Development and Testing of Advanced Imaging Methods For Autonomous Robot Control

By Alan Shepherd

Doctoral Thesis

Submitted in partial fulfilment of the requirement for the award of

Doctor of Philosophy of Loughborough University

2023

Loughborough University

Certificate of Originality
Thesis Access Conditions and Deposit Agreement

Students should consult the guidance notes on the electronic thesis deposit and the access conditions in the University's Code of Practice on Research Degree Programmes

Author.....Alan Shepherd.....

Title Development and Testing of Advanced Imaging Methods For Autonomous Robot Control

I [Alan Shepherd. 32 Quantock Road, Weston super Mare, BS23 4DT], "the Depositor",
would like to deposit [Development and Testing of Advanced Imaging Methods For Autonomous Robot Control], hereafter referred to as the "Work", once it has successfully been examined in Loughborough University Research Repository

Status of access OPEN

Moratorium Period.....None.....years, ending...../.....20.....

Status of access approved by (CAPITALS):.....

Supervisor (Signature).....

School of.....

Author's Declaration *I confirm the following :*

CERTIFICATE OF ORIGINALITY
This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work therein has been submitted to this or any other institution for a degree

NON-EXCLUSIVE RIGHTS
The licence rights granted to Loughborough University Research Repository through this agreement are entirely non-exclusive and royalty free. I am free to publish the Work in its present version or future versions elsewhere. I agree that Loughborough University Research Repository administrators or any third party with whom Loughborough University Research Repository has an agreement to do so may, without changing content, convert the Work to any medium or format for the purpose of future preservation and accessibility.

DEPOSIT IN LOUGHBOROUGH UNIVERSITY RESEARCH REPOSITORY
I understand that open access work deposited in Loughborough University Research Repository will be accessible to a wide variety of people and institutions - including automated agents - via the World Wide Web. An electronic copy of my thesis may also be included in the British Library Electronic Theses On-line System (EThOS).
I understand that once the Work is deposited, a citation to the Work will always remain visible. Removal of the Work can be made after discussion with Loughborough University Research Repository, who shall make best efforts to ensure removal of the Work from any third party with whom Loughborough University Research Repository has an agreement. Restricted or Confidential access material will not be available on the World Wide Web until the moratorium period has expired.

- That I am the author of the Work and have the authority to make this agreement and to hereby give Loughborough University Research Repository administrators the right to make available the Work in the way described above.

- That I have exercised reasonable care to ensure that the Work is original, and does not to the best of my knowledge break any UK law or infringe any third party's copyright or other Intellectual Property Right. I have read the University's guidance on third party copyright material in theses.
- The administrators of Loughborough University Research Repository do not hold any obligation to take legal action on behalf of the Depositor, or other rights holders, in the event of breach of Intellectual Property Rights, or any other right, in the material deposited.

*The statement below shall apply to **ALL** copies:*

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Restricted/confidential work: All access and any copying shall be strictly subject to written permission from the University Dean of School and any external sponsor, if any.

Author's signature...A. Shepherd.....Date...8th September 2022.....

user's declaration: for signature during any Moratorium period (Not Open work): *I undertake to uphold the above conditions:*			
Date	Name (CAPITALS)	Signature	Address

Abstract

The availability of low-cost high-performance computing including General Purpose Graphic Processing units (GPGPU) and low-cost Lidar-based depth cameras offer the potential for the development of an inexpensive Lidar-based tracking system for use in autonomous robot control. Such algorithms can also include development of nonparametric 3D computer models of the robot environment. The 3D model would then enable the robot to physically interact with that environment in a way that would not be possible using alternative robot tracking techniques.

This work investigates the practical use and performance of a Lidar-based imaging and tracking algorithm operating at 30Hz and used to create a 3D model of the environment in real-time. The thesis performs a systematic investigation of the practical implementation and performance of such a real-time system using two low-cost test rigs.

These test rigs were developed and funded as part of the project by the author to investigate the practical aspects of implementing such systems on real hardware platforms using off-the-shelf Embedded computing and low-cost Depth-cameras, Radio-Control (RC) servos and sensors. From this work a lot of practical experience has been gained and some original contributions made in development of the tracking algorithms and specifically within the GPGPU codes. The test rigs also enabled the systematic development and performance evaluation of the tracking algorithms and the 3D model build process.

The performance of several documented Iterative Closest Point (ICP)-based tracking algorithms were evaluated for accuracy and performance and a new system developed here, using image features to improve tracking precision was also tested.

The work also shows that Depth-camera based robot tracking is a very complex problem with no easy solutions and many issues relating to tracking performance were found during the development of this project. These are discussed in detail in the concluding sections of the report.

Acknowledgements

I would like to thank Dr David Mulvaney and Dr Vince Dwyer for there long and continued support throughout Covid lockdown and well beyond into their retirement. I would also like to thank Dr Christopher Ward and Dr Peter Hubbard for picking up the official reins following the retirements of David and Vince. I would like to thank Professor Sam Siewert of Embry Riddle Aeronautical University and co-author of [29] for providing me with Linux real-time test code. I would also like to thank the Wolfson School of Mechanical, Electrical and Materials Engineering for their support and the EPSRC for funding this work.

Dedication

This thesis is dedicated to two people. The first is Michael Fountain my former Maintenance Foreman. Mick took a directionless and callow fifteen-year-old school leaver and turned him into a respectable Engineer. Quite simply he started me on my life's journey. The second is Dr Robert Stirling, founder of Stirling Dynamics Limited and my one-time Manager. Meeting Bob has enriched my life in every meaning of those words. In a long and happy career in Engineering there were of course many more whom I could mention, but these two gentlemen had the most profound and memorable effect.

"It is true that we need a root of personal experience from which to grow our understanding. Each new experience plants another root; the smallest root will serve."

Martha Gellhorn

Contents

Notation and Symbols used

There are some inconsistencies found in mathematical symbol usage between the various different texts referenced in this report. The most recent and relevant definitions and alternatives where these are also commonly encountered are given below. Where there is variation in a cited mathematical text the below symbology will be used.

Notation	Description
c	Constants and scalars are lower case
$\boldsymbol{c}$	Vectors are generally lower case bold
C	Matrices are uppercase
$\bar{x}$	Mean of samples from a distribution p(x)
E[X]	Expectation of a random variable X i.e., $E[X] = \sum_{\forall x} p(x).x$ = mean value
P{X=x}	Probability that X=x a given value
P(x|y)	Probability of outcome 'x' given event or condition 'y'
Bayes Rule	$p(b\|a) = \frac{p(b \cap a)}{p(a)} = \frac{p(a\|b).p(b)}{p(a)}$ the role of 'a' and 'b' can be swapped
∀	For all members of the set/distribution
∃	There exists, alternatively: "for some elements of the set"
$a \in B$	'a' is a member or element of set B
$A \wedge B$	And as in A and B, A & B
$A \vee B$	Or as in A or B, A|B is also used
≐	Equal by definition
¬	Logical Not, however '!' and ~ are also frequently used
$C = AB$	C is the Caley product of matrices A and B (they must be conformable)
$c = \boldsymbol{a}.\boldsymbol{b}$	The dot (inner) product of two vectors, a single value
$C = \boldsymbol{a} \times \boldsymbol{b}$	The cross-product of two vectors, a matrix
Σ	Covariance Matrix: $\Sigma = \frac{1}{N-1}\sum_{k=1}^{N}(x_k - \bar{x})(x_k - \bar{x})^T$
Δ	The Mahalanobis distance where: $\Delta^2 = (x - \bar{x})^T \Sigma (x - \bar{x})$
$N(x\|\mu, \sigma^2)$	A normal or Gaussian distribution

Abbreviation and Acronyms

AccX,Y,Z	ServoTrack Carriage Accelerations (g)
AI	Artificial Intelligence
AMP	Asymmetric Multi-Processing (running different tasks on a multi-core computer)
API	Application Programme Interface (typically a library of user interface functions)
Ar	Depth Camera Aperture Ratio (Image Height divided by Width)
ARC	Autonomous Robot Control
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CFS	Completely Fair Scheduling (a scheme used by Linux for multi-task scheduling)
CSV	Comma separated variables a common text-file-format
CSTF	Comparative Tensor Shape Factor
DMA	Direct Memory Access (a real-time communication mechanism)
EKF	Extended Kalman Filter
FPGA	Field Programmable Gated Array
FoV	Field of View
FPS	Frames per Second
FSM	Finite State Machine (aka a Switch-Case block in the C programming language)
FM	Förstner-Moonen Metric
kNN	k-Nearest Neighbours (A search algorithm)
GPS	Global Positioning System (commonly referred to as Sat. Nav.)
GPGPU	General Purpose Graphical Processing Unit
HRT	Hard Real-Time
IA	Industrial Automation
IR	Infra-Red (light sources used in class 1 and 2 laser beams)
ICP	Iterative Closest Point
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
ISR	Interrupt Service Routine (a real-time computer processing mechanism)
Lidar	Light Detection and Ranging
MCAS	Manoeuvring Characteristics Augmentation System
MDP	Markov Decision Process
MEMS	Micro-Electro-Mechanical-Servo or Sensor
ML	Machine Learning
MLP	Multi-Layer Perceptron

MNIST	Modified NIST dataset (a set of hand-written numbers used in ML training)
NDT	Normal Distribution Transform
NFS	Network File System (a Linux data communications pathway)
OS	A computer Operating System
P2P	Point-to-Point Registration
P2L	Point-to-Plane Registration
PCA	Principal Component Analysis (a data-reduction method)
pdf	Probability Density Function
Pitch	ServoTrack Pitch Sensor Angle (deg)
PitchDem	ServoTrack Pitch Demand (deg)
RTOS	Real-Time Operating System
RANSAC	Random Sampling Consensus
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SLAM	Simultaneous Localisation and Mapping (a robot tracking system, often using EKF)
SMP	Symmetric Multi-Processing (a load-balancing OS for multi-core computers)
SRT	Soft Real-Time
SSD	Solid State Drive
SSH	Linux Secure Shell (an encrypted communications protocol, enabling remote logon)
SVD	Singular Value Decomposition
SVM	Support Vector Machines (a data classification method)
TCP/IP	Transmission Control Protocol/Internet Protocol (an ordered sequence of data packets)
TFPos	ServoTrack Time-of-Flight Sensor Position (m)
(T)SDF	(Truncated) Signed Distance Function
Time1-Time6	Real-Time at various location around ServoTrack rig (sec)
TrackPos	ServoTrack Carriage Position (m)
UDP	User Datagram Protocol (a not necessarily ordered sequence of data packets)
VO	Visual Odometry
Voxel	3D Model volume element
VPS	Visual Positioning System
WiFi	Colloquial Name for IEEE 802.11 Wireless Protocol
Yaw	ServoTrack Yaw Sensor Position (degrees)
YawDem	ServoTrack Yaw Demand (degrees)

1. Introduction

1.1 Thesis Aims and Objectives

The primary focus of this work is the development and validation by test of computer algorithms for use in robot tracking and 3D model development utilising Lidar-based depth cameras as the primary input. The methods are used to develop nonparametric 3D models of the environment so that an autonomous robot can navigate in and interact with that environment. The viability and accuracy of the developed algorithms are tested on real hardware and for this purpose test equipment has also been developed within the scope of the project. Shortcomings of the work and some of the issues raised are also discussed.

The motivational idea for this project (the business case) was that of autonomous control of a humanoid or quadruped robot that could assist emergency services by autonomously navigating in a dangerous and unstructured terrain such as a collapsed building, or a building on fire with little in the way of building plans and no prior robot training. The operational paradigm was that the robot would, under the remote direction of the emergency services, autonomously explore the local environment and send back images so that safety officers could rapidly locate and rescue trapped individuals whilst also reducing risk exposure for the rescue workers.

Such a system was considered beyond the time and cost constraints of a project such as this. Nevertheless, it was felt to be a reasonable starting point for a research study offering many possible directions. The literature survey reflects this with an initial wide range of possible research areas.

During this early exploratory phase, computer vision became an interesting aspect of autonomous robot control with a satisfactory level of algorithmic depth and academic rigor. It also tied in well with a desire to become familiar with parallel computing methods such as General-Purpose Graphic Processing Units (GPGPU's) and hence it was considered to be a worthy research topic for this thesis.

The computer vision algorithms developed here are all based on methods garnered from the open literature. The way these methods have been integrated to form a closed solution is considered an original contribution in that the author is not aware of any similar developments, though in today's flood of internet-driven research publications one could not guarantee that such a parallel activity does not exist.

A large part of the contribution for this work is therefore in the details of the implementation and evaluation of these computer imaging methods on practical and affordable hardware.

A key influence in the direction of the work was the seminal paper on Kinect Fusion by Newcombe and others [21]. The primary ideas of using a GPGPU to reduce depth camera data and fuse this into a nonparametric 3D model have been fully adopted here. However, this original work has been extended to include robot tracking in the 3D environment based on the algorithms developed. Key details of implementation are also different to [21] and in particular the method of data reduction employed, based on principal component analysis (PCA), is a very different to the methods used in the Kinect Fusion work.

The project utilises three hardware technologies which form the platform on which the algorithms are developed and tested. Firstly, Embedded computers using the Linux operating system in real-time are used in the hardware test systems. Secondly, GPGPU's provide a massively parallel computing environment ideally suited to the development of vision algorithms. The devices used here are exclusively NVIDIA™ GPGPU's. Thirdly, Intel™ RealSense® Depth Cameras. These devices provide a low-cost lidar-based 3D depth-image frame of a window on the environment.

To ensure a none moving target, the work is restricted to the application of the algorithms developed here using only the hardware technologies discussed above. Though there was also a considerable learning and development exercise relating to the application of real-time Linux on the Embedded computers used as documented in Appendix A.

The computer tracking algorithm development discussed here is based on a large and diverse body of original research in the areas of machine vision, point-cloud-based computer-vision, 3D modelling and robot navigation. The work is best described as an assembly of some of these methods both in terms of algorithm development and hardware application. However, except where clearly stated, all computer algorithms and codes developed here are the original work of the author and are implemented in the C and C++ programming languages and CUDA®, a variant of C developed by NVIDIA™ specifically for their GPGPU hardware products.

Simulation is an often used means of validation in studies such as this. However, a simulation model, that is not validated against real hardware, is little more than an educated guess and frequently found lacking when applied to a real system. Real hardware application also furnishes a unique set of challenges that are difficult if not impossible to predict ahead of actual implementation. Such a process also results in the accumulation of real experience in the development of algorithms and actual hardware performance. The most interesting parts of this experience, not all of which was positive, is also documented here.

As part of the project, two sub-scale and restricted-scope test rigs have been developed. The first test rig known as "ServoTrack" has been built and used in the validation of tracking algorithms developed using independent sensor data to provide a ground truth against which the accuracy of the tracking algorithms

can be measured. A second wheeled robot known as “ServoTruck” has also been developed to further extend validation of the tracking algorithms in the freely moving context of a ground vehicle and as a platform for a further fully autonomous vehicle development test.

The novel contributions of this work include:

1. Development of a real-time hardware test equipment operating at 30Hz and used for the test and performance evaluation of 3 ICP-based robot tracking algorithms.
2. Development of a real-time tracking algorithm utilising the Förstner-Moonen similarity metric to provide geometric feature matching of point-cloud data and thus improved point-pair matching.
3. The Development of a GPU algorithm for parallel point-pair association that also inhibits multiple associations to a single point.
4. The development of a number of PCA-based GPU algorithms for point-to-point and point-to-plane tracking including per-thread matrix operations for computation of Eigen-values and vectors by the Jacobi method.

Chapter 2 of the report contains a literature survey and covers a range of related study fields including visual odometry, tracking algorithms based on Point-to-Point and Point-to-Plane registration methods and use of Covariance matrices and PCA to reduce size of data clouds.

Chapter 3 details the development of the computer tracking algorithms used here. This section builds on the details of the work of others presented in section 2. There are a number of novel developments for GPGPU implementation these are discussed in some detail in this chapter.

Chapter 4 presents details of the development of hardware and software for the two test rigs. These rigs were developed from scratch to carry out the intended tests and involved a considerable amount of additional real-time programming and development effort for the Linux-based embedded computing environments used. This was nevertheless a very useful exercise in learning how to optimise Linux for use in a multi-core and multi-processor, real-time application. Further details on this development are given in Appendix A.

In chapter 5 results of testing of the tracking algorithms developed in chapter 3 are discussed. A key element of this testing has been the validation of robot motion predicted by the tracking algorithms in comparison with sensor data obtained from the ServoTrack test-rig. A number of different algorithms are considered and an assessment made in terms of accuracy and robustness.

Results from the ServoTruck test rig are also discussed here. This radio-controlled test vehicle has more speed and movement freedom to that of ServoTrack but little in the way of sensors to verify tracking

accuracy. It has nevertheless proven to be a useful development tool in algorithm evaluation and development.

Chapter 6 presents conclusions of the findings of this work with an assessment of the algorithms investigated including their shortcomings.

Chapter 7 discusses possible future developments and recommendations for further work. It also discusses how a fully autonomous system based on the algorithms developed could be made a practical reality.

Appendix A gives details of some of the issues found in developing real-time test hardware using Embedded computing technology and particularly real-time Linux.

Appendix B gives details on calibration of the sensors and servos used on the ServoTrack test rig.

1.2 AI versus IA for Real Robot Applications

Artificial Intelligence (AI) was created as a distinct discipline as the result of a Research Workshop held at Dartmouth College, New Hampshire, USA in the summer of 1956 [1]. John McCarthy together with Marvin Minsky formed the MIT Artificial Intelligence Laboratory, later McCarthy went on to create the Stanford Artificial Intelligence lab.

Since that time there have been very many developments in AI and more recently significant improvements in computing capability combined with breakthroughs in back-propagation algorithms and machine learning (ML) for image and pattern recognition has seen a huge surge in real-world application with expansion of these algorithms into many areas.

However, this tsunami in development activity must be tempered with realistic expectations in a practical environment given the need for a robust, competitive application of these methods on real hardware and especially in safety critical applications.

There are still many open questions and serious challenges as to how AI technology can be safely and efficiently applied to autonomous robotics, which has hitherto been a separate and quite independent field of study base on conventional control systems design strategies, state-machines and many other fixed, algorithmic, programming techniques which, for brevity will be referred to as “Industrial Automation” (or IA).

A key issue of concern for safe robotic applications is causality, with IA codes there will generally be a causality path through the code sequences. In most if not all AI/ML systems the causality path is opaque from the outset, in that the output of an ML network for a given training set is an entirely probabilistic

event based on similarity of the events to the training set. The outcome given by the network for any given input is neither tractable nor predictable.

1.2 IA versus AI in Robot Control

Industrial Automation (IA) methods can be made highly reliable, for instance it is common in Aerospace to talk of failure rates of one failure per billion flight hours (or a 1E-9 probability of an event per flight hour). Similar levels of reliability are far from achievable within AI or ML networks, where it is quite common and accepted to have failure rates of the order of 5 in 100 or worse. This may well be acceptable for a simple image or pattern recognition system with low safety impact in the case of a classification failure, but these levels of reliability cannot be appropriate in any area where failure could lead to harm of any kind, physical or financial.

Nothing is perfect and even with high reliability design goals it is not a foregone conclusion that IA methods are always going to be completely safe as two recent incidents with the Boeing 737MAX MCAS design modification have shown. If a given failure scenario was not included in the design requirements, or not correctly dealt with in the failure mode effects analysis, IA methods may not meet their intended safety objectives, occasionally and unfortunately with disastrous consequences. However, though defective, the causality path through such a system design is tractable and can be investigated and subsequently corrected.

This is not the case for AI where with more training the classification of a given scenario can be improved, but this is also quite likely to lead to over-training whereby the network no longer generalises and thus miss-classifies other more common scenarios.

The increase in computing power per unit cost affords ever more complex and powerful data processing techniques to be leveraged into automated control systems. There is however, the requirement to ensure that these systems are cost effective, robust and above all safe. The attitude should be one of caution in the application of appropriate methods for the use-case and in the case of robotic control a concern for safety should be heavily biased in the direction of traditional IA using fixed and tractable algorithms.

This thesis is based on current research in automation, autonomous robotics, machine vision and visual odometry along with other industrial automation technologies. There is of course some cross-overs between the different AI and IA technologies for instance the visual algorithms used include statistical methods (such as least-squares for example) however the algorithms are tractable and based on fixed coding methods and subject to validation testing against sensor data on the developed test rigs. There were no training datasets needed and the code cannot adapt or evolve without the input of the programmer. There are no black boxes within the code.

2. Literature Survey

2.1 Visual Odometry

Reference [2] defines Visual Odometry as: “estimating the motion of a robot…. using visual input alone”. Visual Fusion takes this approach one step further and is the combination of a number of vision and image processing methods used to track the robot and build a “fused” 3D computer model of the robot environment from the data generated by the camera.

This section is a review of a range of different algorithmic methods used to perform visual odometry. That is the estimation of pose and tracking of a moving robot using optical sensors or lidar devices. The same process can also enable the building of a 3D model of that environment using either a parametric (geometric CAD-based) or nonparametric (voxel-occupancy) models. For completeness both optical feature-based, RGB methods and point-cloud-based algorithms are discussed however, the work will focus on point-cloud-based methods obtained from depth camera data.

2.1.1 Visual Feature-Based Odometry

One solution to the odometry problem is to use feature matching between RGB images using any one of a number of feature descriptor methods such as Scale Invariant Feature Transform (SIFT), Speeded Up Robust Features (SURF) [2]. and possibly in combination with depth data to compute the tracking transformation. These images can be collected simultaneously using stereo optical cameras with a fixed pitch or from monocular cameras after some movement. This second method is known as “structure from motion”.

Advantages of using RGB camera data is that they are readily available with generally higher resolution and cheaper than Lidar or depth cameras. The images are more stable and less noisy though they can suffer changes in lighting conditions and intensity. An advantage of using feature descriptors is that they significantly reduce the quantity of data being processed per frame however the processing cost can be high and recovery of detail and geometry of the environment difficult and computationally expensive.

2.2 Depth Cameras and Lidar

Lidar, the use of lasers and time-of-flight or triangulation to compute the distance of objects can also be used to generate point clouds and from multiple clouds movement of the robot can be derived. More recent developments in so-called depth cameras provide an image frame of depth data for the viewed scene.

A key benefit of 3D imaging using Lidar or Depth Cameras, relative to RGB cameras, is that they generate geometric 3D point-cloud datasets as primary output. From this geometric data (under the assumption

that nothing else in the scene is moving) it is possible to infer robot pose and hence motion between subsequent frames by determining the rigid-body transformation between the point-cloud datasets for each frame. This topic is discussed in some detail in the next section.

2.3 Point to Point Registration

A popular approach for dealing with point-cloud datasets is the use of Registration Algorithms developed from original work defined in [3], [5] and elsewhere. The methods are also often referred to as Iterative Closest Point (ICP). The geometric data provided can also additionally be fused to build a nonparametric or geometric model of the robot environment using methods defined in [4].

However robust control of autonomous robots in a rich environment, based solely on visual/sensor information, remains an open problem in machine vision. Some of the confounding factors will also be considered further in this section.

Many papers on this topic have been presented. In [5] Besl and McKay propose a method of registration between two point-pair matched datasets utilising a quaternion formulation. Whereas Arun and others in [6] utilise a Least-Square method based on Singular Valued Decomposition (SVD). Both methods are closely related and in [7] Eggert and others review the performance of four closely related least-squares methods with the general conclusion that they are comparable with no clear winner. Notably this comparison did not consider the very critical point-pair-matching aspect of the registration process. The SVD-based, point to point (P2P) method described by Arun [6] is given below.

