (dummyquad3, reader3);
        if(dummyquad3.x>rawCam3[N].bdbox.x    &&    dummyquad3.y>rawCam3[N].bdbox.y    &&
dummyquad3.x<C11    &&    dummyquad3.y<C22)
        {
            alongquad3[count_alongquad3].x=dummyquad3.x;
            alongquad3[count_alongquad3++].y=dummyquad3.y;
        }
    }
}
t=0;
x=0;
y=0;
while(t<count_alongquad3)
{
    // y const scanning
    //use bounding box and mark 1 for a projection of a vorex point in the bounding box coordinates
    if((alongquad3[t].x - rawCam3[N].bdbox.x)<H && (alongquad3[t].y-rawCam3[N].bdbox.y)<B){
        quadsurf3[alongquad3[t].x - rawCam3[N].bdbox.x][alongquad3[t].y-rawCam3[N].bdbox.y]=1;}

    t++;
}
//point polygon test fill
CvPoint2D32f double_point;
//1

for(int i=0;i<surf1.XX;i++)
{
    for(int j=0;j<surf1.YY;j++)
    {
        double_point=cvPoint2D32f(double(rawCam1[N].bdbox.x + i),double(rawCam1[N].bdbox.y + j));
        if(cvPointPolygonTest(quad1,double_point,0)>=0)
        {quadsurf1[i][j]=1; /*cout<<"("<i<<"/"<<surf1.XX<<","<<j<<"/"<<surf1.YY<<")";*/}
    }
}

//2

for(int i=0;i<surf2.XX;i++)
{
    for(int j=0;j<surf2.YY;j++)
    {
        double_point=cvPoint2D32f(double(rawCam2[N].bdbox.x + i),double(rawCam2[N].bdbox.y + j));
        if(cvPointPolygonTest(quad2,double_point,0)>=0)
        {quadsurf2[i][j]=1; /*cout<<cvPointPolygonTest(quad2,double_point,0)<<" -
"<<i<<"/"<<surf2.XX<<","<<j<<"/"<<surf2.YY<<")..";*/;}
    }
}

```

```

    }
}
//3

for(int i=0;i<surf3.XX;i++)
{
    for(int j=0;j<surf3.YY;j++)
    {
        double_point=cvPoint2D32f(double(rawCam3[N].bdbox.x + i),double(rawCam3[N].bdbox.y + j));
        if(cvPointPolygonTest(quad3,double_point,0)>=0) {quadsurf3[i][j]=1;}
    }
}
//HIT BC METHOD
int hit_bc=-1;
int bound=0;
//-----
for(int j=1;j<surf1.YY-2;j++)
{
    hit_bc=-1;
    bound=0;
    for(int k=0;k<surf1.XX-2;k++) {if(quadsurf1[k][j]==1) bound++;}
    if(bound!=1)
    {
        for(int i=1;i<surf1.XX-2;i++)
        {
            if(quadsurf1[i][j]+quadsurf1[i+1][j]+quadsurf1[i-1][j]==1 && quadsurf1[i][j]==1) {hit_bc*=-1;}
            else if(quadsurf1[i][j]+quadsurf1[i+1][j]+quadsurf1[i-1][j]==2 && quadsurf1[i+1][j]==0) hit_bc=-1;
            else if(quadsurf1[i][j]+quadsurf1[i+1][j]+quadsurf1[i-1][j]==2 && quadsurf1[i-1][j]==0) hit_bc=1;
            else if(quadsurf1[i][j]==0 && hit_bc==1) quadsurf1[i][j]=1;
            else if(quadsurf1[i][j]==0 && hit_bc==-1) quadsurf1[i][j]=0;
        }
    }
    else if(bound==1)
    {
        for(int k=0;k<surf1.XX-2;k++) {quadsurf1[k][j]=quadsurf1[k][j-1];}
    }
}
//-----
for(int j=1;j<surf2.YY-2;j++)
{
    hit_bc=-1;
    bound=0;
    for(int k=0;k<surf2.XX-2;k++) {if(quadsurf2[k][j]==1) bound++;}
    if(bound!=1)
    {
        for(int i=1;i<surf2.XX-1;i++)
        {
            if(quadsurf2[i][j]+quadsurf2[i+1][j]+quadsurf2[i-1][j]==1 && quadsurf2[i][j]==1) hit_bc*=-1;
            else if(quadsurf2[i][j]+quadsurf2[i+1][j]+quadsurf2[i-1][j]==2 && quadsurf2[i+1][j]==0) hit_bc=-1;
            else if(quadsurf2[i][j]+quadsurf2[i+1][j]+quadsurf2[i-1][j]==2 && quadsurf2[i-1][j]==0) hit_bc=1;
            else if(quadsurf2[i][j]==0 && hit_bc==1) quadsurf2[i][j]=1;
            else if(quadsurf2[i][j]==0 && hit_bc==-1) quadsurf2[i][j]=0;
        }
    }
    else if(bound==1)
    {
        for(int k=0;k<surf2.XX-2;k++) {quadsurf2[k][j]=quadsurf2[k][j-1];}
    }
}

```

```

}
}
//-----
for(int j=1;j<surf3.YY-2;j++)
{
hit_bc=-1;
bound=0;
for(int k=0;k<surf3.XX-2;k++) {if(quadsurf3[k][j]==1) bound++;}
if(bound!=1)
{
for(int i=1;i<surf3.XX-1;i++)
{
if(quadsurf3[i][j]+quadsurf3[i+1][j]+quadsurf3[i-1][j]==1 && quadsurf3[i][j]==1) hit_bc*=-1;
else if(quadsurf3[i][j]+quadsurf3[i+1][j]+quadsurf3[i-1][j]==2 && quadsurf3[i+1][j]==0) hit_bc=-
1;
else if(quadsurf3[i][j]+quadsurf3[i+1][j]+quadsurf3[i-1][j]==2 && quadsurf3[i-1][j]==0) hit_bc=1;
else if(quadsurf3[i][j]==0 && hit_bc==1) quadsurf3[i][j]=1;
else if(quadsurf3[i][j]==0 && hit_bc==-1) quadsurf3[i][j]=0;
}
}
else if(bound==1)
{
for(int k=0;k<surf3.XX-1;k++) {quadsurf3[k][j]=quadsurf3[k][j-1];}
}
}
//-----
//now fill in converse direction