The basis of this least-squares, noniterative method is to minimise the L2 error norm between the pair-matched common subset M of the two point-based datasets, the latest reading (sometime known as the model) P and a reference Q thus:

$$E(\mathrm{P},\mathrm{Q}) = \sum_{(P,Q)\in M}\|\mathrm{P}-\mathrm{Q}\|_2 = \sum_{k=1}^{M}\|\mathbf{p_k}-\mathbf{q}_k\|_2 \tag{2.41}$$

The P2P registration problem is to find the rotation matrix T and translation vector $\boldsymbol{x}$ that minimises the L2 error between the two matched datasets, thus:

$$E^*(\mathrm{P},\mathrm{Q}) = \min_{T,\boldsymbol{x}}\left(\sum_{k=1}^{M}\|\mathrm{T}\mathbf{p_k}+\boldsymbol{x}-\mathbf{q_k}\|_2\right) \tag{2.42}$$

The mean values of the two data sets are given by:

$$\boldsymbol{\mu_p} = \frac{1}{M}\sum_{k=1}^{M}\mathbf{p_k} \tag{2.43}$$

And: $$\boldsymbol{\mu}_q = \frac{1}{M}\sum_{k=1}^{M}\mathbf{q_k} \tag{2.44}$$

The key to the method is that if both point clouds P & Q are normalised about their own mean values, the number of variables to solve reduces from 6 to 3 as normalisation removes any offset between the two.

The covariance matrix H is then given by:

$$H = \frac{1}{M-1}\sum_{k=1}^{M}\left(\mathbf{p_k} - \boldsymbol{\mu_p}\right)\left(\mathbf{q_k} - \boldsymbol{\mu_q}\right)^T \tag{2.45}$$

Using SVD to decompose H into $\boldsymbol{u}\Lambda\boldsymbol{v}^T$ the optimal transformation T between the datasets P & Q is then given by:

$$\mathrm{T} = \boldsymbol{v}\boldsymbol{u}^{\boldsymbol{T}} \tag{2.46}$$

The translation vector can then be obtained from:

$$\boldsymbol{x} = \boldsymbol{\mu_q} - \mathrm{T}\boldsymbol{\mu_p} \tag{2.47}$$

One complication is that SVD can return a reflection rather than a rotation, this can be checked for using the determinant of T and corrected, however this failure mode was only seen when other errors were introduced. Another issue is that errors in computing the transformation matrix T will propagate into further errors in displacement $\boldsymbol{x}$.

2.4 Integrating Registration Methods with Robot Tracking

The above P2P algorithm is quite general and in order to implement it (and all other registration methods investigated here) in a robot tracking algorithm requires careful treatment of axes systems. The depth camera sees everything in its own body-axis system, so the changes in rotation and translation computed by P2P based on the point clouds P & Q must first be transformed into the global axis frame taking due care with regard to translations. Figure 2.1 below illustrates the scenario in a plan view of a simple 2D example.

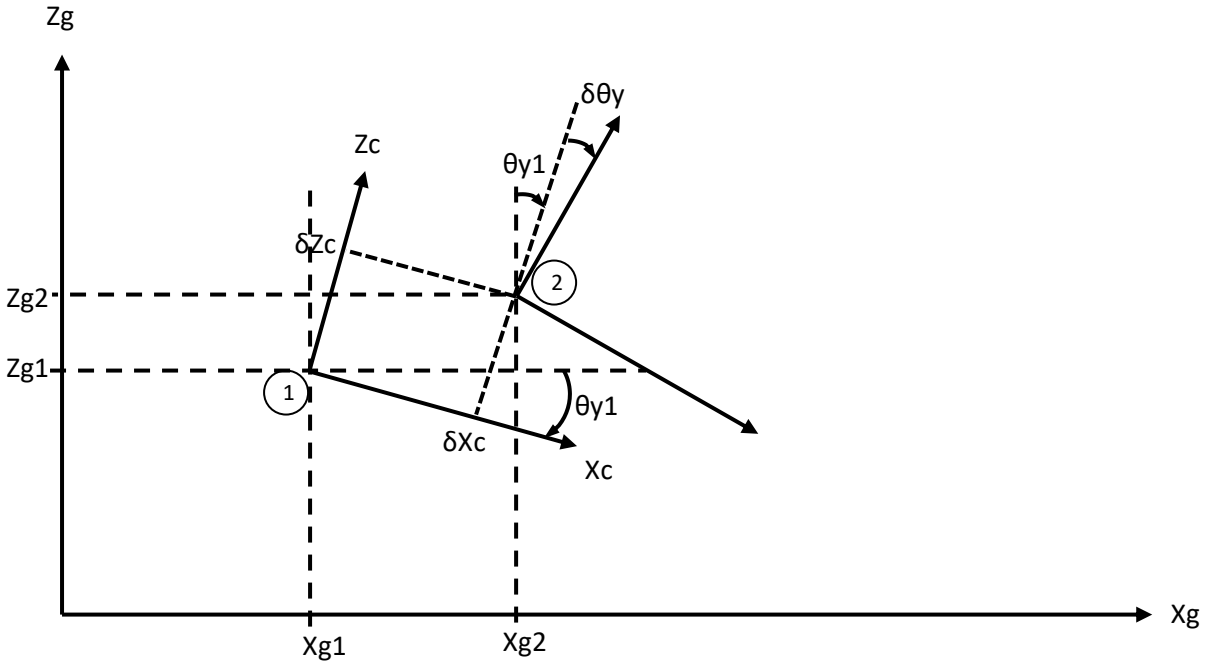


Figure 2.1 Robot Motion Tracking Using P2P Registration Output

In the above scenario the robot has moved from position (1) to position (2) and the P2P algorithm has computed the delta displacement δX and delta rotation matrix δT shown as {δXc, δZc, δθy} in the 2D diagram above. These delta changes are in camera body axes and need to be integrated into the global frame to provide the correct track of the robot. Using the common convention for transformations between the camera frame C and global frame G the transformation matrix T_{CG} is defined thus.

$$\boldsymbol{x_C} = T_{CG}\boldsymbol{x_G} \tag{2.48}$$

which alternately gives:

$$\boldsymbol{x_G} = T_{CG}^{-1}\boldsymbol{x_C} \tag{2.49}$$

The delta position offsets are computed in camera body axes and so must be transformed into global axes before being added to the global position vector $\boldsymbol{x_G}$.

$$\delta\boldsymbol{x_G} = T_{CG}^{-1}\delta\boldsymbol{x_C} \tag{2.50}$$

$$\boldsymbol{x_G} = \boldsymbol{x_G} + \delta\boldsymbol{x_G} \tag{2.51}$$

The delta update δT_{CG} to the transformation matrix T_{CG} can then be applied as a compound product.

$$T_{CG_{t+1}} = T_{CG_t}\delta T_{CG} \tag{2.52}$$

When tested under ideal conditions, with a perfect data-match between frames, the P2P method has been demonstrated to recover the rotation and translation from the point cloud datasets even when the rotation and translational differences are quite large with remarkably high precision. However, under real non-ideal conditions with noisy data there are clearly issues for this method. Reference [13] makes the following point regarding this and other ICP methods:

> *The ICP approach, although breakthrough in its time, presented several possible optimisations and improvements. It assumes that there is a correct correspondence between the points of both clouds which is an assumption that easily fails on real application.*

There are significant issues with the robustness of ICP methods when the datasets contain noise (a feature of all depth sensors tested in chapter 4 of this study). It is also found to be particularly sensitive when the difference between the two depth images is small. This is also a common situation when tracking moving robots since the registration updates are incremental and small. This can lead to rapid build-up of drift in the tracking algorithm.

2.5 Point-to-Plane Registration

A popular alternative to the above P2P method is the point-to-plane (P2L) method defined in [3]. This method is also documented, along with a comprehensive review of other ICP methods and applications, in Pomerleau [8]. This method uses the same pair-matched point cloud datasets and introduces surface normals $\boldsymbol{n}$ of the reference point cloud Q into the objective function thus:

$$E(P,Q) = \sum_{k=1}^{M} \|(T\boldsymbol{p}_k + \boldsymbol{x}) - \boldsymbol{q}_k).\boldsymbol{n}_k\|_2 \tag{2.53}$$

Using the small angle approximation: sin(θ)=θ, cos(θ)=1 and that the product of any two or more sin angles will be negligibly small, yields a skew-symmetric matrix in 3 Euler parameters:

$$T(\alpha,\beta,\gamma) = \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} = [\boldsymbol{r}]_x + I \tag{2.54}$$

where $[\boldsymbol{r}]_x$ is a cross-product operator transforming the vector $\boldsymbol{r}$ to a 3 × 3 skew-symmetric matrix. The full transformation vector $\boldsymbol{\tau}$ is then given by the 6 parameters:

$$\boldsymbol{\tau} = \left[\alpha,\beta,\gamma,t_x t_y t_z\right]^T \tag{2.55}$$

However, this equation needs relating to the physical problem. For example, the right-hand set in Aircraft axes is as shown in figure 2.2 below:

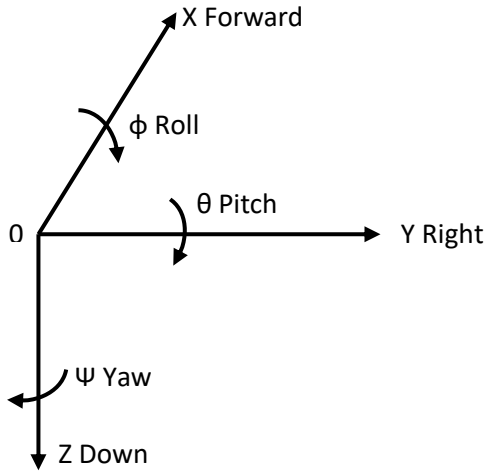


Figure 2.2 Aircraft Conventional Right-Hand Set

The rotation matrix relating body-axis displacements $\boldsymbol{x_a}$ to the inertial-axis system is $\boldsymbol{x_i}$ defined as:

$$\boldsymbol{x_a} = T_{AI}\boldsymbol{x_i} \qquad \text{and conversely } \boldsymbol{x_i} = T_{AI}^{T}\boldsymbol{x_a} \tag{2.56}$$

Rotations are clockwise positive when viewed along each axis from the origin '0' as indicated above. For this system, the Euler transformation matrix T_{AI} is then given by.

$$T_{AI} = \begin{bmatrix} cos\psi cos\theta & sin\psi cos\theta & -sin\theta \\ cos\psi sin\theta sin\phi - cos\phi sin\psi & sin\psi sin\theta sin\phi + cos\psi cos\phi & cos\theta sin\phi \\ cos\psi sin\theta cos\phi + sin\phi sin\psi & sin\psi sin\theta cos\phi - cos\psi sin\phi & cos\theta cos\phi \end{bmatrix} \tag{2.57}$$

The above defined axis convention is commonly used in a range of aircraft and vehicle systems and is very familiar to the author.

The computer graphics axis convention is shown in the following diagram, figure 2.3. Note that the axes are labelled differently. Rotation parameters are suitably renamed, but the rotation axis name labelling is retained.

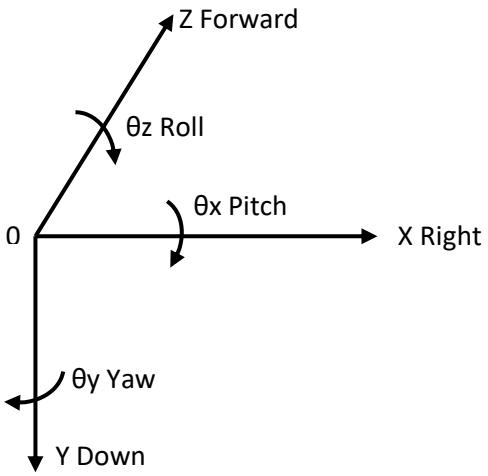


Figure 2.3 Computer Graphics Right-Hand Set

Given the axis renaming some care is needed, the transformation matrix between Camera and Global axis is defined as:

$$\boldsymbol{x_C} = T_{CG}\boldsymbol{x_G} \qquad \text{or} \qquad \boldsymbol{x_G} = T_{CG}^{T}\boldsymbol{x_C} \tag{2.58}$$

The transformation matrix, in terms of the computer graphics vector ordering and angles defined in figure 2.3 above becomes:

$$T_{CG} = \begin{bmatrix} sin\theta_y sin\theta_x sin\theta_z + cos\theta_z cos\theta_y & cos\theta_x sin\theta_z & cos\theta_y sin\theta_x sin\theta_z - cos\theta_z sin\theta_y \\ sin\theta_y sin\theta_x cos\theta_z - cos\theta_x cos\theta_z & cos\theta_x cos\theta_z & cos\theta_y sin\theta_x cos\theta_z + sin\theta_z sin\theta_y \\ sin\theta_y cos\theta_x & -sin\theta_x & cos\theta_y cos\theta_x \end{bmatrix} \tag{2.59}$$

Using the same small angle assumptions gives:

$$T_{cg} \approx \begin{bmatrix} 1 & \theta_z & -\theta_y \\ -\theta_z & 1 & \theta_x \\ \theta_y & -\theta_x & 1 \end{bmatrix} \qquad \text{or} \qquad T_{cg}^{T} \approx \begin{bmatrix} 1 & -\theta_z & \theta_y \\ \theta_z & 1 & -\theta_x \\ -\theta_y & \theta_x & 1 \end{bmatrix} \tag{2.60}$$

The equivalence relations between the angles defined in (2.54) above and those in (2.60) above gives:

$$\begin{aligned} \alpha &= \theta_x \\ \beta &= \theta_y \\ \gamma &= \theta_z \end{aligned} \tag{2.61}$$

Although the Euler matrix has changed in form, it is still a right-hand set and when linearised with the small angle assumptions gives a similar and expected ordering of angles. This is the system convention employed for both small (2.60) and large (2.59) angle transformations in this work.

Returning to the P2L method and using the above assumptions and expanding the objective function we get:

$$E(\mathrm{P},\mathrm{Q}) \approx \sum_{k=1}^{M} \|([\boldsymbol{r}]_x + I)\boldsymbol{p_k} + \boldsymbol{x} - \boldsymbol{q_k}).\boldsymbol{n_k}\|_2$$

$$= \sum_{k=1}^{M} \|(\boldsymbol{r} \times \boldsymbol{p}_k).\boldsymbol{n_k} + \boldsymbol{p_k}.\boldsymbol{n_k} + \boldsymbol{x}.\boldsymbol{n_k} - \boldsymbol{q_k}.\boldsymbol{n_k}\|_2$$

$$= \sum_{k=1}^{M} \|\boldsymbol{r}.(\boldsymbol{p_k} \times \boldsymbol{n_k}) + \boldsymbol{x}.\boldsymbol{n_k} - (\boldsymbol{q_k} - \boldsymbol{p_k}).\boldsymbol{n_k}\|_2$$

$$= \sum_{k=1}^{M} \|\boldsymbol{r}.\boldsymbol{c_k} + \boldsymbol{x}.\boldsymbol{n_k} - \boldsymbol{d_k}.\boldsymbol{n_k}\|_2 \tag{2.62}$$

$$\boldsymbol{c_k} = \boldsymbol{p_k} \times \boldsymbol{n_k}, \;\; \boldsymbol{d_k} = \boldsymbol{q_k} - \boldsymbol{p_k} \tag{2.63}$$

Taking derivatives of the objective function with respect to $\mathbf{r}$ and $\boldsymbol{x}$ we get:

$$\frac{\partial E^2(P,Q)}{\partial r} = \sum_{k=1}^{M} 2\boldsymbol{c_k}(\boldsymbol{r}.\boldsymbol{c_k} + \boldsymbol{x}.\boldsymbol{n_k} - \boldsymbol{d_k}.\boldsymbol{n_k}) = 0 \tag{2.64}$$

$$\frac{\partial E^2(P,Q)}{\partial x} = \sum_{k=1}^{M} 2\boldsymbol{n_k}(\boldsymbol{r}.\boldsymbol{c_k} + \boldsymbol{x}.\boldsymbol{n_k} - \boldsymbol{d_k}.\boldsymbol{n_k}) = 0 \tag{2.65}$$

Rearranging:

$$\sum_{k=1}^{M} \boldsymbol{c_k}(\boldsymbol{r}.\boldsymbol{c_k}) + \boldsymbol{c_k}(\boldsymbol{x}.\boldsymbol{n_k}) = \sum_{k=1}^{M} \boldsymbol{c_k}\,(\boldsymbol{d}_k.\boldsymbol{n_k})$$

$$\sum_{k=1}^{M} \boldsymbol{n_k}(\boldsymbol{r}.\boldsymbol{c_k}) + \boldsymbol{n_k}(\boldsymbol{x}.\boldsymbol{n_k}) = \sum_{k=1}^{M} \boldsymbol{n_k}(\boldsymbol{d}_k.\boldsymbol{n_k})$$

$$\sum_{k=1}^{M} \begin{bmatrix} \boldsymbol{c_k}\boldsymbol{c_k^T}\boldsymbol{r} + \boldsymbol{c_k}\boldsymbol{n_k^T}\boldsymbol{x} \\ \boldsymbol{n_k}\boldsymbol{c_k^T}\boldsymbol{r} + \boldsymbol{n_k}\boldsymbol{n_k^T}\boldsymbol{x} \end{bmatrix} = \sum_{k=1}^{M} \begin{bmatrix} \boldsymbol{c_k}(\boldsymbol{d_k}.\boldsymbol{n_k}) \\ \boldsymbol{n_k}(\boldsymbol{d_k}.\boldsymbol{n_k}) \end{bmatrix} \tag{2.66}$$

Which after further re-arrangement gives:

$$\sum_{k=1}^{M} \begin{bmatrix} \boldsymbol{c_k} \\ \boldsymbol{n_k} \end{bmatrix} \begin{bmatrix} \boldsymbol{c_k^T} & \boldsymbol{n_k^T} \end{bmatrix} \boldsymbol{\tau} = \sum_{k=1}^{M} \begin{bmatrix} \boldsymbol{c_k} \\ \boldsymbol{n_k} \end{bmatrix} (\boldsymbol{d_k}.\boldsymbol{n_k}) \tag{2.67}$$

In simplified form:

$$A\boldsymbol{\tau} = \mathbf{b} \tag{2.68}$$

Where A is a [6x6] matrix, **b** is a [6,1] vector and $\boldsymbol{\tau}$ can be solved using Cholesky decomposition. Clearly a loop k=1...M is required to resolve the matrix A and vector **b** at each iteration step. This is not necessarily an issue for GPU implementation where each thread can compute A and **b** for a corresponding **k** with the partial results coalesced and summed at the end.

An issue for this method may be the small angle assumption, sin(θ) = θ, cos(θ) = 1. This may be a problem for a turning robot where rotation angles violate the assumption leading to errors. It may also contribute to an accumulation of drift. It is also clear that the method requires more computational effort than the P2P method above. This may not be an issue if the method is more effective requiring less iteration, though as already stated P2P does not require iteration in the algorithm proper, though iteration may improve point-pair-matching in both cases as discussed below.

2.6 Point-Pair-Matching

A critical aspect of the point-based registration methods discussed above is that of point-pair matching between the model and the reference datasets. If the data association between points is poor due to sensor noise and/or nonoverlap between data points in the two images the registration loop will be seriously impeded and may result in complete loss of direction in the worst case and unacceptable levels of error leading to tracking drift in other cases.

The object of point-pair-matching being to find the (closest L2 distance) associative mapping between every point in the reference set Q and every point in the latest point set P for at least a common sub-set. An exhaustive search over both datasets requires Nq*Np comparisons to be performed.

Earlier work documented in [8] discusses the formation and use of a dynamic partitioning algorithm "kd-trees" to reduce the serial search burden from O(Np*Nq) to O(Np*log(Nq)) after a O(Nq*log(Np)) build phase. Whilst this algorithm may be efficient for use on a CPU, it is not an efficient algorithm for GPU implementation, where each thread can search in parallel for one match in the reference set Q for each point in the current point set P.

The parallel algorithm used here is based on a GPU implemented, k-nearest neighbours (kNN) algorithm detailed in [9] where, in this application k=1.

However, with the original algorithm detailed in [9] there was no guarantee of a one-to-one correspondence between model and reference pairings when processed in parallel on GPU, thus multiple associations between model and reference point sets could be made. This situation, if not circumvented, will impede accuracy of the registration process. Thus, the algorithm of [9] has been adapted by the author to inhibit the possibility of multiple matches by removing all but the closest match. Implementation details on this mechanism are given in section 3.8.

2.7 Down-Sampling Strategies

A feature of modern Lidar/depth cameras is the generation of a very large quantity of noisy data points. Typical cameras can output more than 300,000 points at an update rate of 30 Hz. Thus point-pair matching on an individual point-by-point basis becomes ineffective even when done in parallel on a high-performance GPU.

There are a number of ways to reduce the problem. One approach is to randomly select points from the cloud and then select the k-nearest neighbours (kNN) and use these clusters as the down-sampled set. There are two problems with this strategy firstly for a very large number of points the random selection process becomes a challenge in itself requiring additional work to ensure that the chosen samples are

reasonably spaced. Secondly further work is then required to perform a kNN search to establish the local groups.

Since the data from depth cameras is regularly ordered in rows pixel-by-pixel over the data frame a much simpler strategy is to down-sample the points based on a regular rectangular grid pattern over the field of view and average the grid contents to obtain the mean vertex for that grid location. Pair-matching is then performed on the reduced and lower density grid datasets.

There are however limits to this method due to the discretisation effect that this grid-based sampling system imposes. Considering a simple two-dimensional analogy of a step change in the sampled surface and two consecutive samples of the same region before and after the camera has moved a small amount. This is represented in the following 2D line sketch, figure 2.4, looked at from above.

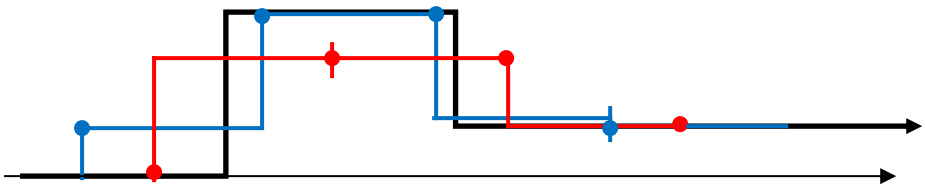


Figure 2.4 Showing Discretisation Effect of Grid-based Down-Sampling

The black line is representative of an actual surface with discrete step changes in height. The blue line represents a down-sampled and averaged surface over a discrete sampling interval. The red line represents a second down-sampled surface taken after the camera has moved a small but unknown amount. As can be seen the surface feature is not faithfully reconstructed by either dataset due to the averaging and discretisation effect.

More problematic however is that the down-sampled surfaces can have very different shapes. The lateral resolution loss is worse than in the depth direction. Lateral resolution is entirely down to the pixel resolution and distance from the camera. This lateral resolution loss impacts registration more severely and will inevitably lead to lateral tracking drift.

Considering the registration matching problem further using another 2D perspective from above to look in more detail at an extreme example shown in figure 2.5 below. In this diagram the blue and red lines show the robot camera horizontal angular field-of-view from two positions. From the original position shown in blue the camera moves forward and to the right indicated by the red lines. Only part of the viewed scene is common to both camera images, it is this “Common Overlap” region that the registration on which the tracking movement of the robot is be computed.