//ENDPOINT METHOD
int p1=0;
int p2=0;
int bnd1chk=0;

//-----
//SURF1
for(int i=1;i<surf1.XX;i++)
{
for(int j=1;j<surf1.YY;j++)
{
if(bnd1chk==0 && quadsurf1[i][j]==1){bnd1chk=1;p1=j;}
else if(bnd1chk==1 && quadsurf1[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf1[i][j]=1;} }
}
}
//SURF2
bnd1chk=0;
p1=0;
p2=0;
for(int i=1;i<surf2.XX;i++)
{
for(int j=1;j<surf2.YY;j++)
{
if(bnd1chk==0 && quadsurf2[i][j]==1){bnd1chk=1;p1=j;}
else if(bnd1chk==1 && quadsurf2[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf2[i][j]=1;} }
}
}

```

```

    }
} //SURF3
bnd1chk=0;
p1=0;
p2=0;
for(int i=1;i<surf3.XX;i++)
{
    for(int j=1;j<surf3.YY;j++)
    {
        if(bnd1chk==0 && quadsurf3[i][j]==1){bnd1chk=1;p1=j;}
        else if(bnd1chk==1 && quadsurf3[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf3[i][j]=1; }
    }
}
} //SURF1
for(int j=1;j<surf1.YY;j++)
{
    for(int i=1;i<surf1.XX;i++)
    {
        if(bnd1chk==0 && quadsurf1[i][j]==1){bnd1chk=1;p1=j;}
        else if(bnd1chk==1 && quadsurf1[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf1[i][j]=1; }
    }
}
} //SURF2
bnd1chk=0;
p1=0;
p2=0;
for(int j=1;j<surf2.YY;j++)
{
    for(int i=1;i<surf2.XX;i++)
    {
        if(bnd1chk==0 && quadsurf2[i][j]==1){bnd1chk=1;p1=j;}
        else if(bnd1chk==1 && quadsurf2[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf2[i][j]=1; }
    }
}
} //SURF3
bnd1chk=0;
p1=0;
p2=0;
for(int j=1;j<surf3.YY;j++)
{
    for(int i=1;i<surf3.XX;i++)
    {
        if(bnd1chk==0 && quadsurf3[i][j]==1){bnd1chk=1;p1=j;}
        else if(bnd1chk==1 && quadsurf3[i][j]==1) {p2=j;bnd1chk=0; for(int
j=p1;j<p2;j++){quadsurf3[i][j]=1; }
    }
}
}
if(writeall==1)
{

```

```

        ofstream f11("abc1.asc");
for(int i=0;i<surf1.XX;i++)
{
    for(int j=0;j<surf1.YY;j++)
    {
        if(quadsurf1[i][j]==1) f11<<i<<" "<<j<<"\n";
    }
}
f11.close();

        ofstream f22("abc2.asc");
for(int i=0;i<surf2.XX;i++)
{
    for(int j=0;j<surf2.YY;j++)
    {
        if(quadsurf2[i][j]==1) f22<<i<<" "<<j<<"\n";
    }
}
f22.close();

        ofstream f33("abc3.asc");
for(int i=0;i<surf3.XX;i++)
{
    for(int j=0;j<surf3.YY;j++)
    {
        if(quadsurf3[i][j]==1) f33<<i<<" "<<j<<"\n";
    }
}
f33.close();
}

//cout<<sub_choice<<" ";
cout<<"(L,B,H)="<<L<<" "<<B<<" "<<H<<" ";

//extrude these created surfaces to form volumes volex 1 volex2 and volex3

//First dynamically create memory

//for 1
volex1=new int*[L];
for(int i=0;i<L;i++)
{
    volex1[i]=new int*[B];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        volex1[i][j]=new int[H];
    }
}
//initialize
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        for(int k=0;k<H;k++)
        {
            volex1[i][j][k]=0;
        }
    }
}
}

```



```

//for 2
volex2=new int**[L];
for(int i=0;i<L;i++)
{
    volex2[i]=new int*[B];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        volex2[i][j]=new int[H];
    }
}
//initialize
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        for(int k=0;k<H;k++)
        {
            volex2[i][j][k]=0;
        }
    }
}

//for 3
volex3=new int**[L];
for(int i=0;i<L;i++)
{
    volex3[i]=new int*[B];
}
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        volex3[i][j]=new int[H];
    }
}
//initialize
for(int i=0;i<L;i++)
{
    for(int j=0;j<B;j++)
    {
        for(int k=0;k<H;k++)
        {
            volex3[i][j][k]=0;
        }
    }
}

```

//depending on "choice" variable build the volume in each subspace

```

if(choice==1 || choice==4)
{
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {

```

```

        for(int k=0;k<H;k++)
        {
            // LB,LH,HB (ij)
            vorex1[i][j][k]=quadsurf1[i][j];
            vorex2[i][j][k]=quadsurf2[i][k];
            vorex3[i][j][k]=quadsurf3[k][j];
        }
    }
}

else if(choice==2)
{
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {
            for(int k=0;k<H;k++)
            {
                // BH,BL,LH
                vorex1[i][j][k]=quadsurf1[j][k];
                vorex2[i][j][k]=quadsurf2[j][i];
                vorex3[i][j][k]=quadsurf3[i][k];
            }
        }
    }
}

else if(choice==3)
{
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {
            for(int k=0;k<H;k++)
            {
                // HL,HB,BL
                vorex1[i][j][k]=quadsurf1[k][i];
                vorex2[i][j][k]=quadsurf2[k][j];
                vorex3[i][j][k]=quadsurf3[j][i];
            }
        }
    }
}

else cout<<"No choice made..!"<<endl;
long int icount=0;
for(int i=2;i<L-2;i++)
{
    for(int j=2;j<B-2;j++)
    {
        for(int k=2;k<H-2;k++)
        {
            vorex[i][j][k]=vorex1[i][j][k]+vorex2[i][j][k]+vorex3[i][j][k];
            if(vorex[i][j][k]==3) {icount++;}
        }
    }
}

```

```

}
//SAMPLE WRITE FILE !!!!
if(writeall==1)
{
    ofstream f1("voxel1.asc");
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {
            for(int k=0;k<H;k++)
            {
                if(volex1[i][j][k]==1) f1<<i<<" "<<j<<" "<<k<<endl;
            }
        }
    }
    f1.close();

    ofstream f2("voxel2.asc");
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {
            for(int k=0;k<H;k++)
            {
                if(volex2[i][j][k]==1) f2<<i<<" "<<j<<" "<<k<<endl;
            }
        }
    }
    f2.close();

    ofstream f3("voxel3.asc");
    for(int i=0;i<L;i++)
    {
        for(int j=0;j<B;j++)
        {
            for(int k=0;k<H;k++)
            {
                if(volex3[i][j][k]==1) f3<<i<<" "<<j<<" "<<k<<endl;
            }
        }
    }
    f3.close();

    if(writeall==1)
    {
        ofstream f4("voxel.asc");
        for(int i=0;i<L;i++)
        {
            for(int j=0;j<B;j++)
            {
                for(int k=0;k<H;k++)
                {
                    if(volex[i][j][k]==3) f4<<i<<" "<<j<<" "<<k<<endl;
                }
            }
        }
        f4.close();
    }

    cout<<"{"<<icount<<"}/"<<long(L*B*H)<<" ";//getchar();
}

```

```

if(icount>5000) {plot=1;iterate_for_roll(L,B,H,volex);}
else cout<<"Not enough voxels available!\n";
plot=0;
//getchar();
}

void iterate_for_roll(int L,int B,int H,int *** volex)
{
    global.v1.x=1;
    global.v1.y=0;
    global.v1.z=0;

    global.v2.x=0;
    global.v2.y=1;
    global.v2.z=0;

    global.v3.x=0;
    global.v3.y=0;
    global.v3.z=1;

double x1,x2,x3,x4,y1,y2,y3,y4,z1,z2,z3,z4;//y1 is local as it is already used by math.h

    int ***frame=NULL;
    int ***wireframe=NULL;
    int ***wireframe_r=NULL;
    int ***wireframe_r2=NULL;
    int ***wireframe_r3=NULL;
    int ***wireframe_t=NULL;
    int ***wireframe_final=NULL;
    int **centroids=NULL;

centroids=new int *[L];
for(int i=0;i<L;i++)
{
    centroids[i]=new int [2];
}
//initialize
for(int i=0;i<L;i++)
{
    for(int j=0;j<2;j++)
    {
        centroids[i][j]=0;
    }
}

for(int i=0;i<L;i++)
{
    int A=0,xc=0,yc=0;
    for(int j=0;j<B;j++)
    {
        for(int k=0;k<H;k++)
        {
            if(volex[i][j][k]==3)
            {
                A=A+1;
                xc=xc+j*1;
                yc=yc+k*1;
            }
        }
    }
}

```

```

if(A==0) {xc=B/2; yc=B/2;}
else
{
xc=int(float(xc)/float(A));
yc=int(float(yc)/float(A));
centroids[i][0]=xc;
centroids[i][1]=yc;
}
//cout<<i<<"-"<<xc<<","<<yc<<endl;
}
//getchar();
//cout<<"3..";
int body_start_volex=0;
int body_end_volex=0;
//find out which is the first and last planes with points
int check=0;
for(int i=0;i<L;i++)
{
for(int j=0;j<B;j++)
{
for(int k=0;k<H;k++)
{
if(volex[i][j][k]==1) check++;
}
}
if(check!=0){ body_start_volex=i; i=L;}
}
check=0;
for(int i=L-1;i>=0;i--)
{
for(int j=B-1;j>=0;j--)
{
for(int k=H-1;k>=0;k--)
{
if(volex[i][j][k]==1) check++;
}
}
if(check!=0){ body_end_volex=i; i=-1;}
}

body_rot.x=body_end_volex-body_start_volex;
body_rot.y=centroids[body_end_volex][0] - centroids[body_start_volex][0];
body_rot.z=centroids[body_end_volex][1] - centroids[body_start_volex][1];
double length11=sqrt(pow(body_rot.x,2) + pow(body_rot.y,2) + pow(body_rot.z,2));
body_rot.x/=length11;
body_rot.y/=length11;
body_rot.z/=length11;

//body is now at 0,B/2,H/2 to L,B/2,H/2

body.x=1; // from L/L
body.y=0;
body.z=0;