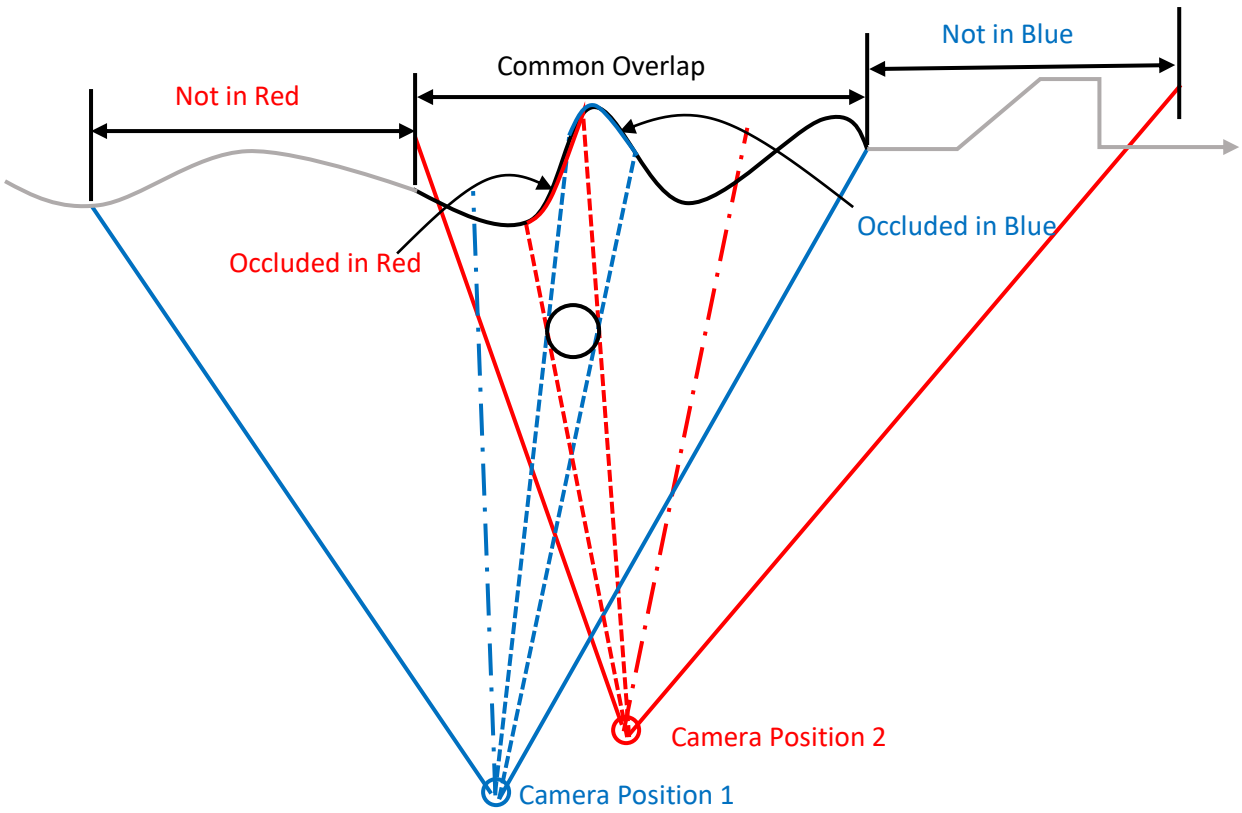


Figure 2.5 Non-Overlap and Occlusion Impact on Registration

The only meaningful region of the image on which registration can be performed is the common overlap indicated above however, this region is also often further broken by regions of occlusion such as caused by the circular object in front of the view. Consequently, there are a large number of data points in both images that cannot be seen (or correctly pair-matched) in the other.

These issues are central to the difficulty of the point-pair-matching problem when used in robot tracking since it is not known a-priori, how much and in what direction the camera has moved, so the boundaries of the common subset are unknown. Points contained in the non-overlapping regions are however still available for pair-miss-matching in both datasets and without further measures to exclude them, quite likely to confound any pair-matching algorithm and result in poor tracking performance leading to drift.

Furthermore, simple selection strategies, such as threshold limiting the L2 norm in the pair-matching process to exclude non-overlapping points, if set too aggressively, can easily lead to no matches at all and a registration failure. Hence some other means of identifying common regions (and a more faithful point correspondence) is required to ensure robust matching in the basic rigid-body registration problem, not to mention the more complex issue of dealing with the situation where other objects are moving in the viewed scene.

To assist in more robust point-pair matching an often-used strategy is to augment the point-cloud vertex data with other information, such as a surface normal corresponding to the local vertex region and/or colour captured from a corresponding RGB-D camera data. These methods require both more care and more effort in the matching process.

2.8 Covariance Matrices and Principal Component Analysis

It has long been recognised by the Computer Vision community [10], [11], [12] & [15] that Covariance matrices are an efficient means of analysing and reducing point-cloud datasets. Furthermore, Principal Component Analysis (PCA) provides an effective means of decomposing this data to provide useful insights when dealing with point clouds. The covariance matrix of a local region of *N* points within a point-cloud dataset is given by:

$$C = \frac{1}{N-1}\sum_{i=1}^{N}(\boldsymbol{x_i} - \overline{\boldsymbol{x}})(\boldsymbol{x_i} - \overline{\boldsymbol{x}})^T \tag{2.69}$$

Where:

$$\overline{\boldsymbol{x}} = \frac{1}{N}\sum_{i=0}^{N}\boldsymbol{x_i} \text{ is the mean of the given set.} \tag{2.70}$$

This [3x3] symmetric matrix is, non-negative-definite and can conveniently be decomposed into real-valued Eigen-values and Eigen-vectors which form an orthonormal set corresponding to the principal components of the dataset from which it is formed. The Eigen-values measure the statistical variance of points $\boldsymbol{x_i}$ along the directions of the corresponding Eigen-vectors. The plane of least variance corresponding to the lowest Eigen-value λ_0 where:

$$\lambda_0 \leq \lambda_1 \leq \lambda_2 \tag{2.71}$$

The associated Eigen-vector $\boldsymbol{v}_0$ therefore provides a least-squares approximation to the surface normal for the selected region. The vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ provide tangent vectors to this normal and if the region is relatively flat, lay within the normal plane.

A convenient measure defined in [10] is that of surface variation where:

$$\sigma_n = \frac{\lambda_0}{\lambda_0+\lambda_1+\lambda_2} \text{ where } \sigma_n = 0 \text{ if all points lay on a flat plane} \tag{2.72}$$

and $\sigma_n = 1/3$ is the maximum value attainable if the distribution of points is completely isotropic, as in a spherical cloud. Note also that $\sigma_n < \epsilon$ can be used as a threshold to assess curvature or "planarity" of the principal plane.

Normal vectors can be used as part of a vertex augmentation in forming point-pair associations for point-cloud registration, however using the covariance matrices a great deal more information can be discerned regarding the shape and structure of the surfaces embedded within the dataset.

There are several ways this information can be grouped and/or segmented for instance using kNN analysis on the PCA-derived surface normals to form related groups to identify for instance common surfaces within the data set. In [11] a PCA decomposition is developed in order to cluster (segment) common regions. This is can be done using an algorithm known as "minimum spanning trees" as defined in [36] and elsewhere.

The similarity or difference between covariance matrices of regions of a point cloud is an obvious basis for performing segmentation of this data into components in a manner similar to that used for photographic images. One method of achieving this is the Mahalanobis distance.

The Mahalanobis distance between a point x and a set of points P with covariance *C* is given by:

$$\Delta^2 = MD = (x - \bar{P})^T C^{-1} (x - \bar{P})$$

This metric could also potentially be used to investigate correspondence between similar regions of two registration images. A new weighting function based on this metric and the Euclidian distance between region centroids would provide a more discriminating and robust pair-matching algorithm from which to compute the registration transformation. In the end this metric was not used as the Forstner-Moonen metric provided a better correspondence between regions, see section 2.9 below.

2.9 The Förstner-Moonen and CSFT Metric

There are other metrics for analysing the similarity (or difference) between covariance matrices and a metric derived to directly compare covariance matrices was developed by Förstner and Moonen [12] working in the area of Photogrammetry. This metric is a direct comparison between covariance matrices (for point-cloud regions *A* and *B*). The Förstner-Moonen Metric (FM) is defined in [12] as:

$$\mathrm{FM} = \sum_{i=1}^{N} log^2[\lambda_i(A, B)] \tag{2.73}$$

Where in [15] it is shown that the joint Eigen-values of $\lambda_i(A, B)$ are the Eigen-values of:

$$\sqrt{A^{-1}} B \sqrt{A^{-1}} \tag{2.74}$$

Any real symmetric, non-negative definite matrix A can be Eigen-decomposed to:

$$A = V \Lambda V^T \qquad \text{Where } \Lambda = diag[\lambda_1, \lambda_2, \ldots \lambda_n] \text{ and } \lambda_i \text{ are Eigen-values.} \tag{2.75}$$

From this it can be shown that:

$$A^{-1} = V\mathrm{R}^{-1}V^T \qquad \text{Where } \mathrm{R}^{-1} = diag[1/\lambda_1, 1/\lambda_2, \dots 1/\lambda_n] \tag{2.76}$$

It is also shown that for such a matrix, a square root matrix also exists and can be formed thus:

$$\sqrt{A^{-1}} = VSV^T \qquad \text{Where } S = diag[\sqrt{1/\lambda_1}, \sqrt{1/\lambda_2}, \dots \sqrt{1/\lambda_n}] \tag{2.77}$$

From [15] the Eigen-values $\lambda_i(A, B)$ can be thus obtained from the Eigen-values of:

$$D = \sqrt{A^{-1}}B\sqrt{A^{-1}} \tag{2.78}$$

They can then be used in the above equation for FM and using this metric, it is then possible to identify similar regions in two registration images for candidate correspondence matching as an additional component of the matching process to more robustly identify correct point-pair associations. For regions that match identically FM=0. As regions diverge in shape the FM metric increases and is thus a measure of geometric dissimilarity. This fact can be used as a means of thresholding during the point-pair matching process so that only points that are generated from regions that match to less than a threshold level on the FM values will be included in the pair-matching process.

It can also be used as a basis for geometrically segmenting components parts of an image, though this is avenue is not followed in this study.

An alternative but very similar methodology is discussed in [13] and [14]. In this case a metric is based on the Eigen-values of comparative covariance regions in two datasets is known as a Comparative Tensor Shape Factor (CTSF), this metric is defined in [14] as:

$$CTSF(P, Q) = \sum_{i=0}^{N}(\lambda_i^A(P) - \lambda_i^B(Q))^2 \tag{2.79}$$

Where A and B denote comparative covariance regions within the two registration datasets P and Q. In [14] the regions A and B are formed using a nearest neighbour search with quite a lot of computational effort required to produce weighted tensors for each pair in the neighbourhood. This is seen as problematic from a real-time implementation. The CSTF metric requires the Eigen-values of the corresponding covariance matrices in A and B. However, the FM metric is easier to compute for a comparison between segmented image frames and is therefore used in preference.

2.10 Normal Distribution Transform and Related Methods

A statistically-based alternative to ICP is a method developed by Magnusson in [16] and called Normal Distribution Transform (NDT). A comparison of performance with ICP is also given in [17]. The application being surveying of mines.

The basic approach is to divide the space into 3D cells in the region with cubes of 0.5m side-lengths. For each cell the mean vertex of the point-cloud within is computed:

$$\mu_p = \frac{1}{N}\sum_{k=1}^{N} p_k \tag{2.80}$$

The covariance matrix for the cell is also computed as:

$$C = \frac{1}{N-1}\sum_{k=1}^{N}(p_k - \mu_p)\,(p_k - \mu_p)^T \tag{2.81}$$

The probability that there is a point at position x in the cell can then be modelled by a normal distribution with pdf given by:

$$f(x) = \frac{1}{c}\exp\left(-\frac{(x-\mu_p)^T C^{-1}(x-\mu_p)}{2}\right) \tag{2.82}$$

Given a set of points x, pose $p = \{\theta_x, \theta_y, \theta_z\}$, and a transformation $T(\mathrm{p}, x_k)$ an objective function is defined as:

$$s(\mathrm{p}) = -\sum_{k=1}^{N} f(T(\mathrm{p}, x_k)) \tag{2.83}$$

Newton's iteration can then be formed thus:

$$\Delta p = -H(s)^{-1} g(s) \qquad \text{Where H(s) and g(s) are the Hessian and gradient w.r.t } s \tag{2.84}$$

The equations for H(s) and g(s) are complex, the full equations and algorithm are given in [17]. In [18] a comparison between this algorithm and point-to-point ICP is made.

Unlike the discrete nature of ICP point matching, NDT gives a piecewise smooth representation of the model point cloud, with continuous first and second order derivatives. However, as the authors point out there are also issues:

> *"NDT has an inherent limitation in the fact that space is subdivided into regular cells. The discretisation artefacts that come from the subdivision process, leading to discontinuities in the surface representations at cell edges, can sometimes be problematic."*

For this reason, the authors develop an enhanced method known as trilinear NDT. The authors go on to compare the three methods and of particular interest is their comparison of the so called "valley of

convergence" where the range of rotational and translational offsets between the different methods is explored. When the methods are compared on this basis, the results are comparable with a minor improvement for standard NDT. From a performance standpoint the basic NDT method appears to be much faster, but although the enhancement known as tri-linear NDT is much slower than ICP though it shows marked improvements over the basic NDT method.

It is important to bear in mind that there are two objectives in this exercise. The primary objective is one of accurately registering robot pose from frame to frame in order to provide an accurate robot global track. The second objective is to construct a 3D model of the global scene for subsequent robot interaction and/or 3D global mapping. Separation of these objectives potentially frees up workload needed during the registration phase allowing for potential process speed optimisation.

The reduced data needed for the registration phase need not necessarily be useable in the 3D model construction phase. This can be accurately constructed using data fusion of the full point-cloud dataset in a global 3D model once an accurate pose is known. Furthermore, it is also then possible to perform loop closure (tracking/pose correction) based on the current frame and the contents of the integrated 3D model at the current pose position once the 3D model is consolidated especially after the robot has performed a closed circuit in the environment.

2.11 Simultaneous Location and Mapping (SLAM)

A large corpus of work has been carried out on this topic in relation to robot position tracking and mapping of its environment, there is thus a great deal in common with the present work. The SLAM problem has received much attention in the literature, an overview of which is given in [20] is summarised as follows:

> *"a moving vehicle attempting to recover a spatial map of its environment, while simultaneously estimating its own pose relative to the map…. This approach uses an extended Kalman filter (EKF) for estimating the posterior distribution over the map and the robot pose."*

A leading proponent of the field is Sebastian Thrun who in [19] with others produced a classic text on the topic. The central conjecture of this probabilistic approach defined in [19] is that:

> *"A robot that carries a notion of its own uncertainty and that acts accordingly is superior to one that does not."*

This is not disputed, though the work also then goes on to state:

> *"However, these advantages come at a price. Traditionally, the two most frequently cited limitations of probabilistic algorithms are computational inefficiency, and a need to*

approximate. Probabilistic algorithms are inherently less efficient than nonprobabilistic ones, due to the fact that they consider entire probability densities. The need to approximate arises from the fact that most robot worlds are continuous. Computing exact posterior distributions is typically infeasible, since distributions over the continuum possess infinitely many dimensions. Sometimes, one is fortunate in that the uncertainty can approximated tightly with a compact parametric model (e.g., discrete distributions or Gaussians); in other cases, such approximations are too crude and more complicated representations most be employed."

This is the main reason that this approach has not been further adopted here, for even with the power of a GPGPU the volume of data to be processed from the Lidar camera tracking in real time at 30Hz was considered already challenge enough and the quality of the 3D model developed by the fusion process testament enough to the accuracy of the tracking processes used.

Whilst there is undisputed benefit in using these methods on a more capable and developed system, for example, to fuse other sensor data such as odometry and accelerometer/orientation devices, with the lidar-computed tracking data, providing an improved model of vehicle motion.

However, owing to time, cost and complexity, the hardware and algorithms developed here are constrained and solely reliant upon the lidar camera data for both robot tracking and mapping of the 3D environment.

2.12 Other Registration Methods

There has been a wealth of other and different attempts at solving the point-cloud registration problem. There are however a number of significant trends in more recent work and some examples of these are given below

2.12.1 RANSAC

Random Sample Consensus is a method of sampling by randomly choosing a small set of observations from a much larger population. Assessment of the fitness of a given sample is generally obtained by building a consensus model for the problem (or against a pre-existing model) and then rejecting outliers which do not fit. It is not the most efficient method of model fitting but since it makes no prior assumptions is a useful method when little is known as to the nature of the data. It works because the probability of hitting a good solution by randomly choosing a small sub-set is quite large. The method is independent of population size but is to some extent dependent on the size of the outlier population, the breakdown point being 50% since beyond that, outliers become inliers.

There have been a number of image registration algorithms based on this method, though most of these are based on feature alignment in optical images. Reference [21] investigates RANSAC applied to point-

cloud data. This method is doubly interesting since it also utilises the methods of [12] discussed in section 2.9 above to identify common features in both registration images. The method performs multiple registrations using these features and then performing RANSAC on them to derive a consensus transformation and rotation. This approach, in common with the SLAM approach discussed above will have a high computational overhead which is seen as too great to realistically achieve 30Hz real-time tracking. There are secondary doubts that a consensus based on two sets of noisy data would achieve anything better than the methods being applied, though this is not proven. The method as deployed in [21] was applied to the orientation of good quality "toy" datasets, which is definitely not the case here.

2.12.2 Semantic Segmentation

Semantic Segmentation is a method of identifying separate components and their boundaries typically within RGB camera images and often using machine learning. The method has not been widely used in relation to point cloud data sets however in [22] this is considered. Unsupervised segmentation of the point cloud does make sense in that as with the above RANSAC example it can be used to identify common regions in both images and use these as a basis for registration, eliminating spurious associations between data points and thereby improving the matching process. However full semantic segmentation utilising a machine learning algorithm with multi-layer perceptron's and convolution. This method is therefore unlikely to meet the real-time requirements of this project. It is also quite likely to be faced with scenarios for which it has not been adequately trained and thus safety and reliability of such a system would also be highly questionable.

2.13 Summarizing Point-Cloud Registration Research

A lot of effort has been expended and many papers written on this difficult (and as yet unsolved) problem. The earliest ICP algorithms are all based on least-squares methods and provide the best possible theoretical match when the datasets presented are perfect. However, real-world data is noisy and there is no exact point-to-point correspondence between points collected from two different depth-camera frames and under these conditions the methods are seen to be prone to imperfect matching errors that consequentially lead to long-term tracking errors, more commonly known as drift.

The NDT-based methods attempt to use point cloud statistical models in place of points themselves but the implementation is still rather mechanical and vulnerable to discretisation artifacts created by breaking the data into discrete voxels much larger than the ones used here, though as discussed above this is a problem for any grid-based reduction.

The computer vision-derived and photogrammetric methods of [10] and [12] seem to offer a better way to develop a richer geometric description of the point cloud data including segmentation of regions within each frame and registration matching by mapping common features in both images.

The key to this is obtaining good matching of features in both images and with the possible future application of RANSAC, the elimination of everything else including components that move due to other bodies, pets, plants and curtains blowing in the breeze. GPUs are capable of far more concurrent real-time processing then CPUs and with future developments can be leveraged to address these issues in a way that was not until recent times possible.

2.14 Visual Fusion

The springboard for the visual-fusion approach used here originates in the work of Newcombe and others in [23]. This paper presents a number of connected ideas including the use of a Microsoft™ Kinect® depth camera and a GPGPU for depth image fusion to produce a 3D nonparametric model of the environment and also to track motion of the (robot) camera.

One of the key ideas of [23] is fusion of point cloud data into a 3D voxel model. The data in this model can then be used by a robot to autonomously interact with that environment via the 3D representation in the developing model.

It could also be used with the methods of [10] and [11] to develop 3D parametric models of the environment and in a reverse engineering process, produce geometric CAD models. This would also significantly reduce the memory needed to hold such a model.

It is also the case that without modern parallel processing computing technologies such as the GPGPU these methods would not be practical especially in a real-time application such as considered here. Hence to some extent the theory is both enabled and biased towards these technologies. Detailed GPGPU implementation of some of the following algorithms are given in more detail in Chapter 3.

Central to this process is the development of the 3D model of the robot environment. There are several methods of nonparametric 3D volumetric model representation used within the computer vision field, these are discussed below.

2.14.1 Volume Occupancy Models

In the simplest of these methods, the 3D world is modelled as a discretised volume of (generally cubic) 3D elements known as voxels. Voxels are to 3D volume models what pixels are to a 2D computer image. The volume model is represented by a 3D matrix of voxels and each voxel element is either occupied, in which

case it is assigned a nonzero value (which can represent a probability or a colour) or it is unoccupied free space simply identified as having a zero value.

This is known as an occupancy model and is the simplest method of modelling a 3D world in a nonparametric way. This world is easily visualised, pixel rays from a computer screen penetrate into the 3D model until they hit a voxel surface (or not). The surface of that voxel can then be rendered (visualised on the computer screen) in a colour appropriate to the application or left blank if nothing was present along that ray. Populating the model is also simply a question of allocating values appropriately along a projection ray. The method is fast and perfectly suited to parallel operation on a GPU where each pixel ray is chased out in its own thread in parallel with every other pixel.

2.14.2 SDF and TSDF Volume Models

The second type of model is slightly more complex and known as a Signed Distance Function or SDF. This methodology, introduced by Curless and Levoy in [4] utilises the same basic voxel model as the occupancy model above, but the method of allocating voxel values is different. To illustrate the SDF volume modelling method a simple parametric sphere example is used, as shown in figure 2.6 below.

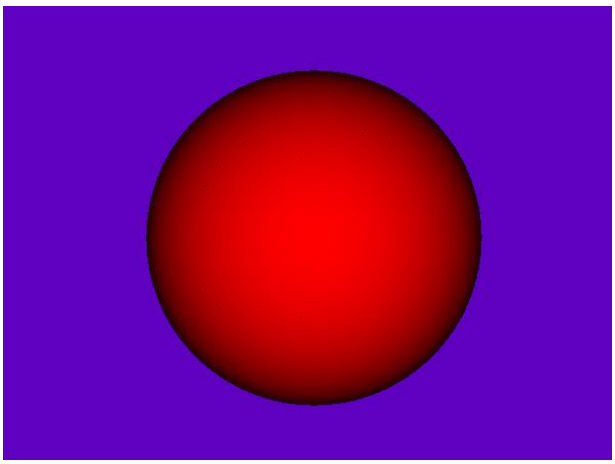
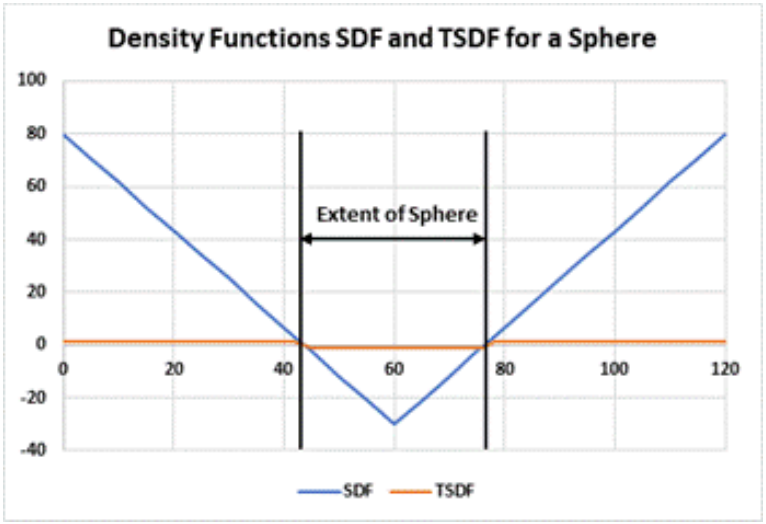


Figure 2.6 Simple 3D Sphere Modelled Using SDF

The SDF volume model has the following properties:

- Everywhere outside the sphere surface is positive and decreasing towards it
- Everywhere inside the surface is negative and increasing towards the nearest surface
- The surface itself is represented by the zero-crossing iso-surface

This is illustrated on the following figure 2.7, where a transect is taken along a diagonal through the 3D volume in the figure 2.6 above from top-near-left through the sphere to bottom-far-right in the 3D Model.



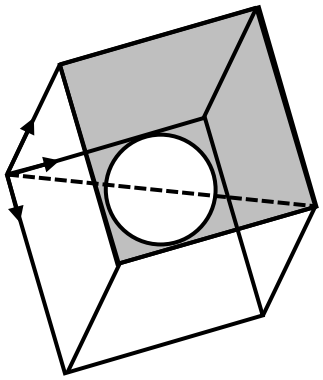


Figure 2.7 SDF and TSDF Functions Plotted for a Sphere

The value of the SDF model is that from any voxel location, the distance to the iso-surface is directly computable so the effort needed to find the surface voxel is minimal. Rendering of the local surface can be interpolated over the nearest neighbouring 3D voxels cell values and thus presented as a smooth image from a relatively crude voxel model as shown with the sphere on figure 2.6 above. In this case surface normals are also computed to provide shading indicative of model depth. However, this shading process is far easier with parametric models than nonparametric models derived from noisy depth sensor data.