//cross product
axis.x=body.y*body_rot.z - body.z*body_rot.y;
axis.y=body.z*body_rot.x - body.x*body_rot.z;
axis.z=body.x*body_rot.y - body.y*body_rot.x;

length11=sqrt(pow(axis.x,2) + pow(axis.y,2) + pow(axis.z,2));

```

```

axis.x=axis.x/length11;
axis.y=axis.y/length11;
axis.z=axis.z/length11;

roll_axis.x=body_rot.x;
roll_axis.y=body_rot.y;
roll_axis.z=body_rot.z;

float degrees=0;
float theta=0;//15*3.14156/180;
float c1=0;//cos(theta);
float s1=0;//sin(theta);
float t1=0;//1-c1;
float conv=0.00001;

float theta_max=0;
int icount=0;
int icount_max=0;
int exit=0;
int z_tail_end=body_end_volex;
int z_tail_start=int(float(body_start_volex)+0.333*float(body_end_volex-body_start_volex)); //guess

float m_0=(centroids[z_tail_start][0]-centroids[body_start_volex][0])/(z_tail_start-body_start_volex);
float m_1=(centroids[z_tail_start][1]-centroids[body_start_volex][1])/(z_tail_start-body_start_volex);
int p_count=0;
int x_imp=0;
int y_imp=0;
//
for(int i=body_start_volex;i<=z_tail_start;i++)
{
x_imp+=centroids[i][0]+m_0*(i-body_start_volex);
y_imp+=centroids[i][1]+m_1*(i-body_start_volex);
p_count++;
}
centroid_plane.x=z_tail_start-body_start_volex;
centroid_plane.y=x_imp/p_count;
centroid_plane.z=y_imp/p_count;
//
//
//
temp1.x=z_tail_start-body_start_volex;
temp1.y=centroids[z_tail_start][0]-centroids[body_start_volex][0];
temp1.z=centroids[z_tail_start][1]-centroids[body_start_volex][1];
//
temp2.x=centroid_plane.y*temp1.z - centroid_plane.z*temp1.y;
temp2.y=centroid_plane.z*temp1.x - centroid_plane.x*temp1.z;
temp2.z=centroid_plane.x*temp1.y - centroid_plane.y*temp1.x;
//
main1.x=temp2.y*temp1.z - temp2.z*temp1.y;
main1.y=temp2.z*temp1.x - temp2.x*temp1.z;
main1.z=temp2.x*temp1.y - temp2.y*temp1.x;
//
main2.x=temp1.x;
main2.y=temp1.y;
main2.z=temp1.z;
//
main3.x=main2.y*main1.z - main2.z*main1.y;
main3.y=main2.z*main1.x - main2.x*main1.z;
main3.z=main2.x*main1.y - main2.y*main1.x;
//

```

```

//normalize main vectors (body coord system)
//
float length_centr=sqrt(pow(main1.x,2) + pow(main1.y,2) + pow(main1.z,2));
main1.x/=-length_centr;
main1.y/=-length_centr;
main1.z/=-length_centr;
//
length_centr=sqrt(pow(main2.x,2) + pow(main2.y,2) + pow(main2.z,2));
main2.x/=length_centr;
main2.y/=length_centr;
main2.z/=length_centr;

length_centr=sqrt(pow(main3.x,2) + pow(main3.y,2) + pow(main3.z,2));
main3.x/=length_centr;
main3.y/=length_centr;
main3.z/=length_centr;

initial.v1=main1;
initial.v2=main2;
initial.v3=main3;

//convert orientation corresponding to choice made initially
double temp11,temp22,temp33,temp44,temp55,temp66,temp77,temp88,temp99;

temp11=initial.v1.x;
temp22=initial.v1.y;
temp33=initial.v1.z;
temp44=initial.v2.x;
temp55=initial.v2.y;
temp66=initial.v2.z;
temp77=initial.v3.x;
temp88=initial.v3.y;
temp99=initial.v3.z;

if(choice==1 || choice==4)
{
    if(sub_choice==1 || sub_choice==4)
    {
        initial.v1.z=temp11;
        initial.v1.y=temp33;
        initial.v1.x=temp22;
        initial.v2.z=temp44;
        initial.v2.y=temp66;
        initial.v2.x=temp55;
        initial.v3.z=temp77;
        initial.v3.y=temp99;
        initial.v3.x=temp88;
    }
    else if(sub_choice==2)
    {
        initial.v1.z=temp22;
        initial.v1.y=temp33;
        initial.v1.x=temp11;
        initial.v2.z=temp55;
        initial.v2.y=temp66;
        initial.v2.x=temp44;
        initial.v3.z=temp88;
        initial.v3.y=temp99;
        initial.v3.x=temp77;
    }
}

```

```

}
//
else if(choice==2)
{
    if(sub_choice==1)
    {
        initial.v1.z=temp33;
        initial.v1.y=temp22;
        initial.v1.x=temp11;
        initial.v2.z=temp66;
        initial.v2.y=temp55;
        initial.v2.x=temp44;
        initial.v3.z=temp99;
        initial.v3.y=temp88;
        initial.v3.x=temp77;
    }
    else if(sub_choice==2)
    {
        initial.v1.z=temp11;
        initial.v1.y=temp33;
        initial.v1.x=temp22;
        initial.v2.z=temp44;
        initial.v2.y=temp66;
        initial.v2.x=temp55;
        initial.v3.z=temp77;
        initial.v3.y=temp99;
        initial.v3.x=temp88;
    }
}
//
else if(choice==3)
{
    if(sub_choice==1)
    {
        initial.v1.z=temp22;
        initial.v1.y=temp11;
        initial.v1.x=temp33;
        initial.v2.z=temp55;
        initial.v2.y=temp44;
        initial.v2.x=temp66;
        initial.v3.z=temp88;
        initial.v3.y=temp77;
        initial.v3.x=temp99;
    }
    else if(sub_choice==2)
    {
        initial.v1.z=temp22;
        initial.v1.y=temp33;
        initial.v1.x=temp11;
        initial.v2.z=temp44;
        initial.v2.y=temp66;
        initial.v2.x=temp55;
        initial.v3.z=temp77;
        initial.v3.y=temp99;
        initial.v3.x=temp88;
    }
}
//
float v1_length=sqrt(pow(initial.v1.x,2) + pow(initial.v1.y,2) + pow(initial.v1.z,2));
initial.v1.x/=v1_length;
initial.v1.y/=v1_length;

```



```

initial.v1.z/=v1_length;
//

float v3_length=sqrt(pow(initial.v3.x,2) + pow(initial.v3.y,2) + pow(initial.v3.z,2));
initial.v3.x/=v3_length;
initial.v3.y/=v3_length;
initial.v3.z/=v3_length;
float v2_length=sqrt(pow(initial.v2.x,2) + pow(initial.v2.y,2) + pow(initial.v2.z,2));
initial.v2.x/=v2_length;
initial.v2.y/=v2_length;
initial.v2.z/=v2_length;
//
if(writeall==1)
{
ofstream axis33("axis3beforeR.asc");
//
for(int u=0;u<100;u++)
{
axis33<<u*initial.v1.x<<" "<<u*initial.v1.y<<" "<<u*initial.v1.z<<endl;
axis33<<2*u*initial.v2.x<<" "<<2*u*initial.v2.y<<" "<<2*u*initial.v2.z<<endl;
axis33<<3*u*initial.v3.x<<" "<<3*u*initial.v3.y<<" "<<3*u*initial.v3.z<<endl;

}
axis33.close();
}
//YAW*****
//about v1
initial_dummy.v1=initial.v1;
initial_dummy.v2=initial.v2;
initial_dummy.v3=initial.v3;
initial_dummy2.v1=initial.v1;
initial_dummy2.v2=initial.v2;
initial_dummy2.v3=initial.v3;
//

for(double degrees=-180;degrees<180;degrees=degrees+0.001)
{

icount=0;
theta=(float(degrees)/180.0)*3.14156;
c1=cos(theta);
s1=sin(theta);
t1=1-c1;
//rotation matrix
double R[3][3]={t1*initial.v1.x*initial.v1.x + c1*t1*initial.v1.x*initial.v1.y - s1*initial.v1.z,t1*initial.v1.x*initial.v1.z
+ s1*initial.v1.y,
t1*initial.v1.x*initial.v1.y + s1*initial.v1.z,t1*initial.v1.y*initial.v1.y +
c1,t1*initial.v1.y*initial.v1.z - s1*initial.v1.x,
t1*initial.v1.x*initial.v1.z - s1*initial.v1.y,t1*initial.v1.y*initial.v1.z +
s1*initial.v1.x,t1*initial.v1.z + c1};
//
initial_dummy2.v2.x=(R[0][0]*initial.v2.x + R[0][1]*initial.v2.y + R[0][2]*initial.v2.z);
initial_dummy2.v2.y=(R[1][0]*initial.v2.x + R[1][1]*initial.v2.y + R[1][2]*initial.v2.z);
initial_dummy2.v2.z=(R[2][0]*initial.v2.x + R[2][1]*initial.v2.y + R[2][2]*initial.v2.z);
//
initial_dummy2.v3.x=(R[0][0]*initial.v3.x + R[0][1]*initial.v3.y + R[0][2]*initial.v3.z);
initial_dummy2.v3.y=(R[1][0]*initial.v3.x + R[1][1]*initial.v3.y + R[1][2]*initial.v3.z);
initial_dummy2.v3.z=(R[2][0]*initial.v3.x + R[2][1]*initial.v3.y + R[2][2]*initial.v3.z);
//

angle_yaw=degrees;
if(initial_dummy2.v3.x>=-conv && initial_dummy2.v3.x<=conv && initial_dummy2.v3.z>0)
{

```

```

        initial.v2=initial_dummy2.v2;
        initial.v3=initial_dummy2.v3;
*/break;
}
else{initial.v1=initial_dummy.v1;initial.v2=initial_dummy.v2;initial.v3=initial_dummy.v3;}
}
if(writeall==1)
{
ofstream axis34("axis3afterY.asc");

for(int u=0;u<100;u++)
{
axis34<<u*initial.v1.x<<","<<u*initial.v1.y<<","<<u*initial.v1.z<<endl;
axis34<<2*u*initial.v2.x<<","<<2*u*initial.v2.y<<","<<2*u*initial.v2.z<<endl;
axis34<<3*u*initial.v3.x<<","<<3*u*initial.v3.y<<","<<3*u*initial.v3.z<<endl;

}
axis34.close();
}
//*****
//PITCH*****
//about v3
initial_dummy.v1=initial.v1;
initial_dummy.v2=initial.v2;
initial_dummy.v3=initial.v3;

initial_dummy2.v1=initial.v1;
initial_dummy2.v2=initial.v2;
initial_dummy2.v3=initial.v3;

for(double degrees=-180;degrees<180;degrees=degrees+0.001)

{

icount=0;
theta=(float(degrees)/180.0)*3.14156;
c1=cos(theta);
s1=sin(theta);
t1=1-c1;
//rotation matrix
double R[3][3]={t1*initial.v3.x*initial.v3.x + c1,t1*initial.v3.x*initial.v3.y - s1*initial.v3.z,t1*initial.v3.x*initial.v3.z
+ s1*initial.v3.y,
                    t1*initial.v3.x*initial.v3.y + s1*initial.v3.z,t1*initial.v3.y*initial.v3.y +
c1,t1*initial.v3.y*initial.v3.z - s1*initial.v3.x,
                    t1*initial.v3.x*initial.v3.z- s1*initial.v3.y,t1*initial.v3.y*initial.v3.z +
s1*initial.v3.x,t1*initial.v3.z*initial.v3.z + c1};
//
initial_dummy2.v2.x=(R[0][0]*initial.v2.x + R[0][1]*initial.v2.y + R[0][2]*initial.v2.z);
initial_dummy2.v2.y=(R[1][0]*initial.v2.x + R[1][1]*initial.v2.y + R[1][2]*initial.v2.z);
initial_dummy2.v2.z=(R[2][0]*initial.v2.x + R[2][1]*initial.v2.y + R[2][2]*initial.v2.z);
//
initial_dummy2.v1.x=(R[0][0]*initial.v1.x + R[0][1]*initial.v1.y + R[0][2]*initial.v1.z);
initial_dummy2.v1.y=(R[1][0]*initial.v1.x + R[1][1]*initial.v1.y + R[1][2]*initial.v1.z);
initial_dummy2.v1.z=(R[2][0]*initial.v1.x + R[2][1]*initial.v1.y + R[2][2]*initial.v1.z);
//