There are issues with the SDF representation. Firstly, the scalar fields of multiple objects in the same SDF volume model interact and this destroys correct image rendering. The solution to this problem is the Truncated SDF (TSDF) as shown by the orange line on the figure 2.7 above. In this case the SDF values are truncated to +1 above the surface and -1 below the surface, with zero at the surface as before. This overcomes the multiple object interference problem, but at the loss of the ability to compute directly to the voxel location of the surface. The surface in this case must be located by iteration over the voxel elements along the viewing ray, the same method as used for the much simpler occupancy model and the reason that the simpler occupancy model is used in this work.

2.15 Depth from 3D Model Volume

One of the problems of using registration methods to track a robot over a long period is residual buildup of drift within the global tracking position.

One method of counteracting this is to compare the latest tracking update from the current and previous camera frame with a delta-update based on the current camera frame and one synthesizes from the 3D Model. This provides a corrective feedback mechanism sometimes known as a “loop-closure” in robot tracking studies.

A scheme for this is obtained by forward projecting from the current camera position into the 3D Model and using this information to synthesis a virtual camera depth frame and then to use this information to perform a corrective registration with the current camera pose.

The envisaged steps are as follows:

1. An ICP registration process is carried out between the reference frame ‘R’ and latest depth camera frame ‘P’.
2. This delta-movement is then used to give a revised estimate of the robot global position in the 3D model.
3. From this estimate a ray-trace from each camera pixel into the 3D model is made over the field-of-view of the virtual camera at that estimated pose.
4. This ray-trace generates a new and independent virtual depth image ‘V’.
5. A further ICP registration “correction” step is now performed between ‘P’ and ‘V’.
6. This correction step is then used to update the estimate of global position for the robot.
7. The cloud data from ‘P’ is then fused into the 3D model at the updated global position.

This correction process will fail if the image in the 3D Model is not yet sufficiently well developed. This may happen when the robot is moving or rotating rapidly so that the scene viewed in ‘P’ does not provide a good overlap with the image recovered in ‘V’. The pair-matching process between the data recovered from the 3D model in ‘V’ and the frame ‘P’ is also prone to point-pair-miss-match errors so this “correction” may not be as effective as would be desired. Finally, the 3D model itself is built with residual errors, however small they may be, and so does not represent a ground truth.

3. Robot Tracking Algorithm Development

3.1 Introduction

Since in large part the contribution of this work centres on the implementation, incremental development and testing of algorithms from earlier published works, it is important to express their development in an efficient and readable manner. In discussing the work, a combination of conventional mathematics (which tell nothing of actual implementation), flow charts (which can be quite space inefficient) and code fragments are used. The latter option is seen as the most practical and concise and hence used most often here. To aid the reader, each code fragment is contained within a bounding box and is headlined with CPU or GPU according to the platform on which it is designed to run and the name of the routine from which it is extracted. All codes were written in C and C++ (for the CPU) and CUDA® for the GPU.

To aid reading key **parameters** and **functions** referred to in the report commentary and the code fragments are indicated in **bold**. The code fragments also contain additional comments indicated by the double slash "//" commonly used for comments in C++ programming.

3.2 Tracking Algorithm Sign Conventions

The axis system used here, in common with other computer graphics work, is a right-hand set as illustrated in the following sketch. As with all right hand sets the angles are clockwise-positive when viewed from the origin along the axis of rotation. Figure 3.1 below shows a typical graphic volume presentation as viewed on a computer screen; the same conventions are used for the depth camera and 3D model.

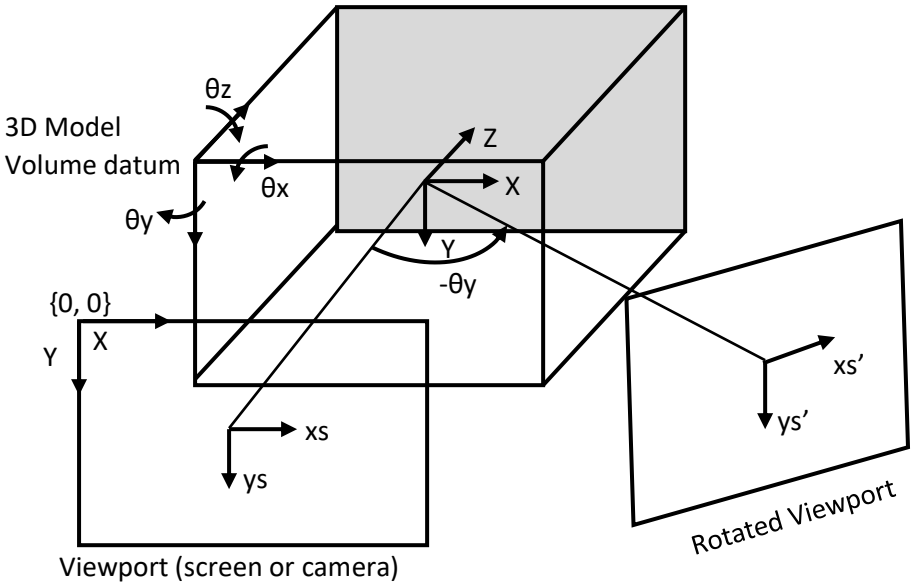


Figure 3.1 Axis System Convention Used

The 6-DoF camera global pose is captured by the (4x4) homogeneous rigid-body transformation matrix:

$$\mathrm{T}_{CG}(\theta, \mathrm{x}) = \begin{bmatrix} R & \mathrm{X} \\ 0^T & 1 \end{bmatrix} \tag{3.1}$$

Where R represents the (3x3) Euler rotation matrix defined in section 2.5 and X the (3x1) translation components. The transformation maps vertex data in the camera frame X_c into the global frame X_g by:

$$X_g = \mathrm{T}_{CG} X_c \qquad \text{Where } X_c = (x_c, y_c, z_c, 1)^T \tag{3.2}$$

Camera depth data is provided as a pixel-array of depth values d(u, v) where u and v represent the camera pixel coordinates in horizontal and vertical directions respectively (with (0,0) being the upper left corner of the camera image) and depth d is in camera depth units.

This depth pixel array can be converted to 3D vertex data in the camera axis using the following projection equations:

$$\left.\begin{array}{ll} z_c = K_d d & \text{Where } \mathrm{K_d} \text{ converts camera units to metres} \\ x_c = z_c \left(u - \frac{C_x}{2}\right) \frac{1}{K_c} & C_x, C_y \text{ are the camera pixel dimensions} \\ y_c = z_c (v - \frac{C_y}{2}) \frac{1}{K_c} & k_c \text{ is the isotropic camera projection constant} \end{array}\right\} \tag{3.3}$$

3.3 Robot Tracking Algorithm Overview

In this section the algorithms used to develop robot tracking and control are discussed. The work presented here provides both a means of tracking the robot in 6 degrees of freedom and also accurately constructing a 3D model of the environment in which the robot can navigate.

The diagram shown on figure 3.2 below illustrates the basic scheme. Though please note that different tracking algorithms have been developed and tested and only the visual tracking code, within the dotted outline, has been developed in this report.

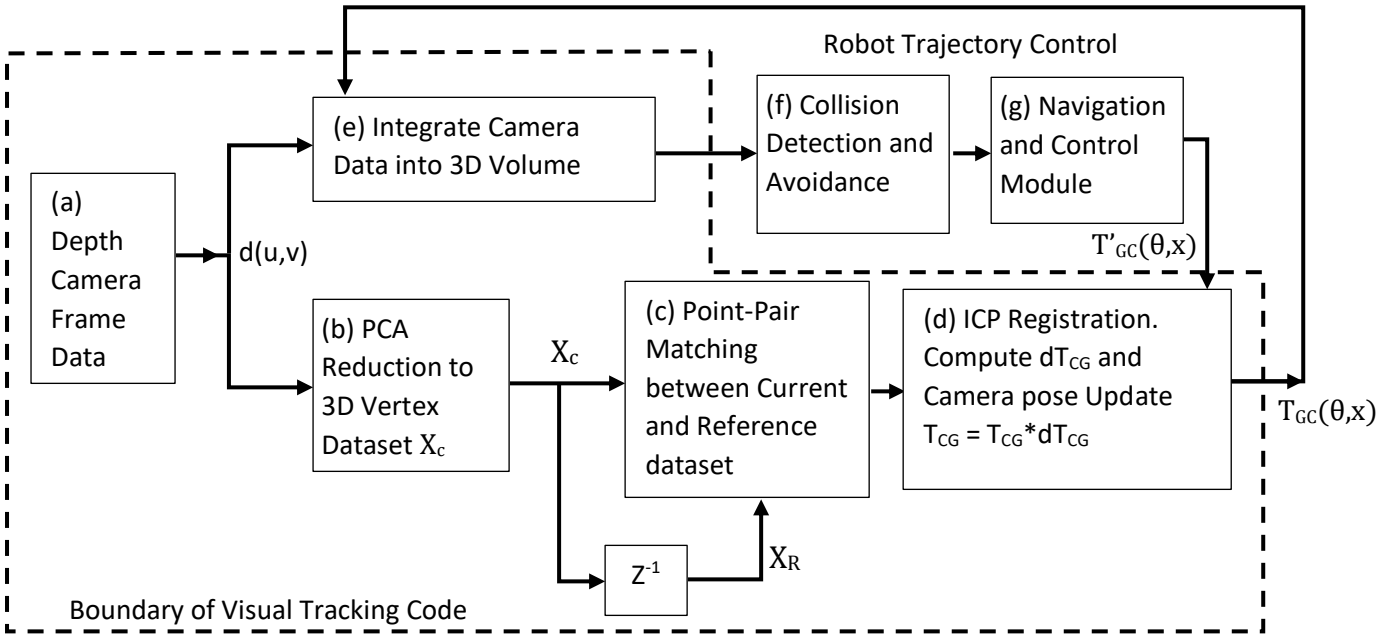


Figure 3.2 Robot Tracking Algorithm Schematic

The schematic in figure 3.2 above identifies the final target architecture for a robot control scheme and is discussed as follows.

Central to autonomous control of a mobile robot is the ability to navigate without collisions in the local environment. In this process the robot will use a depth camera. Data from this camera at (a) follows two paths. At (b) the data is decomposed into a set of 3D vertices and a point-pair match at (c) between this vertex set X_C and the reference frame X_R is used as input to a registration process at (d).

This process (d) computes the delta transformation dT_{CG} between the current vertex set X_c and the reference vertex set X_R. The revised global transformation between camera and global frame $T_{GC}(\theta,x)$ is then updated. In situations where the registration process fails due to the camera's view being blocked, this module will revert to using the robot state information provided in $T'_{GC}(\theta,x)$. However, if the robot cannot safely recover its position in the 3D model it will be forced to stop and the 3D model reset.

The updated pose transformation is then used to integrate/fuse the current unreduced camera vertex data into the global 3D volume model at (e). This is a simple voxel occupancy model with all cells zero until populated from camera vertex data. The volume model at (e), in conjunction with the state and geometry data for the robot is used in (f) to provide steering input to avoid collisions with the environment based on the equations given in [34] appropriately modified for a mobile robot. Navigation and control updates computed in (g) will be used to guide the robot to its final objective.

The Navigation and Control module (g) maintains robot states $T'_{GC}(\theta,x)$, and also responds to additional sensors, teleoperation and collision avoidance inputs to steer a course that meets the objectives without causing damage to itself or the environment. In the following section the theory and practice of the depth camera visual fusion process including registration is discussed in detail.

3.4 Bilateral Depth Data Pre-Filtering

In an effort to reduce depth camera sensor noise pre-filtering can be employed. The bilateral filtering technique used was originally developed for processing of 2D camera images. The following figure 3.3 shows a typical sub-section of an unfiltered depth frame taken using the D435i depth camera.

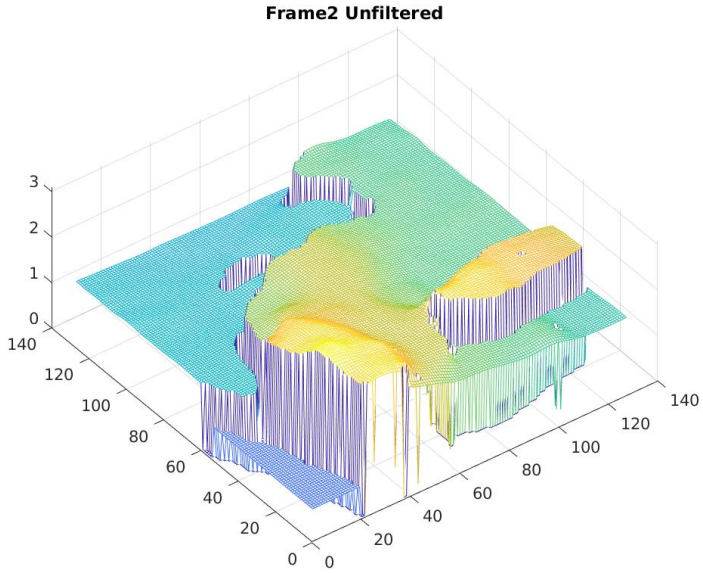


Figure 3.3 Unfiltered Depth Data Take using D435i Camera

Since registration is based on matching subsequent frames, it is instructive to investigate the delta difference between two subsequent stationary frames as is shown on figure 3.4 below.

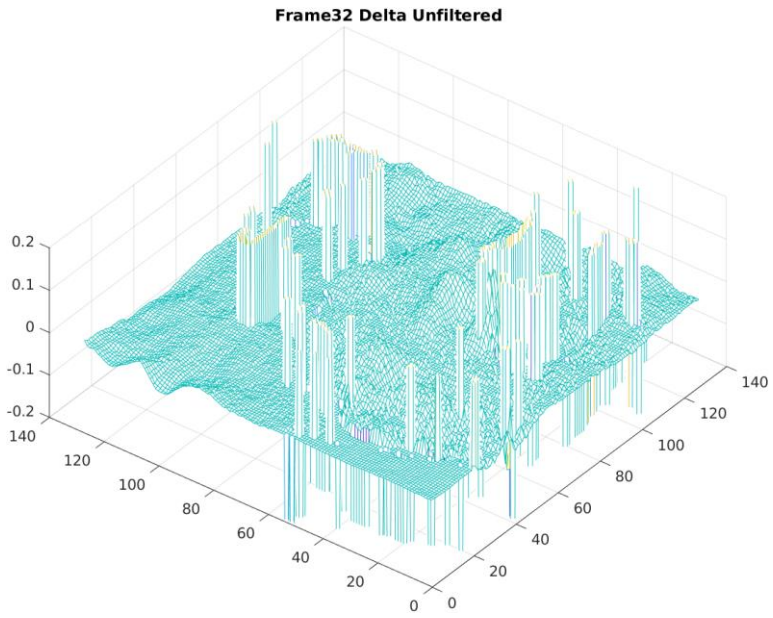


Figure 3.4 Depth Frame Delta for D435i Camera

There are two levels of noise shown in the delta-plot on figure 3.4 above. There are many large positive and negative spikes, these are associated with sharp edge discontinuities between image features and drop-outs at these edges. These edge noise components are unstable from frame to frame and hence lead to the large spikes seen on the above plot. Note also that these spikes are cropped in the plot to allow low level noise to be rendered visible.

The second type of noise has the form of an almost continuous waviness within a range of +/-0.05m the coherence of this waviness is surprising; this noise is however consistent with detailed analysis of depth camera mean and standard deviation measured in camera testing (discussed in chapter 4 below).

Both the depth camera and image field were stationary during this image acquisition process and the ambient lighting conditions favourable (low light in a room with minimal daylight and curtains drawn) so both types of noise are believed to be entirely due to the depth camera capture mechanics and its onboard acquisition process firmware.

Bilateral Filtering is a popular technique used in processing colour and monochrome camera images as it can smooth out blemishes whilst at the same time preserving edges. The following figure 3.5 shows the action of this filter when applied to the two separate frames and the delta between them and compares directly with the unfiltered image in figure 3.4 above.

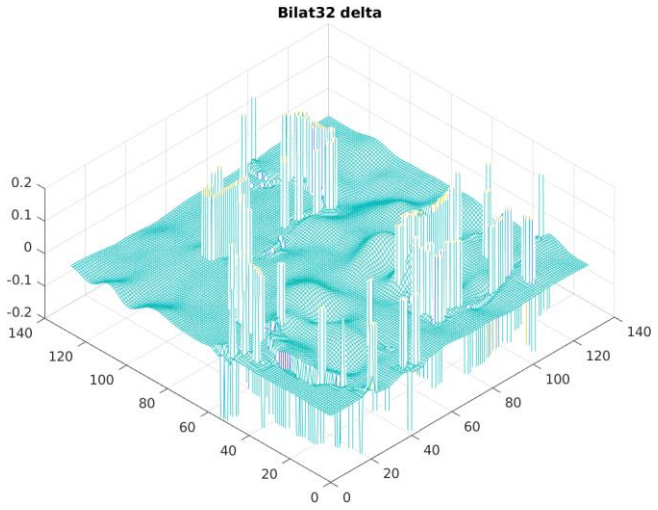


Figure 3.5 Delta Depth Frame after application of a Bilateral Filter

It is notable that this type of filtering has had very little effect on the sharp edge spikes, perhaps not surprising given it is an edge-preserving filter. It has also made the low-level noise smoother and more coherent. It is also notable that the Frobenius norm (F) for the filtered (F=33.66) and unfiltered delta (F=33.85) is hardly changed for the chosen filter parameter settings. Furthermore, an extensive search of filter parameter settings showed that it was not possible to reduce the delta-frame F-norm without significantly eroding image features.

Thus, it is concluded that whilst this filter has a quite useful smoothing effect on image features it is not effective at removing the frame-to-frame edge spikes. Therefore, in order to deal with these troublesome spikes another method is required. The method actually employed here is based on the cellular reduction approach documented in the next section. Whereby any cells containing dropouts (leading to these edge spikes) is simply dropped from further analysis. Whilst this may seem quite a crude process leading to many cells being removed it is seen to be very effective, though it does also put some constraints on cell size. If the cells are large in relation to the image size the strategy can also remove too much useful data.

3.5 GPU Programming

Since one of the key enablers for this work was the use of a GPU to process depth camera data in parallel, the following sections inevitably discuss programming GPU kernels (the parallel execution functions) in some detail. A basic overview of the GPU processing methodology used is given here. To illustrate this with a typical example, the bilateral filter kernel discussed in section 3.4 is used. The code components are encapsulated in boxes with **bold** text used to highlight key variables in the following descriptions.

The float data type is used extensively in GPU kernel code, even when integers are being processed. Much use is also made of extended data types such as float2 and int3 etc. These are simple 2 or 3-element vector structures with an extensive range of supporting function macros using these types.

All GPU functions are "launched" from the CPU with a call to a CPU-based launcher function. The following 3 lines of figure 3.6 show a typical CPU call sequence for the bilateral filter. The line numbers do not exist in the code, they are added for the descriptions following the figures.

```
1    cudaMemcpy(b_Depth, b_Filt, CAM_BYTES, cudaMemcpyHostToDevice);
2    bilateralFilterLauncher(b_Depth, c_Depth, arrFullSz, bilat_ksz, ssinv2, sdinv2);
3    cudaMemcpy(c_Out, c_Depth, CAM_BYTES,cudaMemcpyDeviceToHost);
```

Figure 3.6 Main Program Bilateral Filter Launch Code Sequence

Line	Description
1	Copies **CAM_BYTES** of camera data from the CPU (Host) to the GPU (Device) Memory. In this case **b_Filt** is copied into **b_Depth**.
2	This CPU function call launches the GPU to filter the camera array in **b_Depth** to produce the filtered result in **c_Depth** in GPU memory. Other call arguments are: arrFullSz is an int2 struct of the camera pixel dimensions (640x480 in this case) bilat_ksz, is the size of the bilateral filter element ssinv2 and ssdinv2 are bilateral filter parameters
3	Copies the processed **c_Depth** back from GPU (device) memory to the CPU (host) array **c_Out.** This operation is seldom necessary unless the data is to be further processed in the CPU.

There are two components to the GPU processing sequence. The launcher specifies how the data is to be shared out and processed by the parallel GPU threads and a GPU kernel function that will be called in parallel for all threads in the launch to actually process the data. Firstly, the internals of the CPU Launch function.

```
void bilateralFilterLauncher (UINT16* src, UINT16* dst, int2 arrSz, int bilat_ksz,
                                                                    float ssinv2, float sdinv2) {
4       dim3 block (16, 16);
5       dim3 grid (divUp (arrSz.x, block.x), divUp (arrSz.y, block.y));
6       bilateral_kernel<<<grid, block>>>(src, dst, arrSz, bilat_ksz, ssinv2, sdinv2);
7       cudaDeviceSynchronize();
}
```

Figure 3.7 Bilateral Filter CPU Launcher Function

Line	Description
4	Defines the **block** dimension for this GPU call, the (16,16) dimensions divide neatly into 640 and 480.
5	Defines the **grid** dimensions for the GPU call. The **divUp** function simply rounds up integer division to ensures that there are enough GPU threads allocated to complete the call and will cause an excess of threads to be launched if the division would leave a fractional result. Any redundant excess threads will be trapped in the kernel function and return immediately. In this case the grid dimension is (640/16, 480/16) = (40, 30) a total of 40x30x(16x16) parallel threads = (640x480), 307200 threads are needed.
6	Is where the kernel threads are kicked off in parallel on the GPU, the <<<**grid**, **block**>>> dimensions being used by the GPU to allocate the threads across the GPU cores. It is important to note there is no further sequencing of the parallel threads and their execution order is completely random and can be different from one call to the next.
7	**cudaDeviceSyncronize**() this call will not return control to the CPU until all GPU threads have processed to completion. For well-behaved code it is a good idea to call this CPU function after every kernel launch so that each GPU launch step is completed atomically before the next can proceed.

Secondly, the GPU kernel function is called in parallel for each launched thread. The full internal details of the filter are omitted from figure 3.8 below as they are not important to this discussion, the way in which the data is accessed by individual threads is key.

```
__global__ void bilateral_kernel(UINT16* src, UINT16* dst, int2 arrSz, const int bilat_ksz,
                                                    const float ssinv2, const float sdinv2) {
//
// This function performs a bilateral filter on src data and puts it in dst, both sized by arrSz
//
8       int x = threadIdx.x + blockIdx.x * blockDim.x;
9       int y = threadIdx.y + blockIdx.y * blockDim.y;
10      int cols = arrSz.x;
11      int rows = arrSz.y;
12      if (x >= cols || y >= rows) return;
13      const int i2d = x + y * cols;
14      float value = float(src[i2d]);

// … process code for this filter…

15      dst[i2d] = (UINT16)(result);        // set the result (for this thread) and return(void)

}
```

Figure 3.8 Bilateral Filter Kernel Function Internals

Line	Description
8&9	Each thread is called with a unique combination of **threadIdx.x** & **.y** and **blockIdx.x** & **.y** indices with row and column variables corresponding to its location in the mapping schema set up in the above launcher. This thread-specific information is passed to the parallel kernels by the organising layer of the GPU launcher at launch time. Thus, each pixel of the original camera data is allocated to a specific and unique thread and the 2D nature of the data is conserved in the, **x** & **y** indices.
10&11	The pixel array dimensions are made explicit in **rows** and **cols** for ease of reading/coding.
Line 12	This application is set to launch 307200 threads, however the GPU has only 2304 cores, so this will require the cores to be scheduled to process the data in batches. This batch processing happens within the GPU scheduler management. The call will require 134 batches of 2304 cores to fully process the data resulting in an excess of 2304*134 – 640x480 = 1536 core/thread launches and they will be launched, so this line of code is critical in killing them before they cause memory violation errors or GPU hangs!
13,14&15	The camera pixel data is stored in row-major order. The storage arrays **src** & **dst** are simple linear memory arrays so **i2d** provides a linear index into the row and col of these arrays. It is also noteworthy that the **x** & **y** indices are meaningful in relation to the 2D geometry of the arrays. This filter processes array elements immediately surrounding the **x, y** index for this pixel/thread, so conservation of this geometric relationship is essential to correct filter operation.