angle_pitch=degrees;
if(initial_dummy2.v1.x>=-conv && initial_dummy2.v1.x<=conv)
{
initial.v1=initial_dummy2.v1;
initial.v2=initial_dummy2.v2;

```

```

*/break;
}
else{initial.v1=initial_dummy.v1;initial.v2=initial_dummy.v2;initial.v3=initial_dummy.v3;}

}
//*****
if(writeall==1)
{
ofstream axis35("axis3afterP.asc");

for(int u=0;u<100;u++)
{
axis35<<u*initial.v1.x<<","<<u*initial.v1.y<<","<<u*initial.v1.z<<endl;
axis35<<2*u*initial.v2.x<<","<<2*u*initial.v2.y<<","<<2*u*initial.v2.z<<endl;
axis35<<3*u*initial.v3.x<<","<<3*u*initial.v3.y<<","<<3*u*initial.v3.z<<endl;

}
axis35.close();
}
//ROLL*****
//about v2
initial_dummy.v1=initial.v1;
initial_dummy.v2=initial.v2;
initial_dummy.v3=initial.v3;
initial_dummy2.v1=initial.v1;
initial_dummy2.v2=initial.v2;
initial_dummy2.v3=initial.v3;

//
for(double degrees=-180;degrees<180;degrees=degrees+0.001)

{

icount=0;
theta=(float(degrees)/180.0)*3.14156;
c1=cos(theta);
s1=sin(theta);
t1=1-c1;
//rotation matrix
double R[3][3]={t1*initial.v2.x*initial.v2.x + c1*t1*initial.v2.x*initial.v2.y - s1*initial.v2.z,t1*initial.v2.x*initial.v2.z
+ s1*initial.v2.y,
t1*initial.v2.x*initial.v2.y + s1*initial.v2.z,t1*initial.v2.y*initial.v2.y +
c1,t1*initial.v2.y*initial.v2.z - s1*initial.v2.x,
t1*initial.v2.x*initial.v2.z- s1*initial.v2.y,t1*initial.v2.y*initial.v2.z +
s1*initial.v2.x,t1*initial.v2.z*initial.v2.z + c1};
//
initial_dummy2.v3.x=(R[0][0]*initial.v3.x + R[0][1]*initial.v3.y + R[0][2]*initial.v3.z);
initial_dummy2.v3.y=(R[1][0]*initial.v3.x + R[1][1]*initial.v3.y + R[1][2]*initial.v3.z);
initial_dummy2.v3.z=(R[2][0]*initial.v3.x + R[2][1]*initial.v3.y + R[2][2]*initial.v3.z);
//
initial_dummy2.v1.x=(R[0][0]*initial.v1.x + R[0][1]*initial.v1.y + R[0][2]*initial.v1.z);
initial_dummy2.v1.y=(R[1][0]*initial.v1.x + R[1][1]*initial.v1.y + R[1][2]*initial.v1.z);
initial_dummy2.v1.z=(R[2][0]*initial.v1.x + R[2][1]*initial.v1.y + R[2][2]*initial.v1.z);

angle_roll=degrees;
if((initial_dummy2.v3.y>=-conv && initial_dummy2.v3.y<=conv)||((initial_dummy2.v1.z>=-conv &&
initial_dummy2.v1.z<=conv))
{
initial.v1=initial_dummy2.v1;
initial.v3=initial_dummy2.v3;
}
/*
initial_dummy2.v3.x=initial.v1.y*initial.v2.z - initial.v1.z*initial.v2.y;

```

```

initial_dummy2.v3.y=initial.v1.z*initial.v2.x - initial.v1.x*initial.v2.z;
initial_dummy2.v3.z=initial.v1.x*initial.v2.y - initial.v1.y*initial.v2.x;
initial.v3=initial_dummy2.v3;
float v3_length=sqrt(pow(initial.v2.x,2) + pow(initial.v2.y,2) + pow(initial.v2.z,2));
initial.v3.x/=v3_length;
initial.v3.y/=v3_length;
initial.v3.z/=v3_length;
*/
    break;}
else{initial.v1=initial_dummy.v1;initial.v2=initial_dummy.v2;initial.v3=initial_dummy.v3;}

}

//*****
*****

if(writeall==1)
{
ofstream axis36("axis3afterR.asc");

for(int u=0;u<100;u++)
{
    axis36<<u*initial.v1.x<<" "<<u*initial.v1.y<<" "<<u*initial.v1.z<<endl;
    axis36<<2*u*initial.v2.x<<" "<<2*u*initial.v2.y<<" "<<2*u*initial.v2.z<<endl;
    axis36<<3*u*initial.v3.x<<" "<<3*u*initial.v3.y<<" "<<3*u*initial.v3.z<<endl;

}
axis36.close();
}
if(writeall==1)
{
ofstream axis1("axis1.asc");
axis1<<"0,0,0\n";

for(int u=0;u<50;u++)
{
axis1<<u*axis.x<<" "<<u*axis.y<<" "<<u*axis.z<<endl;
axis1<<u*roll_axis.x<<" "<<u*roll_axis.y<<" "<<u*roll_axis.z<<endl;
axis1<<"0,0,"<<u<<endl;
}
axis1.close();

ofstream axis2("axis2.asc");

for(int u=0;u<500;u++)
{
axis2<<"0,0,"<<u/3<<endl;
axis2<<"0,"<<u/2<<"0\n";
axis2<<u<<"0,0\n";
}
axis2.close();

ofstream axis3("axis3.asc");

for(int u=0;u<50;u++)
{
    axis3<<u*initial.v1.x<<" "<<u*initial.v1.y<<" "<<u*initial.v1.z<<endl;
    axis3<<2*u*initial.v2.x<<" "<<2*u*initial.v2.y<<" "<<2*u*initial.v2.z<<endl;
    axis3<<3*u*initial.v3.x<<" "<<3*u*initial.v3.y<<" "<<3*u*initial.v3.z<<endl;

}
axis3.close();

```

```

ofstream main11("main1.asc");

for(int u=0;u<50;u++)
{
    main11<<u*main1.x<<" "<<u*main1.y<<" "<<u*main1.z<<endl;
    main11<<u*main2.x<<" "<<u*main2.y<<" "<<u*main2.z<<endl;
    main11<<u*main3.x<<" "<<u*main3.y<<" "<<u*main3.z<<endl;
    main11<<2*u*main1.x<<" "<<2*u*main1.y<<" "<<2*u*main1.z<<endl;
    //last statement shows the normal vector to be longer
}
main11.close();
}
if(choice==1      &&      sub_choice==1)      {angle_pitch=180+angle_pitch;angle_yaw=90-angle_yaw;
angle_roll=angle_roll+180.0;}
if(choice==1 && sub_choice==2) {}
if(choice==1 && sub_choice==3) {}

if(choice==2      &&      sub_choice==1)      {angle_yaw=angle_yaw-180;angle_pitch=-180-
angle_pitch;angle_roll=(90+angle_roll);}
if(choice==2 && sub_choice==2) {angle_yaw=-angle_yaw;angle_pitch=angle_pitch+90;angle_roll=-angle_roll;}
if(choice==2 && sub_choice==3) {}

if(choice==3 && sub_choice==1) {angle_pitch=-angle_pitch-90;angle_yaw=-angle_yaw;angle_roll=-90-angle_roll;}
if(choice==3      &&      sub_choice==2)      {angle_pitch=-1*angle_pitch;angle_yaw=-1*(90+90-
angle_yaw);angle_roll=90+angle_roll;}
if(choice==3 && sub_choice==3) {}

if(choice==4 && sub_choice==4) {angle_yaw=90+angle_yaw;angle_pitch=180+angle_pitch;angle_roll=-angle_roll;}

int ik=0;
cout<<"{Y,P,R}="<<angle_yaw<<" "<<angle_pitch<<" "<<angle_roll<<endl;
//getchar();
if(plot==1)
{
    plot_new_pitch=cvPoint(int(35.00+float((float(N-N_start)/float(N_end-N_start))*float(graphs->width-
70))),int(35.00+float((180-angle_pitch)*float(graphs->width-70)/float(360))));
    plot_new_roll=cvPoint(int(35.00+float((float(N-N_start)/float(N_end-N_start))*float(graphs->width-
70))),int(35.00+float((180-angle_roll)*float(graphs->width-70)/float(360))));
    plot_new_yaw=cvPoint(int(35.00+float((float(N-N_start)/float(N_end-N_start))*float(graphs->width-
70))),int(35.00+float((180-angle_yaw)*float(graphs->width-70)/float(360))));

    //cout<<"{"<<plot_new_pitch.x<<" "<<plot_new_pitch.y<<"\n";
    //if(fabs(distance_ptsf1(plot_new_pitch,plot_old_pitch))>0 &&
    fabs(distance_ptsf1(plot_new_pitch,plot_old_pitch)<50)
    if(distance_ptsf1(plot_new_pitch,plot_old_pitch)<50){cvLine(graphs,plot_old_pitch,plot_new_pitch,cvScala
r(0,0,155),1.5,8,0);ik++;}
    if(distance_ptsf1(plot_new_roll,plot_old_roll)<50)
    {cvLine(graphs,plot_old_roll,plot_new_roll,cvScalar(155,0,0),1.5,8,0);ik++;}
if(distance_ptsf1(plot_new_yaw,plot_old_yaw)<50)
    {cvLine(graphs,plot_old_yaw,plot_new_yaw,cvScalar(0,155,0),1.5,8,0);ik++;}

if(ik==3) A1<<float(N-N_start)/float(N_end-N_start)<<" "<<angle_pitch<<" "<<angle_yaw<<" "<<angle_roll<<endl;
    plot_old_pitch=plot_new_pitch;
    plot_old_roll=plot_new_roll;
    plot_old_yaw=plot_new_yaw;
ik=0;
//
//
tail.x=centroids[z_tail_start][0]-centroids[body_end_volex][0];
tail.y=centroids[z_tail_start][1]-centroids[body_end_volex][1];
tail.z=z_tail_start-body_end_volex;

```