Figure 3.9 gives a summary of the steps for a typical GPU kernel launch from within CPU code:

```
Copy data to be processed from CPU to GPU
Call the CPU Launch function->
        Compute the grid and block dimensions
        Call the GPU <<<grid, block>>>kernel_function()->
                Set the internal thread parameters from threadIdx and blockIdx supplied
                Perform the parallel function code
                <-return(void)
        cudaDeviceSynchronize()        // wait for all threads in this kernel to complete
        <-Return from the Launcher
copy data from GPU to CPU (if strictly necessary)
```

Figure 3.9 Summary of a Typical CPU-GPU Launch Process

In the following sections all kernel launches follow the same basic process steps outlined above.

As defined in [24] through [26] there are very many ways to program GPU's using the pre-defined CUDA interfaces. The strategy used here was to obtain the simplest sub-set of interfaces capable of providing an efficient, workable route to achieving the objectives and by proceeding thus, obtain robust and verifiable GPU code.

For example, all memory accessed in GPU kernels is stored in GPU-global memory. The code therefore relies on the GPU memory management system to coalesce and pre-fetch data on a per-thread basis. This approach may not be the most efficient, but is far simpler than managing GPU-local memory access in all its various flavours.

Only a single CPU thread is used to launch all GPU kernels in strict sequence. Multiple CPU threads and GPU-Streams are not used for concurrent GPU processing. The case for and complexity of concurrent-parallel processing was not seen as justified for the work undertaken.

Kernel atomic functions (such as atomicAdd) are used sparingly where needed as they ensure that the operation will be carried out by that thread without interference from other threads. These are limited-scope operations are rapid and thread collisions at this level are relatively benign. There are no critical sections in any of the GPU kernel code to avoid thread-level bottlenecks.

The following sections discuss the main features of the code developed as a series of code fragments relating to the main algorithms given in sections 3.10 and figure 3.20 section 3.11 and figure 3.21. The full code listings are quite lengthy and would require significantly more space than is given here, they are available from [27] for anyone interested to follow up on the actual code.

3.6 PCA Depth Image Reduction

The camera produces a (640x480) pixel image for each frame taken at 30 frames per second. To reduce this data to a manageable rate a grid-based, cellular, data-reduction technique using PCA as discussed in section 2.8 above, is adopted here and applied as a pre-process to all the algorithms developed. This method provides a means of data reduction that is less sensitive to noise and provides a smaller, richer description of the viewed scene.

Each depth camera image is subdivided into a regular grid of CellSz segments. The actual CellSz used is a key parameter in all the methods tested and is set in a data file prior to program launch, the currently allowed CellSz.x = CellSz.y values are: 2, 4, 5, 8, 10, 16, 32.

This allocation is controlled within the PCA_Reduce_Launcher code to ensure that grid and block dimensions selected are consistent with the CellSz selected and depth array size. Figure 3.10 below shows part of the code for the specific case CellSz.x = 5 along with the kernel launch call.

```
//   CPU - PCA_Reduce_Launcher
   int2 Bl;
   int2 Gr;
   switch(cellSz.x)
   {
….
      case 5:
         Bl.x=16;Bl.y=16;
         Gr.x=8; Gr.y=6;
         break;
….
   }
   dim3 block ( Bl.x, Bl.y );
   dim3 grid ( Gr.x, Gr.y );

   PCA_Reduce_kernel<<<grid, block>>>(d_Depth, arrSz, cellSz,… other args);
   cudaDeviceSynchronize();
}
```

Figure 3.10 Part of PCA_Reduce_Launcher Showing grid and block sizing for CellSz.x=5

Each cell is then processed in parallel by the GPU with one thread per cell to produce a covariance matrix for that cell. Figure 3.11 below gives an overview of the initial vertex averaging and cell removal process.

```
// GPU – PCA_Reduce_kernel(float* d_Depth, int2 cellSz, int* dMap,
                                .....float* pointVec  …… float* dWorkMat…int* dAtomVec)
// not all arguments are shown
// d_Depth     the depth camera array
// cellSz      the PCA cell size
// dMap        a map index array into dWorkMat used for this thread
// pointVec    PCA processed vertex, normal and other data for this thread
// dWorkMat a GPU global array pre-allocated to hold internal workspace for all threads
// dAtomVec  an array containing atomic values, used for data-sequencing and diagnostics
//
   const int Cellx = threadIdx.x + blockIdx.x * blockDim.x;
   const int Celly = threadIdx.y + blockIdx.y * blockDim.y;
   const int colsd = arrSz.x;

// trap and log overshoots (should be zero)
   if( ((Cellx+1)*cellSz.x>arrSz.x) || ((Celly+1)*cellSz.y>arrSz.y) ){
      atomicAdd(&dAtomVec[5], 1);      // should not happen
      return;
   }
const float total = float(cellSz.x*cellSz.y);  // total pixels in this cell

// the start index for this cell in d_Depth[]
   const int startd = Cellx*cellSz.x + Celly*colsd*cellSz.y;     // in d_Depth
   const int mapIdx = Get2DIndex(Cellx, Celly, mapSz.x);      //2D map location of this cell
   dMap[mapIdx] = -1;         // -1 indicates a rejected cell in dMap

for(x=0;x<cellSz.x;x++){       // Reject any cell which contains zero elements
     for(y=0;y<cellSz.y;y++){       // perform the vertex mean summation along the way
          ix = Cellx*cellSz.x + x;
          iy = Celly*cellSz.y + y;
          in2d = startd + x + y*colsd;            // indexing into the camera depth array
          Zp = d_Depth[in2d];
          if(Zp<CAM_DMIN ) {                      // if Zp is less than the camera minimum
              atomicAdd(&dAtomVec[2], 1);  // bump the count of null cells
              return;                             // and leave
          }
          Zm += Zp;
          Xm += float(ix - CAM_CX)*Zp*CAM_Kinv;      // project from camera to 3D
          Ym += float(iy - CAM_CY)*Zp*CAM_Kinv;      // vertex and sum
     }
}
// compute the mean vertex for this cell
Xm = Xm/total;
Ym = Ym/total;
Zm = Zm/total;
```

Figure 3.11 PCA Reduction Part 1 – Initial Vertex Averaging and Cell Rejection

In the above kernel fragment each thread loops over one specific cell located by thread parameters **Cellx** and **Celly** in the data array, **ix** and **iy** are the camera pixel coordinates within this cell and **in2d** is the linear index into the depth array. This part gets the depth **Zp** and computes the means Zm, Xm & Ym over all pixels in this cell using the camera projection equations defined in equation 3.3 where:

$$\text{CAM\_Kinv} = \frac{1}{K_c}, \text{CAM\_CX} = \frac{C_x}{2}, etc.$$

dMap is a 2D array of the same size as the cell array used in this data reduction, **mapIdx** is the index into this data map for the **CellX** and **CellY** of this thread. This conserves the 2D spatial layout of the camera data for use later in the algorithm.

Part 2 of the PCA reduction algorithm is given in figure 3.12 below.

```
// GPU – PCA_Reduce_kernel -  Part 2
// Build the S matrix S(3,total) from the cell image patch
   for(y=0;y<cellSz.y;y++){
      for(x=0;x<cellSz.x;x++){
         in2d = startd + x + y*colsd;
         const int ix = CellX*cellSz.x + x;
         const int iy = CellY*cellSz.y + y;
         const int inS =  x + y*cellSz.x;
//
         Zp = d_Depth[in2d];
         Xp = float(ix - CAM_CX)*Zp*CAM_Kinv;
         Yp = float(iy - CAM_CY)*Zp*CAM_Kinv;
         S(0,inS) = Xp - Xm;
         S(1,inS) = Yp - Ym;
         S(2,inS) = Zp - Zm;
      }
   }
// Compute the Covariance Matrix C = S*S'/(total-1)
   for (i=0;i<3;i++){
      for (j=0;j<3;j++){
         C(i,j) = 0.0;
         for(k=0;k<total;k++){
            C(i,j) = C(i,j) + S(i,k)*S.Trans(k,j);
         }
         C(i,j) = C(i,j)/float(total-1);
      }
   }

// Perform Jacobi iteration on C to get Eigen-values and vectors V for this cell
     Jacobi_GMat(C, V, W, R);          // W & R are internally-used work matrices
     for(i=0;i<3;i++){
          Lambda[i] = C(i,i);      // Eigen-values are returned on the lead diagonal in C
     }
     ShellSort_GPU(Lambda, Idx3, 3);    // sort the E-values into ascending order
// use Idx3 to swap the E-vectors to match E-value-order
     for(i=0;i<3;i++){
          for(j=0;j<3;j++){
               W(i,j) = V(i,Idx3[j]);
          }
     }
```

Figure 3.12 PCA Reduction Part 2 – The Reduction Process

The first part computes the S matrix used to form the covariance matrix *C* for this cell where:

$$C = \frac{1}{total-1} S * S^T$$

The functions used to perform Eigen-decomposition **Jacobi_GMat** and **ShellSort_GPU** are standard textbook algorithms taken from [32] and [33] respectively. All mathematical matrices, as distinct from depth image matrices, are indexed in standard column-major ordering.

The thread-local matrices and vectors: C, V, W, R, S & Lambda are all mapped onto (and stored in) GPU global memory. Details of this mapping mechanism are discussed in section 3.11 below.

Each thread that completes with a valid cell, produces a vector entry in the output matrix **pointVec**. **AtomicAdd** ensures each vector is given a unique location within **pointVec** and since threads are randomly launched, **myIndex** is also stored in the **dMap** to conserve the relationship between the PCA reduced points in pointVec and their location of origin in the 2D schema. The details are given in figure 3.9 below:

```
// GPU – PCA_Reduce_Kernel – Part 3
   const int myIndex = atomicAdd(&dAtomVec[0], 1);    // unique output index for this cell
   dMap[mapIdx] = myIndex;                            // store the index in dMap

// save the mean vertex and normal for this cell
   pointVec[myIndex                ] = Xm;      // mean X
   pointVec[myIndex +    pntSz.x] = Ym;      // mean Y
   pointVec[myIndex + 2*pntSz.x] = Zm;      // mean Z
   pointVec[myIndex + 3*pntSz.x] = Vj.x;    // nx
   pointVec[myIndex + 4*pntSz.x] = Vj.y;    // ny
   pointVec[myIndex + 5*pntSz.x] = Vj.z;    // nz
   pointVec[myIndex + 6*pntSz.x] = startW; // starting point in dWorkMat for this cell
   pointVec[myIndex + 7*pntSz.x] = det;     // determinant of covariance for this cell
   pointVec[myIndex + PVEC_ELEMENT*pntSz.x] = 0.0;     // fTest used in point-pair matching
```

Figure 3.13 PCA Reduction Part 3 – The Main Outputs

Note other outputs are computed in PCA_Reduce and used in the CellSearch algorithm these are discussed later in section 3.12 and figure 3.26.

3.7 PCA Surface Reconstruction

In order to verify that the PCA reduction provides an accurate approximation to the viewed scene it is helpful to reconstruct the original depth image from the PCA reduced data contained in **pointVec** and **dMap** shown in figure 3.13 above.

The depth camera image capture process is effectively an inverse projection of the 3D scene into a 2D pixel-image frame as illustrated in figure 3.14 below.

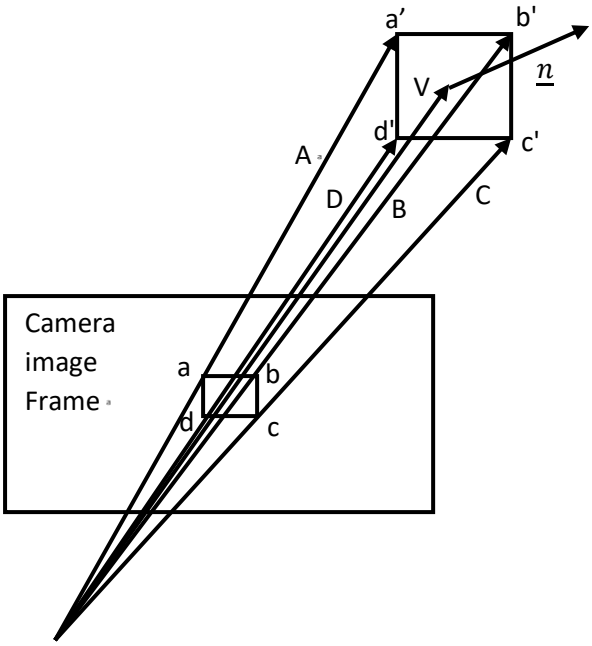


Figure 3.14 Mapping of 3D data into the Camera Image Frame

After PCA reduction the data contained in pointVec refers to the real-world surface position vector V and surface normal $\underline{n}$ for the image cell in question. dMap locates the cell within the camera image grid from which {V, $\underline{n}$} were generated. The ordinals {a, b, c, d} represent the pixel boundaries for the cell within the camera image frame. These ordinals also provide the projection boundaries of the real-world surface {a', b', c', d'} which are the points where the vectors {A, B, C, D} terminate at the 3D surface.

Since all points {a', b', c', d'} are coplanar to the PCA reduced surface then any vector (A-V) will be perpendicular to the surface normal $\underline{n}$ and by the vector dot product rule.

$$(A - V).\underline{n} = 0 \tag{3.4}$$

$$A.\underline{n} = V.\underline{n} \tag{3.5}$$

Expanding the vector components:

$$(A_x i + A_y j + A_z k).\left(n_x i + n_y j + n_z k\right) = n_x V_x + n_y V_y + n_z V_z \tag{3.6}$$

For each cell, the location of the cell image pixel coordinates is known in the ordinals {a, b, c, d}.

Thus, for point $a,\ u = a_x,\ v = a_y$, substituting these in the projection equations (3.3) and setting $K_{inv} = \frac{1}{K_c}$ gives:

$$A_x = Z_a \left(a_x - \frac{Cx}{2}\right) K_{inv} \tag{3.7}$$

$$A_y = Z_a \left(a_y - \frac{Cy}{2}\right) K_{inv} \tag{3.8}$$

$$A_z = Z_a \tag{3.9}$$

Rearranging 3.6 to 3.9 in terms of depth Z_a gives

$$Z_a = \frac{n_x v_x + n_y v_y + n_z v_z}{n_x\left(a_x - \frac{Cx}{2}\right)K_{inv} + n_y(a_y - \frac{Cy}{2})K_{inv} + n_z} \tag{3.10}$$

Proceeding in a similar way Z_b, Z_c & Z_d can also be obtained. These are the ordinals of the vertex of each corner of the projected surface. Noting that this surface will be oriented in 2 directions relative to the camera and that cellSz.x and cellSz.y are the cell sizes in the x & y directions, the gradients are then defined by:

$$M_{ab} = \frac{Z_b - Z_a}{cellSz.x} \tag{3.11}$$

$$M_{ad} = \frac{Z_d - Z_a}{cellSz.y} \tag{3.12}$$

The cell interior points Z_p can then be populated using:

$$Z_p = M_{ab} * x + M_{ad} * y + Z_a \tag{3.13}$$

Where x and y iterate over pixels in each given cell starting from the top left ordinate.

This provides a basis for reconstruction of the 3D image from the PCA reduced data contained in **pointVec** and **dMap**. This process is used in chapter 5, section 5.2 to validate the PCA reduction method used.

3.8 Point-Pair Matching Algorithm

Accurate point-pair matching is critical to all the algorithms tested. The point-pair matching problem is close enough to a related GPU algorithm know as k-Nearest Neighbours (or kNN) that it can be used in this role. Point-pair matching can be considered as the kNN problem with k=1. The algorithm developed here is based on work and code documented in [9] and has been further adapted by the author for the specific purpose of point-pair matching on GPU. The process is to find the closest correspondence between vector points in the latest surface (stored in **pointVec**) and a previous reference surface (stored in **refVec**). To achieve this a **dMatch** index vector is generated. This contains the indices in **pointVec** that most closely matches the points contained in **refVec**. Thus:

pointVec[**dMatch**[Idx]] ~ **refVec**[Idx] Where ~ means closest pair-match

The pair-match algorithm first selects the closest point as shown in figure 3.15 below.

```
// GPU point_pair_match_2 – part 1

// Get the refVec point to compare in this thread
   const int refIdx = threadIdx.x + blockIdx.x * blockDim.x;
   const int Points=6;
   for(i=0;i<Points;i++){
      Xr[i] = refVec[refIdx + i*pntSz.x];
   }
   float savMin = 1.0E+3;          // a very large initial value
   int idxMin = -1;                // assume no match found
   dMatch[refIdx] = idxMin;        // initialise dMatch failed

// test this refVec point against all points in pointVec
   for (int pntIdx=0; pntIdx<pointCount; pntIdx++) {
      for(i=0;i<Points;i++){
         Xp[i] = pointVec[pntIdx + i*pntSz.x];
         Dx[i] = Xp[i] - Xr[i];
      }
      float TestD = norm3df( Dx[0], Dx[1], Dx[2] );  // the L2 delta norm

// selects the closest match
      if(TestD<savMin){
         savMin=TestD;
         idxMin=pntIdx;
      }
   }
```

Figure 3.15 Point-Pair Matching Algorithm Part 1

This first part of the algorithm searches for the closest L2 match between the current threads reference vector **refVec**[**refIdx**…] over all elements of the latest **pointVec**. The L2 match includes the first 6

elements of the vector, the vertex and the surface normal. The output at this point will be **savMin** (the closest L2 match) and the **idxMin** of this match in **pointVec**. If this **SavMin** passes the **DisMin** threshold a further test of vector angular alignment is performed as shown in figure 3.16 below.

```
// GPU point_pair_match_2 – part 2

   if(savMin < DistMin){ // matching L2 distance threshold test
      float Sxr=0.0;
      float Sxp=0.0;
      float Arp=0.0;
      for(i=0;i<Points;i++){
         Xp[i] = pointVec[idxMin + i*pntSz.x];
         Arp += Xr[i]*Xp[i];
         Sxr += Xr[i]*Xr[i];
         Sxp += Xp[i]*Xp[i];
      }
      Sxr = sqrtf(Sxr);
      Sxp = sqrtf(Sxp);
      const float cosRP = fabsf(Arp)/(Sxr*Sxp);
      if(cosRP > CosMin){            // cosine angle matching threshold

//         See point_pair_match_2 – Part 3 below
      } else{
         idxMin = -idxMin;          // failed on CosMin
      }
   } else {
      idxMin = -idxMin;             // failed on DistMin
   }
   dMatch[refIdx]=idxMin;
}
```

Figure 3.16 Point-Pair Matching Algorithm Part 2

If **SavMin** is less than the threshold **DistMin** value, further testing is carried out, otherwise the algorithm returns with this dMatch[**refIdx**] = **idxMin** = **-idxMin** indicating no match found for this element in **refVec**.

If **SavMin** qualifies, the cosine of the angle between the two vectors **cosRP** is then evaluated and tested against the cosine threshold **CosMin.**

If this also qualifies Part 3 of the algorithm is then evaluated see figure 3.17 below.

Because the GPU algorithm addresses the problem in parallel, then for every point in **refVec** a GPU thread simultaneously searches for its closest companion in **pointVec.** There is then the possibility that multiple points in **refVec** will be matched to the same point in **pointVec** and this needs to be inhibited as it will degrade accuracy of the registration process. Ideally for iterative closest point (ICP) algorithms, there should be <u>one and only one</u> point-pair-correspondence between any single point in the two vectors. The

following code fragment in figure 3.17 below is designed to prevent multiple matches and is only active once a candidate match for the current **refIdx** has been found (indicated by **cosRP**>**CosMin** in part 2 above).

```
// GPU – point_pair_match_2 – Part 3
if(cosRP>CosMin){                              // a match candidate has been found
   float* fTest = &pointVec[idxMin + PVEC_ELEMENT*pntSz.x]; // get any previous value
   int* prevIdx = &ipVec[idxMin];              // get the previous match index
   if(savMin < fTest[0]){                      // is this a closer match?
      atomicExch(fTest, savMin);               // if so, swap fTest to savMin
      if(prevIdx[0] > -1){                     // if there was a previous match
        dMatch[prevIdx[0]]=-1;                 // break it
        atomicAdd(&dAtomVec[4],1);             // count the number of match breaks
        atomicAdd(&dAtomVec[3],-1);            // decrement the match count
      }
      atomicExch(prevIdx, refIdx);             // reset prevIdx to this refIdx
      dMatch[refIdx] = idxMin;                 // this is the new match
      atomicAdd(dAtomVec[3],1)                 // increment the match count
   } else {
      idxMin=-idxMin;                          // failed to match closer than
   }                                           // previous match held in fTest
}
```

Figure 3.17 Point-Pair Match Part 3 - Showing Association Limiting

Prior to the part 3 of point_pair_match_2, two key pointers **fTest** and **prevIdx** are obtained from data stored in **pointVec** and **ipVec**. These two components are both initialised prior to the launch of point_pair_match_2. The **fTest** value in **pointVec** is initialised to 1.0 and all **ipVec** elements are set to -1.

When a candidate match is found it is tested against any previous **fTest** in **pointVec**. If this new match **savMin** is closer, it becomes the **fTest** for this point. If a previous match existed **(prevIdx>-1)** it is broken and the new match saved: atomicExch**(prevIdx, refIdx)**. This process works in parallel so the **atomicAdd** and **atomicExch** (atomic add and exchange) functions ensure that these operations cannot be concurrently overwritten by other threads. For most cases, where there is no previous match at this **refIdx, fTest**=**savMin** and **prevIdx**=**refIdx**. However, when there is a previous association, it is broken and the new match is set. **dAtomVec[4]** is a diagnostic and counts the number of occasions where multiple associations were attempted. **dAtomVec[3]** counts the actual number of pair-matches made.

It is important to note however that the above code inhibits, but does not completely prevent, multiple mappings. To achieve complete prevention would require a critical section over this entire section of code. Critical sections are bad practice within GPU code as they prevent parallel execution resulting in major performance degradation due to thread bottlenecks caused by the critical sections. Note that the inner part of the above algorithm only executes when multiple pairings are detected and this situation is

sufficiently rare that the algorithm, though not perfect, has been validated by test and proven to be both efficient (little impact on performance) and effective in removing multiple pairings.

3.9 Computing Means

There are various methods of performing serial operations such as summing vectors in parallel using GPU's. The method used here is to fragment the vectors into a number (**blockDim.x)** of **SegSzX** sub-vectors, compute the partial sums of these sub-vectors into a **WorkVec** and finally sum the partials before finding the means.

A slight complication is that we are only interested in computing the means of the common sub-sets, that is the vector components that have been pair-matched in the point-pair matching process discussed in section 3.8 above. The first part of this process is illustrated in the code fragment shown on figure 3.18 below. The two vectors **refVec** (the reference vector) and **pointVec** (the latest data) are processed together using the **dMatch** vector to test for matched pairs, the corresponding components are then independently summed into the holding vector **WorkVec,** this is initialised to zero before this kernel is called.

```
// GPU – compute_mean_2 – Part 1
    const int cellx = threadIdx.x + blockIdx.x * blockDim.x;
    const int startv = cellx*SegSzX;   // the start position for this segment
    const int startw = cellx*workSzX;  // the start position for WorkVec

// sum all vectors in this thread, store result in WorkVec[cellx]...
    for( i=0;i<SegSzX;i++){
        const int in2r = startv + i;   // refVec is the master
        if(in2r >= refCount) break;    // last segment may exceed refCount so break
        const int in2v = dMatch[in2r]; // pointVec location from dMatch
        if(in2v < 0) {
            continue;                  // if not matched exclude the point
        }
        WorkVec[startw]   += 1.0;      // checkSum should == MatchCount
        WorkVec[startw+1] += refVec[in2r            ];      // refX
        WorkVec[startw+2] += refVec[in2r +   pntSz.x];      // refY
        WorkVec[startw+3] += refVec[in2r + 2*pntSz.x];      // refZ
        WorkVec[startw+4] += pointVec[in2v            ];    // pointX
        WorkVec[startw+5] += pointVec[in2v +   pntSz.x];    // pointY
        WorkVec[startw+6] += pointVec[in2v + 2*pntSz.x];    // pointZ
    }
```

Figure 3.18 Computing the Means refVec and pointVec – Part 1

The second part of this process is given in Figure 3.19 below.

```
// GPU – compute_mean_2 – Part 2
    unsigned int countValue = atomicInc(&countMean, blockDim.x);
    compute_meanDone = (countValue == (blockDim.x - 1));

// only the last thread to execute will do this part
    if(compute_meanDone ){
        for(int i=0;i<countMean;i++){
            Sum += WorkVec[i*workSzX];
            RefX += WorkVec[i*workSzX+1];
            RefY += WorkVec[i*workSzX+2];
            RefZ += WorkVec[i*workSzX+3];
            PointX += WorkVec[i*workSzX+4];
            PointY += WorkVec[i*workSzX+5];
            PointZ += WorkVec[i*workSzX+6];
        }
        dResVec[0] = RefX/Sum; dResVec[1] = RefY/Sum; dResVec[2] = RefZ/Sum;
        dResVec[3] = PointX/Sum; dResVec[4] = PointY/Sum;dResVec[5] = PointZ/Sum;
        countMean=0;
   }
```

Figure 3.19 Second Part of Computing the Means refVec and pointVec

The key to this method is ensuring that all threads have executed before the partial fragments are finally summed. This is facilitated by a couple of GPU-specific, file-scope variables declared in the GPU code file thus:

__device__ unsigned int **countMean** = 0;
__shared__ bool **compute_meanDone**;

The **countMean** variable keeps track of the number of threads that have completed their partial summation using an atomic increment to ensure the count is correct. These are simple linear 1-dimensional GPU blocks, so the sequence is complete when all threads in **blockDim.x** have completed. This is indicated when **compute_meanDone** == **true**. All threads will pass this way, but only one thread (and it does not matter which) will do so when this is true. At that point the final means can be computed from the sub-totals contained in the global vector **WorkVec**. Having completed this, **countMean** is reset for the next launch.

The same reduction methods are used for compute_P2P_Error_2, compute_Covariance_2 used in the P2P algorithm discussed in section 3.10 below and shown on Figure 3.20. It is also used for compute_P2L_Error and compute_P2L_Update discussed in section 3.11 and shown on figure 3.21 below.

The remaining GPU algorithms were written, developed and tested by the author, and are described rather more briefly in the following sections 3.10 and 3.11.

3.10 Point-to-Point Registration Algorithm

The Point-to-Point (P2P) registration process was initially selected as it is one of the simplest. Although it is often discussed as part of an Iterative Closest Point (ICP) algorithm, this method does not of itself require iteration. This is a Least-Squares method, and given good correspondence between two point-cloud based images, the algorithm will compute the registration transformation which minimises the L2 norm accurately and in one step. Unfortunately, the registration process for depth camera data is more complex with many sources of error between the images to be registered. Known issues which affect registration process quality are:

- Depth camera sensor noise.
- Image partial occlusion due to change in viewpoint caused by camera movement.
- Regions of non-overlap between successive images due to camera motion.
- Adverse lighting conditions (sunlight and artificial light with a high UV content).
- Low surface reflectivity of viewed objects.

In addition to the above which will affect frame-to-frame registration quality causing a residual drift build-up due to incremental errors in registration, there are also major confounding factors for this type of visual robot tracking system including:

- Objects being too close or too far from camera to be correctly imaged.
- Mode-collapse due to camera facing a simple flat surface (e.g., roll becomes undefined).
- Exogenous motion in the scene due to other moving bodies (e.g., pets, people, or machines).
- Exogenous motion due to deformable items (e.g., plants or curtains moving in a breeze).

However, in common with many other similar studies, these effects will not be included in this analysis. The fundamental assumption being that the scene is static, rigid-body and the only moving component is the robot. The point-to-point algorithm discussed in section 2.3 is shown as implemented in figure 3.20 below.

The P2P algorithm shown in outline on figure 3.20 below and is intended as a guide to the actual code rather than a definitive description of it. This is a host (CPU) function calling multiple GPU kernel launchers within each box on the diagram. The _Launcher postfix has been omitted to reduce the diagram size. With the exception of computations, where conventional mathematic/code are shown, key inputs to the kernels are given inside the function (brackets) with primary outputs given on the line below the launcher function. Section numbers discussing the internals of key functions are also given in the text boxes.

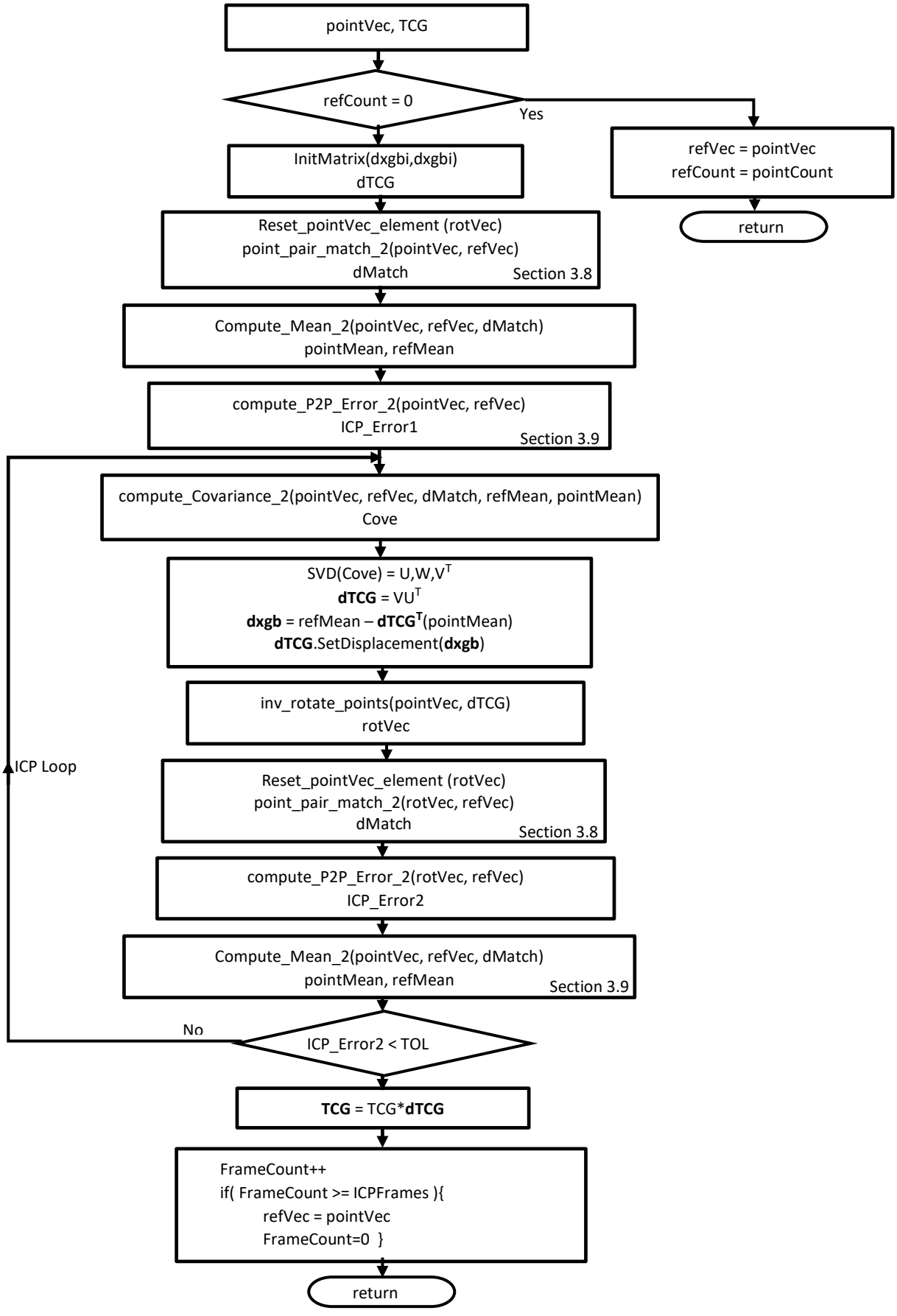


Figure 3.20 Point-to-Point Algorithm

The objective of the P2P algorithm is to update the current global pose **TCG** from the computed delta transformation **dTCG** based on the PCA reduced camera data contained in **pointVec** and the equivalent data from a reference frame contained in **refVec**, hence for the first pass **RefCount = 0** and **refVec** is simply equated to **pointVec**.

point_pair_match_2 forms pair-matching of points in **pointVec** and **refVec** see section 3.8 above for a detailed discussion of this algorithm.

Compute_Mean_2 produces the mean vertex over the data in **refVec** and **pointVec** for the pair-matched components as discussed in section 3.9 above.

compute_P2P_Error_2 then computes the L2 norm of the difference between **pointVec** and **refVec** based on the **dMatch** vector of associations. Its output **ICP_Error1** is a simple cost function based on the L2 delta norm divided by the number of matched points in **dMatch**.

compute_Covariance does exactly that in accordance with Equation 2.45 for the pair-matched **pointVec** and **refVec**. The output **Cove** is then used to compute the vectors **U** and **V** from the Singular Value Decomposition (**SVD**) from which **dTCG** is then computed.

A weakness of this algorithm is that both **dTCG** and **dxgb** are coupled thus errors in **dTCG** will propagate further errors in the displacement vector **dxgb**.

The **SVD** algorithm used here is taken from companion code provided with [33] subsequently modified for the data types used here.

Since the delta transformation **dTCG** and **dxgb** computed above represent the delta transformation between **pointVec** and **refVec**, inverse rotation of **pointVec** by **dTCG** giving **rotVec** should align closely with **refVec**. This is achieved with **inv_rotate_points**.

point_pair_match_2 is again called; this refines pair matching associations between **rotVec** (the rotated **pointVec**) and **refVec**. This is the essential (only) purpose of the ICP loop.

compute_P2P_Error_2 then computes **ICP_Error2** between the rotated **pointVec** and **refVec**. If this error is less than **TOL** the loop completes and the global tracking transformation **TCG** is updated. If this is not the case the ICP loop is repeated, the only change being the point-pair associations found between the rotated pointVec (rotVec) and refVec leading to a revised covariance and **dTCG** update.

However not shown in the algorithm above is a test to check if ICP_Error_2 is reducing relative to the initial ICP_Error_1. If not, the loop terminates after 5 iterates. It is found that this loop resolves (or fails to converge) within very few iterates owing to the least-squares nature of the point-to-point algorithm. Test evaluation of this algorithm is presented in chapter 5 below.

3.11 Point-to-Plane Registration Algorithm

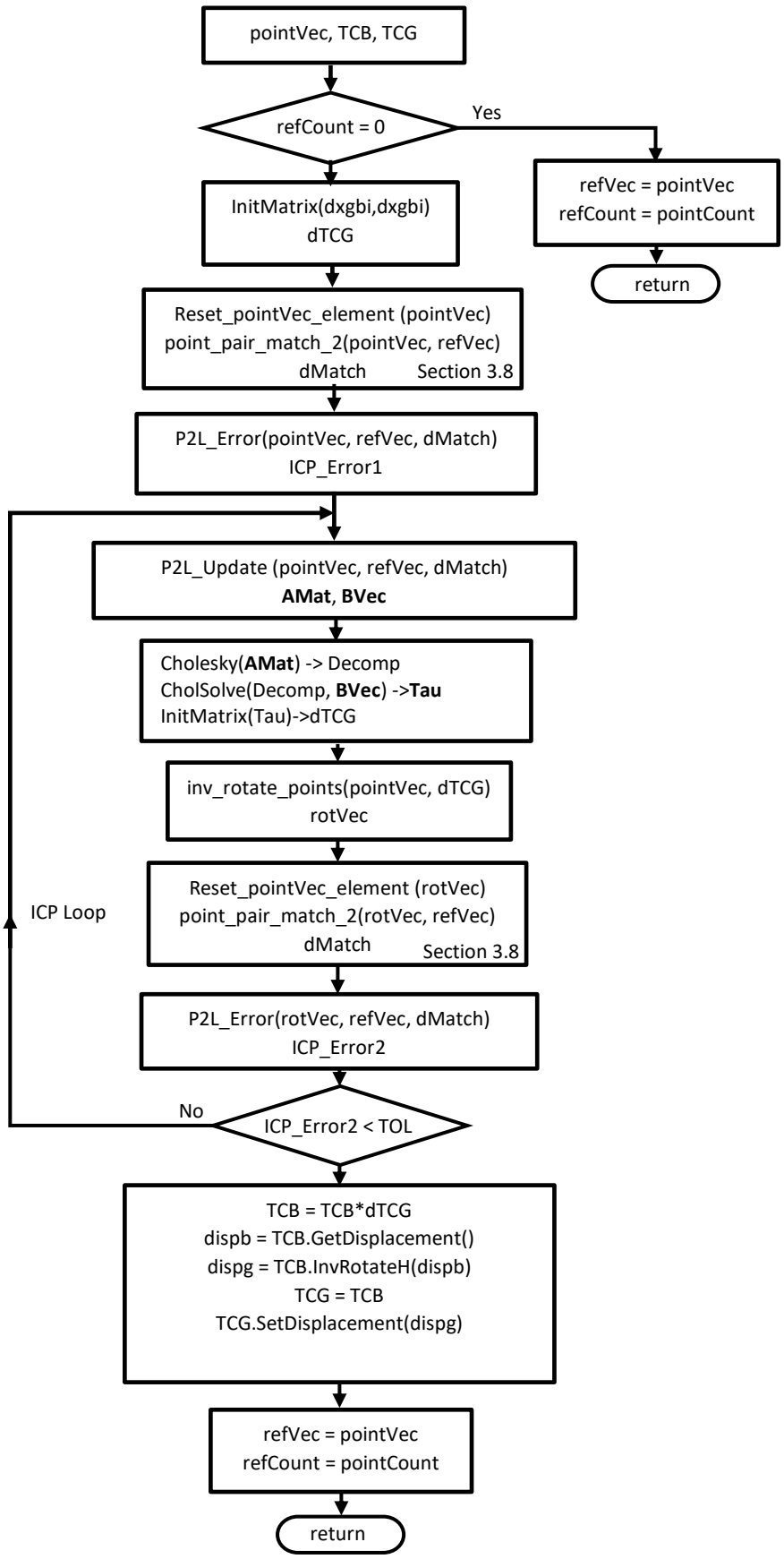


Figure 3.21 Point-to-Plane Algorithm

The point-to-plane (P2L) algorithm illustrated on figure 3.21 above shares many common features with the point-to-point algorithm discussed in section 3.10 and these need not be discussed further here. The mathematical development of the method is described in section 2.5 above.

P2L_Update computes **AMat**[6x6] and **BVec**[6] for the pair-matched data in **pointVec** and **refVec**. This is a GPU kernel so the data contained in **pointVec** and **refVec** are processed in parallel segments using similar methods to that used for **Compute_Mean_2** described in section 3.9 above. This kernel involves significant matrix and vector processing so, for each thread segment, the local memory required is mapped to a GPU global **dWorkVec** vector. Figure 3.22 below shows the initial mapping section.

```
// GPU – compute_P2L_Update Part 1
   float *Bsub, *Csub, *pk, *qk, *ck, *dk, *nk;   // vector data pointers
    const int cellx = threadIdx.x + blockIdx.x * blockDim.x;
    int startv = cellx*SegSz;
    int startw = cellx*WorkStep;
//
// start index for this thread into dWorkVec
    Index=startw;
    GMat Asub(GetData(dWorkVec,Index,36),6,6);
    Bsub = GetData(dWorkVec,Index,6);
    Csub = GetData(dWorkVec,Index,6);
    pk = GetData(dWorkVec,Index,3);
    qk = GetData(dWorkVec,Index,3);
    ck = GetData(dWorkVec,Index,3);
    dk = GetData(dWorkVec,Index,3);
    nk = GetData(dWorkVec,Index,3);
```

Figure 3.22 Initial dWorkVec Memory Mapping for P2L_Update Part 1

The **Index** parameter is initialised to the computed **startw** location in **dWorkVec,** based on the variable **cellx** for this thread's workspace**.** This is a large GPU global memory allocation set up during program initialisation and prior to any call. The **GetData** function provides pointer mapping into **dWorkVec** to allocate data space internally for each of the matrices and vectors used.

GetData simply returns the pointer at the current **Index** location in **dWorkVec,** this pointer is then passed to a **GMat** matrix object **Asub** which is sized as a [6x6] matrix. Before exiting **GetData** bumps **Index** by the size of the last argument (36 in the case of the first call) so that **Index** is now set to the correct location in **dWorkVec** for the next call (**Bsub** in the above example).

For each of the remaining vectors, the pointer at the current **Index** is returned and **Index** is incremented by the allocation. Thus, all of the memory used in this GPU kernel is contiguously allocated from **dWorkVec** for each thread segment.

The details of the body of this kernel shown in figure 3.23 follow the matrix and vector mathematics given in section 2.5 and since they map closely in both name and operation, are not repeated here.

In the final part of the algorithm, using the same vector reduction method used for computing the means given in section 3.9 above, the sub-matrices **Asub** and **Bsub** are summed into **AMat** and **BVec** which are in turn mapped to the results vector **dResVec** as shown in the code fragment figure 3.23 below.

```
// GPU – compute_P2L_Update – Part 2
    if(compute_P2L_updateDone){
// Sum A and B matrices for all threads in this kernel
        Index=0;
        GMat AMat(GetData(dResVec,Index,36),6,6);
        BVec = GetData(dResVec,Index,6);
        AMat.Zero();
        for(i=0;i<6;i++) { BVec[i] = 0.0;}
        for(k=0;k<P2L_updateCount;k++){
            Index = k*WorkStep;
            GMat Asub(GetData(dWorkVec,Index,36),6,6);
            Bsub = GetData(dWorkVec,Index,6);
            for (i=0;i<6;i++){
                BVec[i] += Bsub[i];
                for(j=0;j<6;j++){
                    AMat(i,j) += Asub(i,j);
                }
            }
        }
        P2L_updateCount=0;
        compute_P2L_updateDone=false;
    }
```

Figure 3.23 Final Summation and Memory Mapping for P2L_Update

The results are returned as **AMat** and **BVec** and then used as shown on Figure 3.21 above in **Cholesky** and **CholSolve** to produce the six-parameter vector **Tau,** this is then used to form **dTCG**. The **Cholesky** and **CholSolve** codes are implemented on the CPU transcribed from algorithms given in [33].

3.12 Cell Search Algorithm

This method builds on the PCA decomposition process detailed in section 3.6 above to produce the Förstner-Moonen (FM) similarity metric between any two point-cloud regions. Further details of the mathematics of the FM method are given in section 2.9 above. This method is used as a further augmentation of the P2L method (known here as CellSearch) by providing an additional FM similarity test in the point-pair matching process, excluding any pair-matches that are not similar in shape.

To illustrate the working of the metric, the Matlab image in figure 3.24 below shows an inverted chair with point cloud regions C1 to C10 identified on the plot. The inversion is due to the different axis system used in Matlab and the ServoTrack Image capture software.

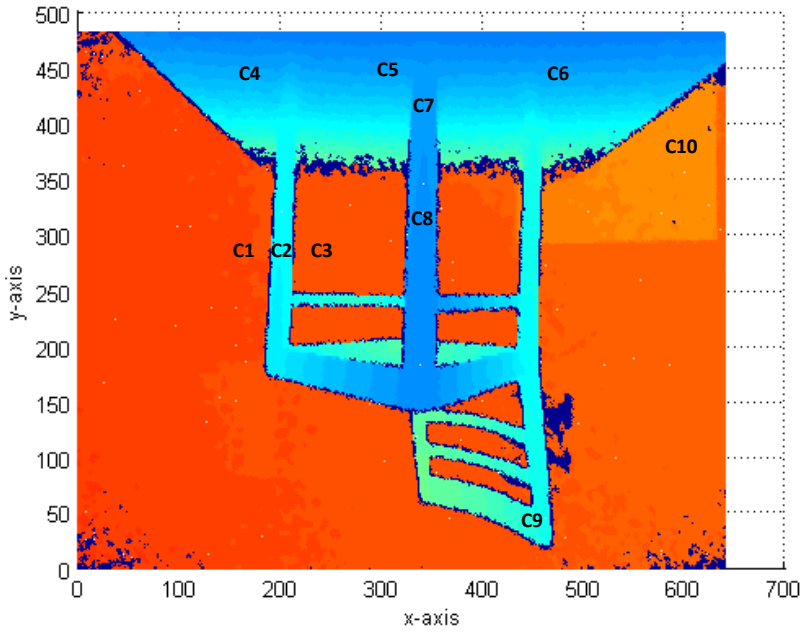


Figure 3.24 Inverted Depth Image of Chair with Point Cloud Regions Identified

The following table shows the FM similarity metric for all of the regions identified in figure 3.24 above; the table is symmetric about the lead diagonal; hence the lower half is not shown.

FM	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
C1	0	16.979	**4.6618**	65.777	62.778	64.892	19.068	25.178	32.731	7.475
C2	16.979	0	8.2196	21.427	19.842	19.306	**2.1271**	**3.1624**	8.9228	7.1823
C3	symmetric		0	45.812	45.015	46.564	13.785	18.288	18.624	**3.1349**
C4				0	**2.7733**	**5.7745**	26.408	27.777	25.498	45.147
C5					0	**3.1721**	22.774	23.307	26.207	42.97
C6						0	21.9	21.101	32.317	43.519
C7							0	**0.5424**	13.725	11.022
C8								0	16.401	14.713
C9									0	18.331
C10										0

Table 3.1 Showing the FM-Metric for the Identified Regions on Figure 3.24

Since the FM metric identifies similarity between regions, the smaller the metric value the closer in "shape" the point cloud is to the region being matched (with FM=0 for identical regions), the smallest pair-matches (most similar regions) are highlighted in **bold** in table 3.1 above.

From this table, it can be seen that C1 is closest in shape to C3 not surprising given that both are located on the back wall of the test image. It can also be seen that C3 is closest to C10 a flat radiator 100 mm in front of the back wall. Item C2 is clearly closely connected to C7 and C8, all are associated with the chair legs. Items C4, C5 and C6 are obviously connected all are part of the table surface on which the chair is placed. Item C9 is not strongly connected to anything but most closely associates with C2.

Table 3.1 shows both the appeal and limitations of this metric, in that it is not strongly differentiating but does indicate when regions are geometrically similar and thus provides a further test of the similarity when pair-matching the PCA-reduced point cloud segments.

The CellSearch algorithm is very similar to the point-to-plane algorithm above where the **point_pair_match_2** GPU kernel launchers are replaced by a **pca_pair_match** kernel launcher. The key difference is the FM similarity test on the candidate point-pairs in the matching process which includes thresholding on the FM-metric. Following the basic pair-matching process used in **point_pair_match_2** and discussed in section 3.8 above, the additional code shown in figure 3.25 below is included in **pca_pair_match**

```
// GPU – pca_pair_match
//
// perform the FM test on points that meet the above pair-matching criteria

   if(idxMin>=0){
      float FM = ComputeFM(idxMin, refIdx, pointVec, refVec, pntSz, dWorkMat, iTest );

// perform the similarity test
      if(FM>FMThresh){
         idxMin = -idxMin;
      } else{
         atomicAdd(&dAtomVec[5],1);   // keeps a count of the number of FM matches made
      }
   }
//
   dMatch[refIdx] = idxMin;
```

Figure 3.25 FM Similarity Test Included in pca_pair_match

As can be seen from the figure 3.25 above, most of the work for this test is carried out in the GPU kernel function **ComputeFM**.