```

if(choice==1 || choice==4) {xforCOM=-1*(rawCam1[N].ebox.center.x-xo1);yforCOM=-
1*(rawCam1[N].ebox.center.y-yo1);zforCOM=-1*(float(rawCam2[N].ebox.center.y-
yo2)+float(rawCam3[N].ebox.center.y-yo3))/2.0;}
else if(choice==2) {xforCOM=-1*(rawCam2[N].ebox.center.x-xo3);yforCOM=-1*(float(rawCam1[N].ebox.center.y-
yo2)+float(rawCam3[N].ebox.center.y-yo3))/2.0;zforCOM=-1*(rawCam2[N].ebox.center.y-yo2);}
else if(choice==3) {xforCOM=-1*(float(rawCam1[N].ebox.center.x-xo2)+float(rawCam2[N].ebox.center.x-
xo3))/2.0;yforCOM=-1*(rawCam3[N].ebox.center.y-yo2);zforCOM=-1*(rawCam3[N].ebox.center.x-xo2);}
//COM<<" "<<((xforCOM*1.2/90.0)-6.0)<<" "<<(((yforCOM+500.0)/300.0)*1.6 - 5.9383)<<"
"<<(((zforCOM+85.0)/50.0)-9.0)<<endl;

float ALL=sqrt( pow(xforCOM,2)+pow(yforCOM,2)+pow(zforCOM,2));
COM/*<<float(N_start)/float(N_end-N_start)<<" "*"<<xforCOM/ALL<<" "<<yforCOM/ALL<<"
"<<zforCOM/ALL<<endl;
fkpc<<"Zone I=2 Datapacking = point"<<endl<<(tail.x-xforCOM)/ALL<<" "<<(tail.y-yforCOM)/ALL<<" "<<(tail.z-
zforCOM)/ALL<<" \n"<<" "<<(xforCOM)/ALL<<" "<<(yforCOM)/ALL<<" "<<(zforCOM)/ALL<<endl;

vect_tail.x=centroids[z_tail_start][0]-centroids[body_end_volex][0];
vect_tail.y=centroids[z_tail_start][1]-centroids[body_end_volex][1];
vect_tail.z=z_tail_start-body_end_volex;

vect_thorax.x=centroids[body_start_volex][0]-centroids[z_tail_start][0];
vect_thorax.y=centroids[body_start_volex][1]-centroids[z_tail_start][1];
vect_thorax.z=body_start_volex-z_tail_start;
//cross product
//
test1.x=vect_thorax.y*vect_tail.z - vect_thorax.z*vect_tail.y;
test1.y=vect_thorax.z*vect_tail.x - vect_thorax.x*vect_tail.z;
test1.z=vect_thorax.x*vect_tail.y - vect_thorax.y*vect_tail.x;

testmag=sqrt(pow(test1.x,2) + pow(test1.y,2) + pow(test1.z,2));
test1.x/=testmag;
test1.y/=testmag;
test1.z/=testmag;

if(N==N_start) {a.x=xforCOM;a.y=yforCOM;a.z=zforCOM;}
else if(N==(N_start+N_skip)) {am.x=a.x; am.y=a.y; am.z=a.z;a.x=xforCOM;a.y=yforCOM;a.z=zforCOM;}
else if(N>=(N_start+(2*N_skip)))
{
amm.x=am.x;amm.y=am.y;amm.z=am.z; am.x=a.x;am.y=a.y;am.z=a.z;
a.x=xforCOM;a.y=yforCOM;a.z=zforCOM;
test2a.x=a.x-am.x;test2a.y=a.y-am.y;test2a.z=a.z-am.z;
test2b.x=am.x-amm.x;test2b.y=am.y-amm.y;test2b.z=am.z-amm.z;
//cross product
test2.x=test2a.y*test2b.z - test2a.z*test2b.y;
test2.y=test2a.z*test2b.x - test2a.x*test2b.z;
test2.z=test2a.x*test2b.y - test2a.y*test2b.x;
x3=test2.x;
y3=test2.y;
z3=test2.z;
testmag=sqrt(pow(test2.x,2) + pow(test2.y,2) + pow(test2.z,2));
test2.x/=testmag;
test2.y/=testmag;
test2.z/=testmag;

test1x2.x=test1.y*test2.z - test1.z*test2.y;
test1x2.y=test1.z*test2.x - test1.x*test2.z;
test1x2.z=test1.x*test2.y - test1.y*test2.x;

testmag=sqrt(pow(test1x2.x,2) + pow(test1x2.y,2) + pow(test1x2.z,2));

```

```

//cross<<N<<" "<<testmag<<endl;

//x4det use
x1=a.x;
y1=a.y;
z1=a.z;
//
x2=am.x;
y2=am.y;
z2=am.z;
//
//
x4=amm.x;
y4=amm.y;
z4=amm.z;
//
a11=x1*x1 + y1*y1 + z1*z1;
a22=x2*x2 + y2*y2 + z2*z2;
a33=x3*x3 + y3*y3 + z3*z3;
a44=x4*x4 + y4*y4 + z4*z4;
//
M11=x4det(x1,y1,z1,1,x2,y2,z2,1,x3,y3,z3,1,x4,y4,z4,1);
M12=x4det(a11,y1,z1,1,a22,y2,z2,1,a33,y3,z3,1,a44,y4,z4,1);
M13=x4det(a11,x1,z1,1,a22,x2,z2,1,a33,x3,z3,1,a44,x4,z4,1);
M14=x4det(a11,x1,y1,1,a22,x2,y2,1,a33,x3,y3,1,a44,x4,y4,1);
M15=x4det(a11,x1,y1,z1,a22,x2,y2,z2,a33,x3,y3,z3,a44,x4,y4,z4);
//
if(fabs(M11)>0)
{
    xnot=M12/M11;
    ynot=-M13/M11;
    znot=M14/M11;
    rnot=xnot*xnot + ynot*ynot + znot*znot -M15/M11;
    rnot=rnot/1000;
}
if(M11==0)
{
    xnot=0;
    ynot=0;
    znot=0;
    rnot=0;
}

cout<<endl<<rnot<<endl;
if(M11!=0)cross<<N<<" "<<testmag<<" "<<rnot<<endl;
}
}
cvShowImage("plot",graphs);
delete(frame);
delete(wireframe);
delete(wireframe_r); //method1
delete(wireframe_r2);
delete(wireframe_r3);
delete(wireframe_t);
delete(wireframe_final);
delete(centroids);
}

```