Some steps are required in preparation for **ComputeFM**, the process starts within PCA Reduction and though not covered in section 3.5, there are some additional steps within **PCA_Reduce** which precomputes data needed to form the FM metric. The code fragment in figure 3.26 below shows the further actions taken in **PCA_Reduce** in preparation for **ComputeFM**.

```
// GPU – PCA_Reduce – Part 4
   sum=0.0; det=1.0;                          // compute FM intermediates
   for(i=0;i<3;i++){
      SLambdaM1[i] = rsqrtf(Lambda[i]);       // rsqrtf is a GPU specific = sqrt(1/Lambda)
      sum += Lambda[i];
   }

   for(i=0;i<3;i++){
      for(j=0;j<3;j++){ C(i,j) = S(i,j);  }          // recover the Covariance C stored in S
   }
// post multiply by V'
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){ W(i,j) = SLambdaM1[i]*V(j,i); }        //  [1/sqrt(Lambda)].V'
      }
   }

// pre-multiply by V giving AM1 = inverse of A
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         AM1(i,j) = 0.0;
         for(k=0;k<3;k++){
            AM1(i,j) += V(i,k)*W(k,j);                          // V.[sqrt(1/Lambda)].V'
         }
      }
   }
//
// Store the covariance matrix in pointVec
// this is needed for processing in ComputeFM and removes the
// need for two copies of dWorkMat from which the other matrices
// can be recovered
//
   int pointStart = PVEC_ELEMENT+1;
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         pointVec[myIndex + pointStart*pntSz.x] = C(i,j);
         pointStart++;
      }
   }
```

Figure 3.26 Additional PCA_Reduce Steps to Furnish Data for ComputeFM

The Covariance matrix **C** is passed in the elements of **pointVec**. The pre-computed $\mathbf{A}^{-1}$ (**AM1**) is passed via the common work-vector **dWorkMat** to **pca_pair_match** where it is then passed to **ComputeFM**. There is only one **dWorkVec** and so it is necessary to carry the data used to compute the metric by the two separate pathways otherwise the data from **pointVec** would overwrite the data in **refVec**.

Also, there is a need for **dWorkVec** to store the other working matrices used in **ComputeFM** below. This approach is reasonably space efficient however it does require that pointVec maintains its workspace location within **dWorkVec** this is held in **WVEC_START** in the following code fragment, figure 3.27 below.

The actual computation of the FM metric is carried out in GPU function **ComputeFM** outline details of which are shown in the code fragment figure 3.27 below. The mathematics for this is given in section 2.9

```
// GPU – ComputeFM

// recover the covariance matrix CRef from refVec
   int refStart = PVEC_ELEMENT+1;
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         CRef(i,j) = refVec[RefIdx + refStart*pntSz.x];
         refStart++;
      }
   }
//
// get the location of pointVec matrices in dWorkMat
   const int wrkSrc = (int)pointVec[PntIdx + WVEC_START*pntSz.x];
   float* pWorkSource = &dWorkMat[wrkSrc];
//
// matrices for pointVec CPnt & AM1Pnt
//
   int InIndex=0;
…
   GMat AM1Pnt(GetData(pWorkSource,InIndex,9),3,3);
…
// Compose SPnt = AM1Pnt*CRef*AM1Pnt

   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         VPnt(i,j) = 0.0;
         for(k=0;k<3;k++){ VPnt(i,j) += CRef(i,k)*AM1Pnt(k,j); }
      }
   }
//
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         SPnt(i,j) = 0.0;
         for(k=0;k<3;k++){  SPnt(i,j) += AM1Pnt(i,k)*VPnt(k,j); }
      }
   }
// Perform Jacobi iteration on SPnt to get Eigen-values Lambda on lead diagonal
   iTest += Jacobi_GMat(SPnt, VPnt, WPnt, RPnt);

// The FM metric is then formed as the sum of log^2(Lambda(i))
   float FM = logf(SPnt(0,0))*logf(SPnt(0,0))  + logf(SPnt(1,1))*logf(SPnt(1,1))
                  + logf(SPnt(2,2))*logf(SPnt(2,2));
```

Figure 3.27 ComputeFM – Computing the Förstner-Moonen Metric

3.13 Point-to-Plane with Loop Closure

It is inevitable that ICP registration methods, when used for robot tracking, will lead to long-term drift. This is because the integration process governing the global pose transformation step namely:

$$T_{CG\,t+1} = T_{CG\,t} dT_{CG} \tag{3.14}$$

Is essentially open loop. The revised global position is incrementally modified by the last registration step in dT_{CG} but, there is no intrinsic means to correct for drift buildup due to small, but nonnegligible, registration errors. However, a 3D model of the environment is simultaneously being generated by fusion of depth camera image data. Since drift buildup is a gradual process, in theory it should be possible to use the data in the 3D model to provide a corrective to the incremental drift process.

This process, known as “loop closure” in robot tracking studies, is not strictly a feedback loop closure as understood in Control Systems parlance since, the “loop closure” step will be based on the accumulation of past data within the 3D model. This data has also been subject to incremental drift, so this can never be an absolute loop closure. In theory the method should significantly curtail drift rate and hence improve overall tracking accuracy, especially over long runs which include revisiting scenes previously built into the 3D model. The Point-to-Plane with Loop Closure algorithm, proposed in this study, is summarised in figure 3.28 below.

As shown on Figure 3.28 below, if **ClosedLoop** is not selected or the current **Frame** does not match **CLStep** the loop is open and essentially the same as the point-to-plane algorithm discussed in section 3.11 above. With the loop closed however further steps are taken, the most significant being the synthesis of a virtual depth camera image **v_Depth** from the 3D model held in **d_Vol** taken at the latest **TCG** position, this is carried out in **Depth_From_Vol** and discussed further in section 3.14 below.

This **v_Depth** image is then used in a second call to **PCA_Reduce** but this time to update the reference vector **refVec**. Both **pointVec** and **refVec** are now updated and used in a second call to **P2Plane_ICP** to re-compute **TCG** as a corrective step to the open loop ICP process.

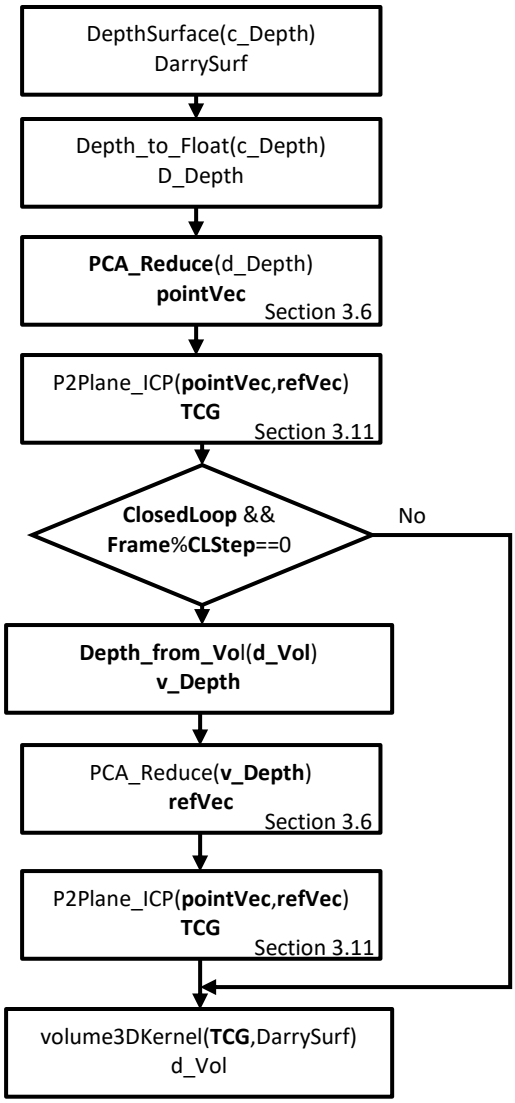


Figure 3.28 Point-to-Plane with Loop Closure Algorithm

3.14 Depth Recovery from Model Volume (Depth_from_Vol)

The principle of operation of the recovery of a depth image from the 3D model is ray-tracing from a virtual camera with the same pixel coordinate system as the real depth camera. This camera is located and oriented in the 3D model at the current pose contained in **TCG**. Then for each pixel over the array, **p(u, v)** a ray is traced into the model volume using the same projection equations (3.3) defined for building the model originally. When this ray contacts a non-zero voxel element the distance along the ray **Z** is computed and correcting for the angle of the ray relative to the camera axis the depth **d** is computed in the virtual camera image frame. Thus, for every pixel in the camera array a triad (u, v |d) is computed. The principle is illustrated in figure 3.29 below.

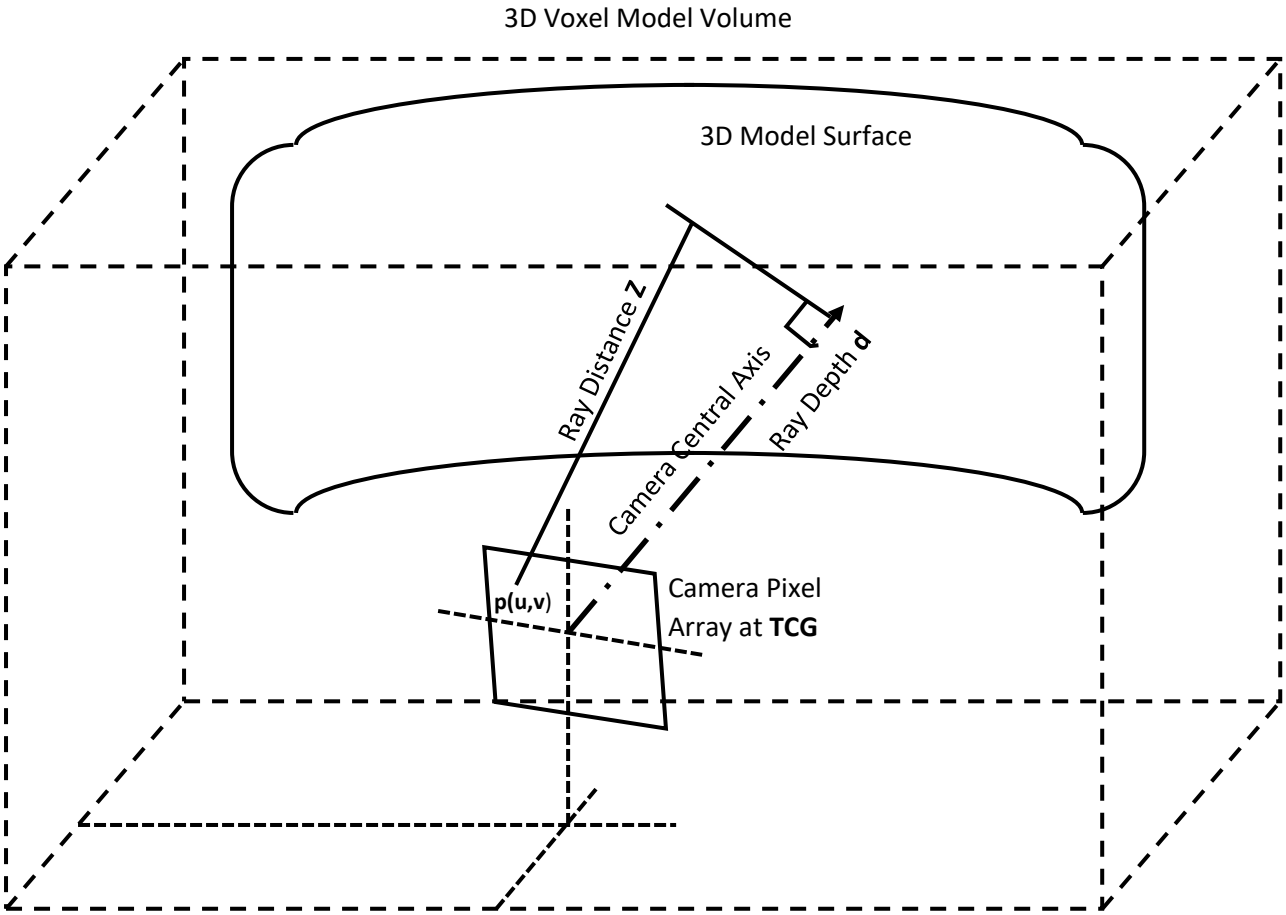


Figure 3.29 Showing Sketch of Depth Recovery from Model Volume

If the projected pixel ray does not contact a populated voxel within the bounds of the model, then its depth value is set zero. There will also be no return for any populated voxel closer than the minimum standoff for the camera, as is the case for the real camera. The rays are also limited to a maximum cut-off range as set for the real camera.

To simplify the computation the minimum and maximum points along the ray are computed using the transformation. Points along the ray are then traced (stepped) using linear interpolation. Thus, for each pixel ray only two transformation operations are required since no valid voxel contacts outside of this limited ray line are possible. The algorithm is presented in the code fragment shown on figure 3.30 below.

```
// GPU – compute_Depth_From_Vol

const int u = threadIdx.x + blockIdx.x * blockDim.x;
const int v = threadIdx.y + blockIdx.y * blockDim.y;

int in2d = u + v*arrSz.x;       // index into camera pixel array
v_Depth[in2d] = 0.0;            // default value

const float tanPsi   = float(u - CAM_CX)*CAM_Kinv;
const float tanTheta = float(v - CAM_CY)*CAM_Kinv;

CamMin.z = CAM_DMIN;
CamMin.x = tanPsi*CAM_DMIN;             // = tan(psi)*zmin
CamMin.y = tanTheta*CAM_DMIN;           // = tan(theta)*zmin
GlobMin = TCG.InvR(CamMin) + xgbf;      // in metres

CamMax.z = CAM_DMAX;
CamMax.x = tanPsi*CAM_DMAX;             // = tan(psi)*zmax
CamMax.y = tanTheta*CAM_DMAX;           // = tan(theta)*zmax
GlobMax = TCG.InvR(CamMax) + xgbf;          // in metres

PixMin = GlobMin*ViewScale;             // 3D minimum point on ray in voxels
PixMax = GlobMax*ViewScale;             // 3D maximum point of ray in voxels
PixGrad = PixMax-PixMin;                // 3D gradient along pixel ray

Ray pixRay = {PixMin, PixGrad};     // Ray is a 3D struct {float3 origin, gradient} Ray;

len = Length(pixRay);
dt = 1/len;
f = t = 0.0;

// hit detection loop
while (t<1.0 && f < 0.5) {
   t += dt;                                 // t interpolates along pixRay
   pos = paramRay(pixRay, t);               // paramRay performs 3D interpolation along pixRay
   ipos = make_int3(pos.x, pos.y, pos.z);
   f = d_vol[flatten(ipos, volSize)];       // obtain the value at this voxel
}

// if a hit is detected populate the camera pixel with the corrected depth
if(f>=0.5){
   const float deno = rsqrtf(1.0 + tanPsi*tanPsi + tanTheta*tanTheta);
   dep = t*len*deno/ViewScale + CAM_DMIN;
   v_Depth[in2d] = dep;
}
```

Figure 3.30 Depth from Volume Algorithm (Depth_from_Vol)

The global camera pose is provided as arguments in **TCG** and **xgbf**. For each pixel (**u,v**) in the virtual camera frame the angles **tanPsi** & **tanTheta** are computed and the local camera minimum and maximum coordinates computed. These are then converted into global minimum and maximum coordinates for the limits of this pixel ray. These are then converted into 3D model voxel coordinates and cast into a 3D struct **pixRay**. This ray is traced in the volume model for a hit (**f>=0.5**), if this is found, the loop exits and the camera depth corresponding to that voxel is computed and returned in **v_Depth.** This is the distance **dep** from the virtual camera along its polar axis, the same as would be computed by a real camera viewing a real scene.

Note some of the ray-limiting operations have been omitted from the above fragment to simplify the description as the algorithm also limits the ray to remain within the 3D model bounds.

This completes the detailed discussion of the GPU and CPU algorithms developed for this project. Their implementation and test are covered in detail in chapter 5 of this report. The following chapter discusses the development of the test hardware used to validate these algorithms.

4. Hardware Development and Testing

4.1 Depth Camera Testing

Three different Intel RealSense™ Depth Cameras were tested as the project developed. These are relatively new and quite expensive but not nearly as expensive as equivalent industrial devices costing many thousands of pounds which were well beyond the cost constraints of this project.

4.1.1 Depth Camera Performance Data

The D435i and D455 Depth Cameras shown on figure 4.1 below are binocular camera systems with a central IR illuminator and two IR sensors. Depth is computed using a dedicated internal ASIC by triangulation based on an IR sensor baseline of 50mm for the D435i and 95mm for the D455. This camera system is prone to dropout when both IR sensors cannot see the same return to compute the angle and hence depth of that region. This happens frequently at the edges of features in view. Frame rates of up to 90 frames per second (FPS) are possible but to reduce data accumulation rate and fit within bandwidth constraints 30FPS are used throughout.

Figure 4.1 Intel RealSense® D435i and D455 Depth Cameras

Figure 4.2 shows a L515 Lidar camera, it is based on a raster scanning IR laser. A MEMS device controls scanning of the laser beam, and an IR photodiode detector senses the return. Depth is then computed by time-of-flight of the light pulse. The maximum frame rate for this device is 30 FPS.

Figure 4.2 Intel RealSense® D515 Lidar Camera

The following table gives comparison for the cameras for published data taken from [37] and [38] and some tested camera parameters. Intel recommended camera settings are indicated in bold, though for consistency all cameras were tested in VGA (640x480) mode. Note the L515 camera has since been discontinued, the reasons for this are not known.

	Device		
Parameter	**D435i**	**D455**	**L515**
Technology	Binocular IR	Binocular IR	Scanning IR Lidar
IR Baseline (mm)	50	90	N/A
Depth Image VGA	640x480	640x480	**640x480**
Depth HD/XGA	**848x480**	**848x480**	1024x768
FPS	6,15,30,60,90	6,15,30,60,90	30
Depth Resolution	16bit UINT (1.0 mm)	16bit UINT (1.0mm)	16bit UINT (0.25 mm)
Depth Range[1]	0.28 to 3.0m	0.32 to 4.0m	0.53 to 4.0m
Depth FofV VGA (deg)	H:75±3 V:62±1	H:75 V:62	H:70 V:55
Depth Accuracy (spec.)	≤2% up to 2m	+/- 2% up to 4m	<5mm @1m, <14mm @9m
Depth Std. Dev.	24-37mm @ 2.5m	< 2% of Depth	<5mm @ 3.0m
Lighting Condition	Not stated	Not stated	< 500 lux for spec. perf.
Wavelength / power	850±10 nm / 150mW	850±10 nm / 360mW	844-875nm / 240mW
RGB Sensor	Omni Vision OV2740	Omni Vision OV2740	Omni Vision OV2740
RGB Format/FPS	1920 x 1080 / 30	1920 x 1080 / 30	1920 x 1080 / 30
RGB FofV (degrees)	H:69.4 V:42.5	H:69.4 V:42.5	H:69 V:42
IMU (6DOF)	Bosch BMI055	Bosch BMI055	Bosch BMI085

Table 4.1 Comparing Camera Performance Parameters

[1] The Depth Range shown above is less than that published in the datasheets [37] & [38] but found by test to give reliable depth returns. All are devices are designated Class 1 (eye-safe) lasers.

4.1.2 Depth Camera Image Quality Under Different Lighting Conditions

During testing it was noticed that the quality of the depth data varied significantly from one camera to the other. Some of this variation was expected since the cameras have very different characteristics as seen on table 4.1. The quality of depth data was also seen to vary significantly with lighting conditions. This is illustrated qualitatively on the following figures 4.3 to 4.14. Figures 4.3 to 4.5 show RGB colour images under the different test lighting conditions taken with the L515 RGB camera. Note that although L515 is used for this test, the RGB camera is the same on all three devices. It is also deduced that there must be some lighting compensation within the RealSense Camera RGB image processing software since the lighting level varied considerably from the low light to fully illuminated test conditions and this is not reflected in the images reproduced on figures 4.3 to 4.5.

Figures 4.6 to 4.8 show depth images taken under very low lighting conditions with false-colour depth images obtained by screen capture from the Intel® RealSense™ SDK program: “realsense-viewer” with default settings operating in 2D mode. There were no other light sources than a small daylight leakage through closed curtains for these tests.

Figures 4.9 to 4.11 show results for the three cameras when the test scene is illuminated by a very powerful wall-mounted 4000 lumen, 40W (equivalent to a 100W incandescent) LED Strip Light. Surprisingly perhaps none of the camera’s depth images are significantly adversely affected by this light source.

Figures 4.12 to 4.14 show the same test but with a ceiling-mounted, 100W, 1800 lumen, 3000K Halogen light bulb. All depth images are impacted to varying degree by this light source. However, the quality of the L515 camera is clearly the most adversely affected by this source. Since all three depth cameras use ~850nm LED light sources to illuminate their target, lighting conditions are always going to be a concern for performance of this type of depth camera, given scan rate and key design constraint of a Class 1, eye-safe laser source.

From these tests it is very clear that the L515 scanning laser camera gives vastly superior quality depth images relative to both the D435i and D455 cameras. The two binocular IR cameras giving very similar quality output despite the much larger baseline (and size) of the D455 camera. This result was also surprising.

Figures 4.6 to 4.14 show only static depth images and consequently do not convey the truly dynamic nature of depth camera images (relative to RGB camera images). To better comprehend some of this

variability, a series of tests are performed to understand the accuracy of these devices for tracking and 3D model development. These results are presented in the following sections.

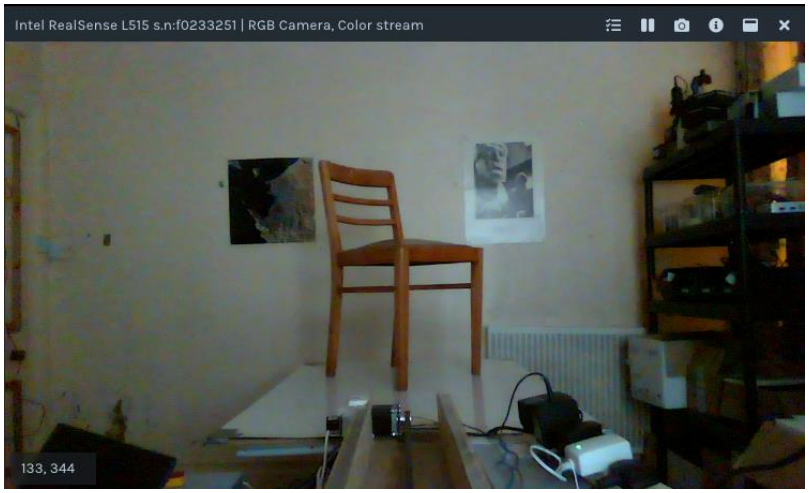


Figure 4.3 L515 Camera RGB Image Under Low Light Conditions

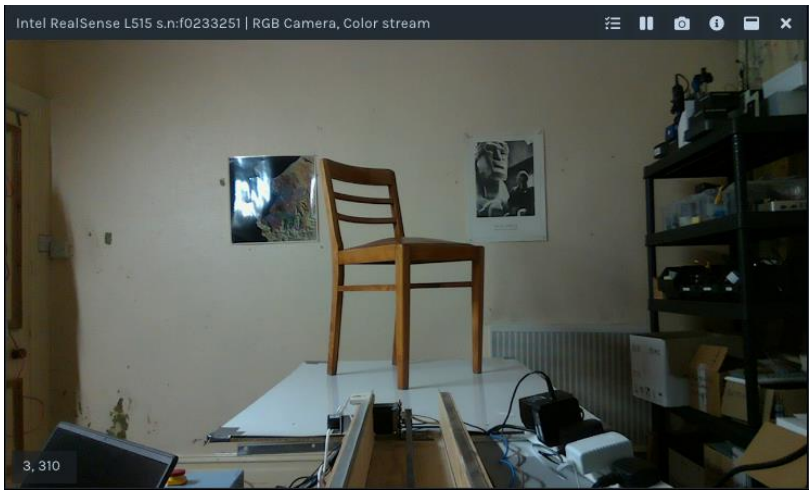


Figure 4.4 L515 Camera RGB Image Under LED Side Lighting

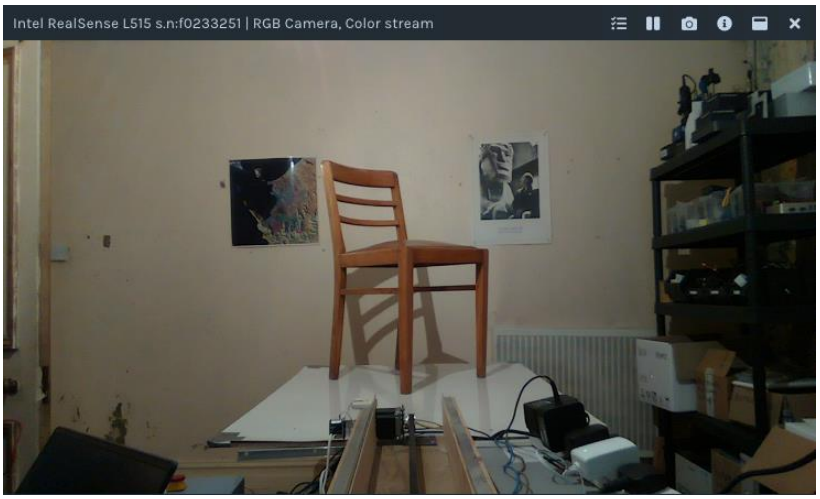


Figure 4.5 L515 Camera RGB Image Under Tungsten Overhead Lighting

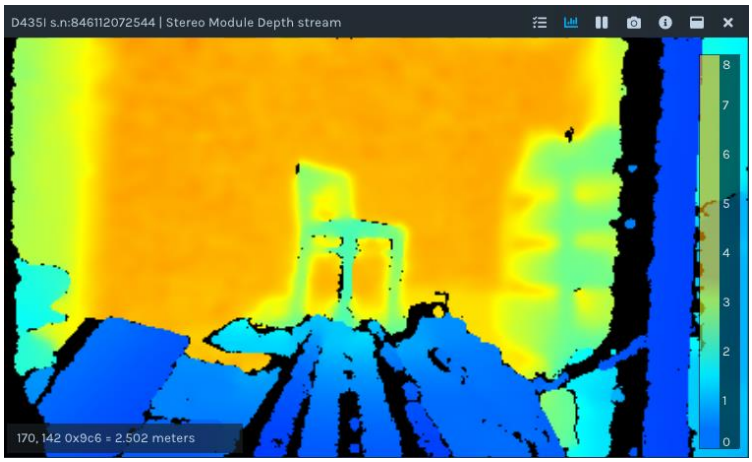


Figure 4.6 D435i False Colour Depth Image Under Low Light Conditions

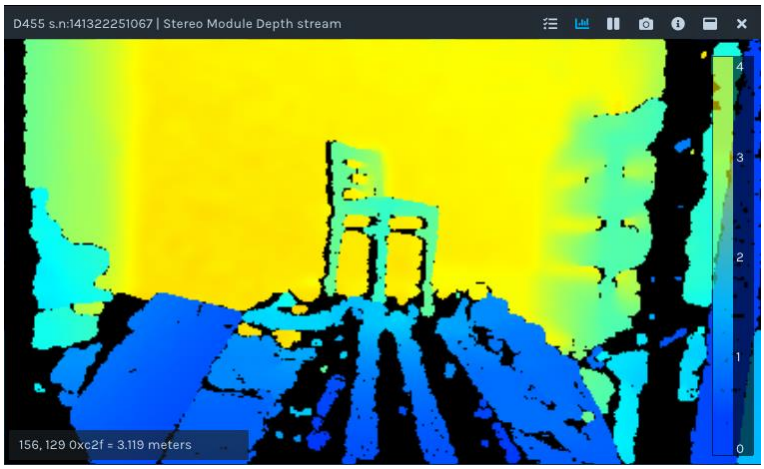


Figure 4.7 D455 False Colour Depth Image Under Low Light Conditions

Figure 4.8 L515 False Colour Depth Image Under Low Light Conditions

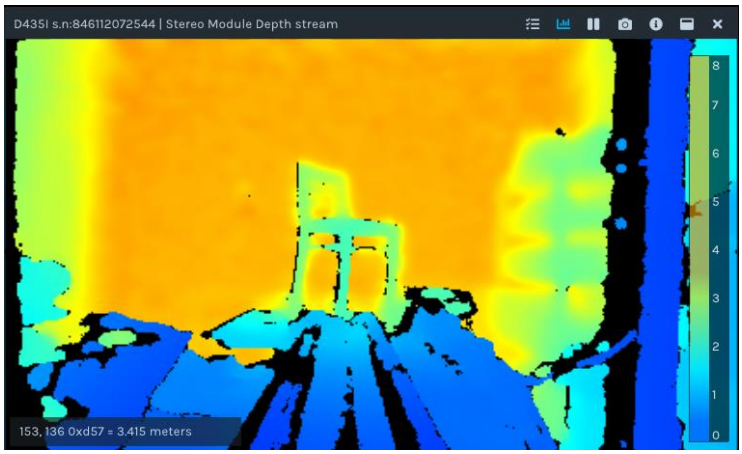


Figure 4.9 D435i False Colour Depth Image Under LED Lighting

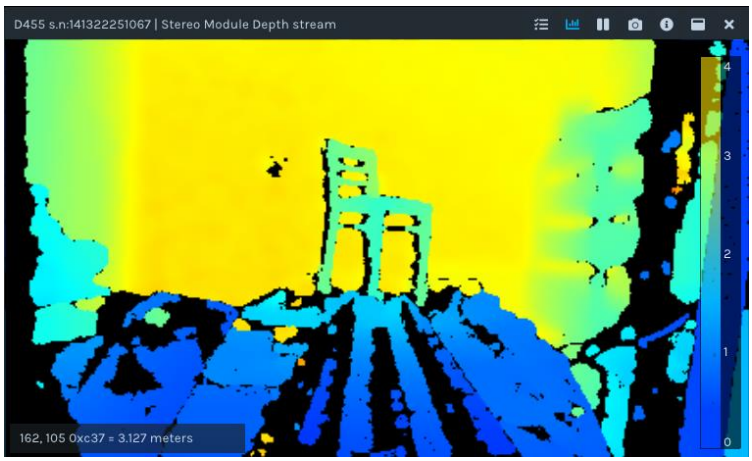


Figure 4.10 D455 False Colour Depth Image Under LED Lighting

Figure 4.11 L515 False Colour Depth Image Under LED Lighting

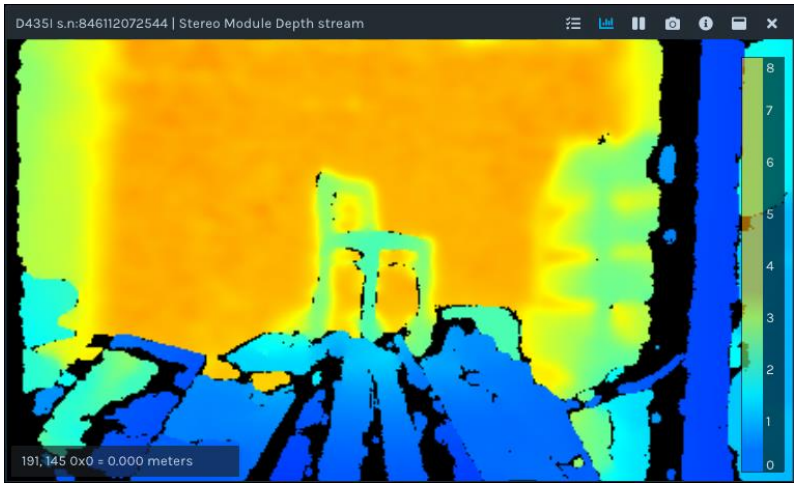


Figure 4.12 D435i False Colour Depth Image Under Tungsten Lighting

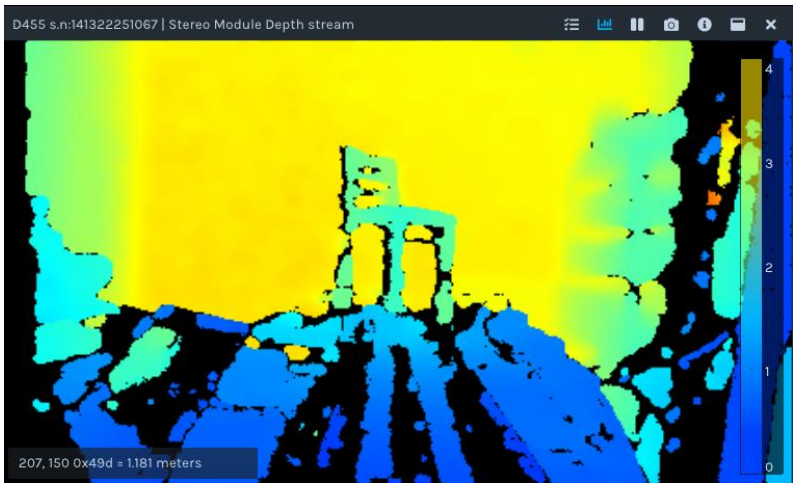


Figure 4.13 D455 False Colour Depth Image Under Tungsten Lighting

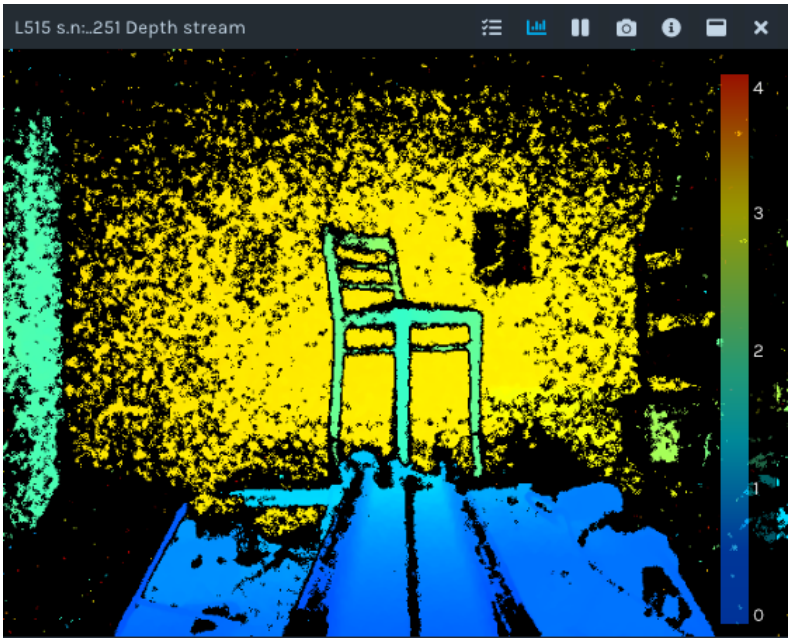


Figure 4.14 L515 False Colour Depth Image Under Tungsten Lighting

4.1.3 Depth Camera Calibration Testing

All depth cameras tested produce a 2-byte unsigned integer (UINT2) value indicative of depth returned within a pixel matrix in a similar manner to the colour values in an RGB image. To recover the three physical dimensions from the depth image located at pixel coordinates (u, v) the following projection equations are used.

$$Z = ViewScale * d(u, v) \quad (4.1)$$

$$X = (u - CAM\_CX) * Z * CAM\_Kinv \quad (4.2)$$

$$Y = (v - CAM\_CY) * Z * CAM\_Kinv \quad (4.3)$$

Where:

$ViewScale$ converts from the camera UINT2 depth return to depth in metres.

$d(u, v)$ represents the camera depth return at pixel image coordinates (u, v).

CAM_CX and CAM_CY are the centre pixel location for the camera

CAM_Kinv is the inverse of the camera projection constant CAM_K.

X, Y and Z follow the convention of computer graphical systems with X positive to the right, Y positive down and Z positive into the screen/page. The origin of the depth camera image being the upper and left most pixel.

Whilst the value of CAM_K or its more useful inverse CAM_Kinv can be derived simply from the published data, other aspects of the camera system, such as depth accuracy and lateral resolution are also of interest to this study. Data for these items are not available within the published performance data. Consequently, testing was carried out to better understand the performance of the three camera systems.

To test the camera calibration a simple flat round target of known diameter (0.6m) was set at a distance measured using an industrial class 2 laser device with a stated accuracy of +/- 1.5mm. Using test programs within the Intel™ Software Development Kit (SDK) downloaded from the RealSense® web-site [39] the distance to this target was also established from the camera centre pixel return. A false colour depth image of the test target is shown on figure 4.15 below.

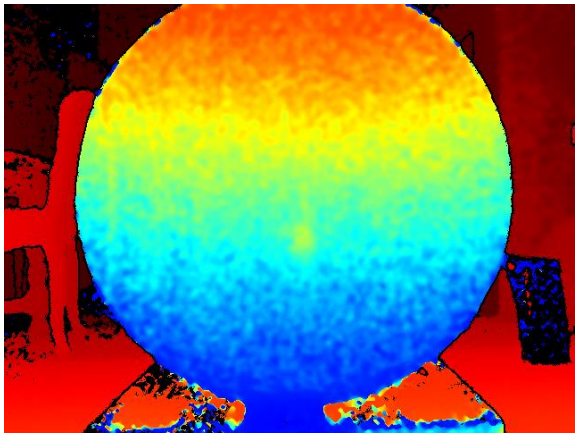


Figure 4.15 L515 Depth Test Image with 0.6m Diameter Target at 0.56m

The following table 4.2 shows a summary of results recorded from the test. The difference between an average of 500 points for the depth camera was compared with the physically measured depth. As can be seen from the first two columns the correspondence is very good. To get an understanding of lateral accuracy using image analysis the location of the edge pixels of the target near the centre were taken and the lateral dimension of the target in pixels was then estimated and from this the computed projection constant CAM_K and the target diameter is estimated. As can be seen the resulting lateral error is less than 2% over the range tested for this camera.

Measured Depth (m)	Camera Depth (m)	CAM_K	Diam. Est.	Error %
0.56	0.564	459	0.598	-0.20
0.6	0.599	461	0.602	0.32
0.76	0.763	463	0.604	0.63
1.2	1.221	466	0.608	1.27
1.47	1.475	467	0.609	1.53
1.77	1.777	456	0.595	-0.86
2	2.024	455	0.594	-0.99
2.2	2.226	453	0.590	-1.60
2.47	2.472	457	0.597	-0.57
2.75	2.768	457	0.596	-0.71
3	3.019	453	0.591	-1.57

Table 4.2 L515 Lidar Depth Calibration Test Data

A similar test methodology was carried out for the D435i and D455 Depth cameras and for the RGB camera. Data for the sensor systems tested is summarised in table 4.3 below.

Sensor	W	H	FoV (HxV)	Ar (H/W)	CAM_K(theory)	CAM_K(test)
RGB (all)	1280	720	69° x 42°	0.5625	931	928
L515 Lidar	640	480	70° x 55°	0.75	457	458
D435i Depth	848	480	87° x 58°	0.566	446	432
D455	640	480	87° x 58°	0.566	446	440

Table 4.3 Sensor Parameters for the L515 and D435i Cameras

4.1.3 Depth Accuracy Testing

In this section the accuracy of the depth cameras, their depth standard deviation as a function of mean distance is investigated. The test setup is slightly different to the depth constant calibration test shown above and illustrated on figure 4.16 below.

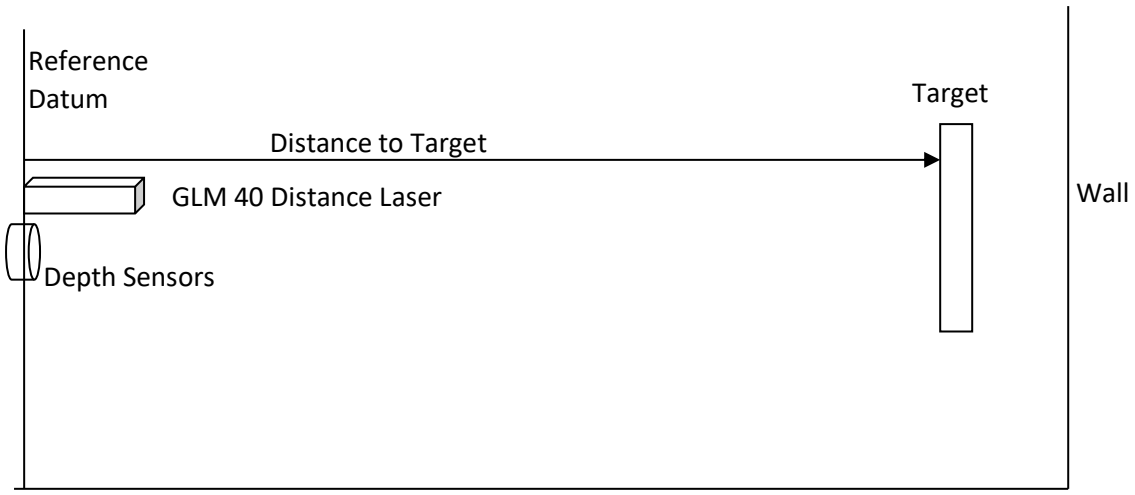


Figure 4.16 Schematic of Distance Measurement Test Setup

The sensors were tested separately, but the combined results are shown on the following graph. For the depth sensors, the Intel RealSense™ SDK program “rs_distance” was run for 500 samples and the mean value for each set distance recorded. The following Figure 4.17 shows the standard deviation on these depth measurements.

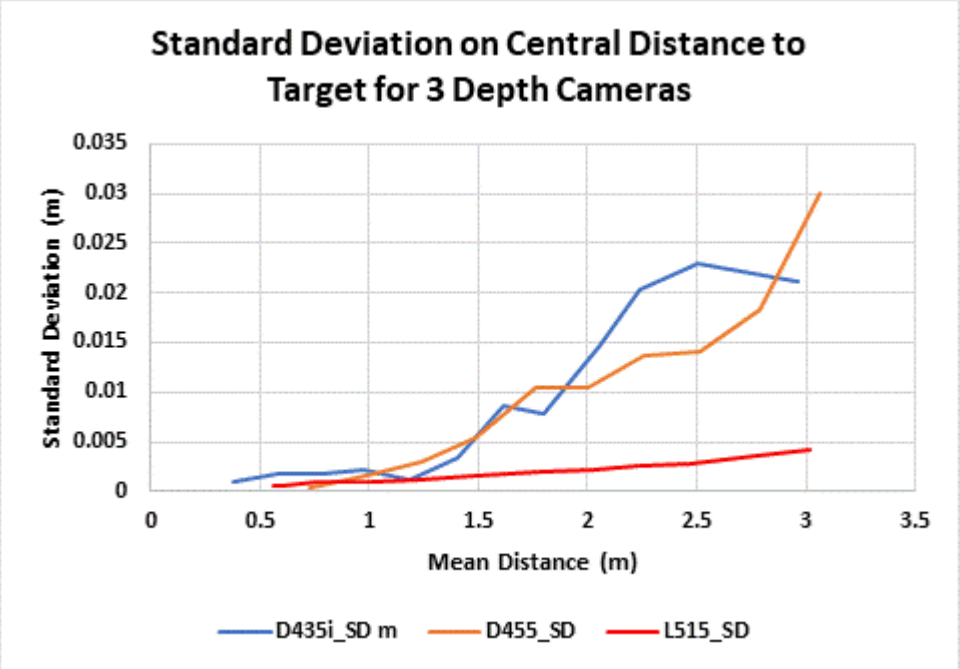


Figure 4.17 Comparison of Standard Deviation for the 3 Depth Cameras

The standard deviation for the D435i and D455 cameras exceeds 5mm (the size of the voxels used in the 3D models) at 1.5m whereas the L515 crosses 5mm above 3m clearly the L515 is a superior depth/distance measurement device in this test. The D455 camera shows some improvement over the D435i in the range 2-3m but this is far less than expected given the increase in baseline distance and camera size.

However, the above results show only part of the story, if data from the depth sensors are to be used to perform accurate image registration, then this will also be a function of lateral resolution.

4.1.4 Depth Image Lateral Resolution Testing

Depth image features were found by test to be particularly prone to erosion and distortion at feature edges, this is illustrated for two of the cameras on figure 4.18 below. This is also a function of distance from the camera. The results shown on figure 4.18 were obtained using the RealSense™ SDK program “rs-distance”. During this test an assessment of lateral accuracy of these devices is also carried out.

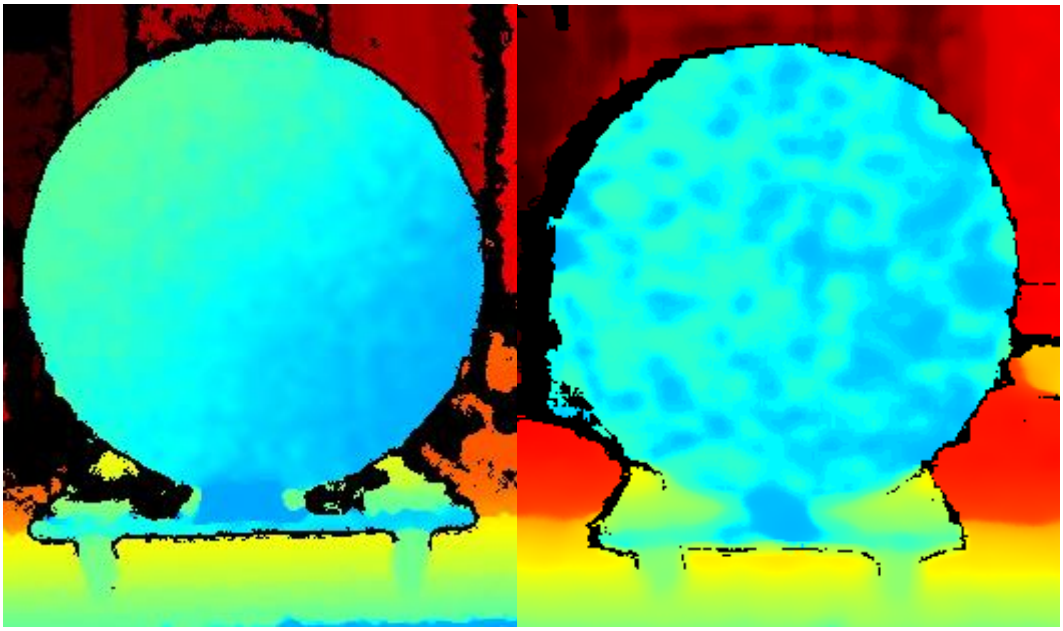


Figure 4.18 Target Depth Images from L515 (left) and D435i (right) at Distance of 1.2m

As can be seen in figure 4.18 above the depth image of the circular target on the right, for the D435i, is quite distorted with false readings extending radially outwards from the edge of the target. These edge features were not entirely static from frame to frame but were persistent nonetheless. Whist the comparable L515 image on the left is a much cleaner representation of the target, it also exhibits dropout (shown black) particularly behind the target and close to the ground plane.

Under ideal conditions, the depth sensor has a finite X-Y resolution and this relates to depth into the image. Assuming an isotropic sensor then for the L515 Lidar, 640 pixels in the X axis are uniformly shared over a 70-degree field of view. For each image pixel, the lateral distance covered will increase with depth, as illustrated on the following figure 4.19 below.

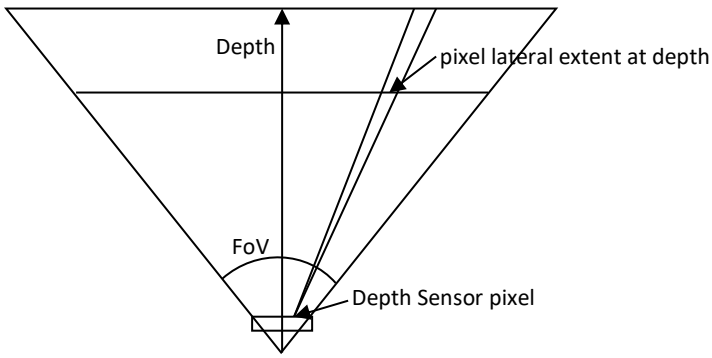


Figure 4.19 Illustrating Lateral Resolution with Depth

The depth camera cannot be expected to resolve objects smaller than the projected lateral extent of each pixel for a given depth. In practice this theoretical performance is likely to be made worse by other factors such as sensor noise, drop-out and the depth capture mechanics of the different devices. This is illustrated on a highly zoomed L515 image of a test placard shown on figure 4.20 below. Individual pixels are clearly visible and the camera behaviour is seen to be isotropic.

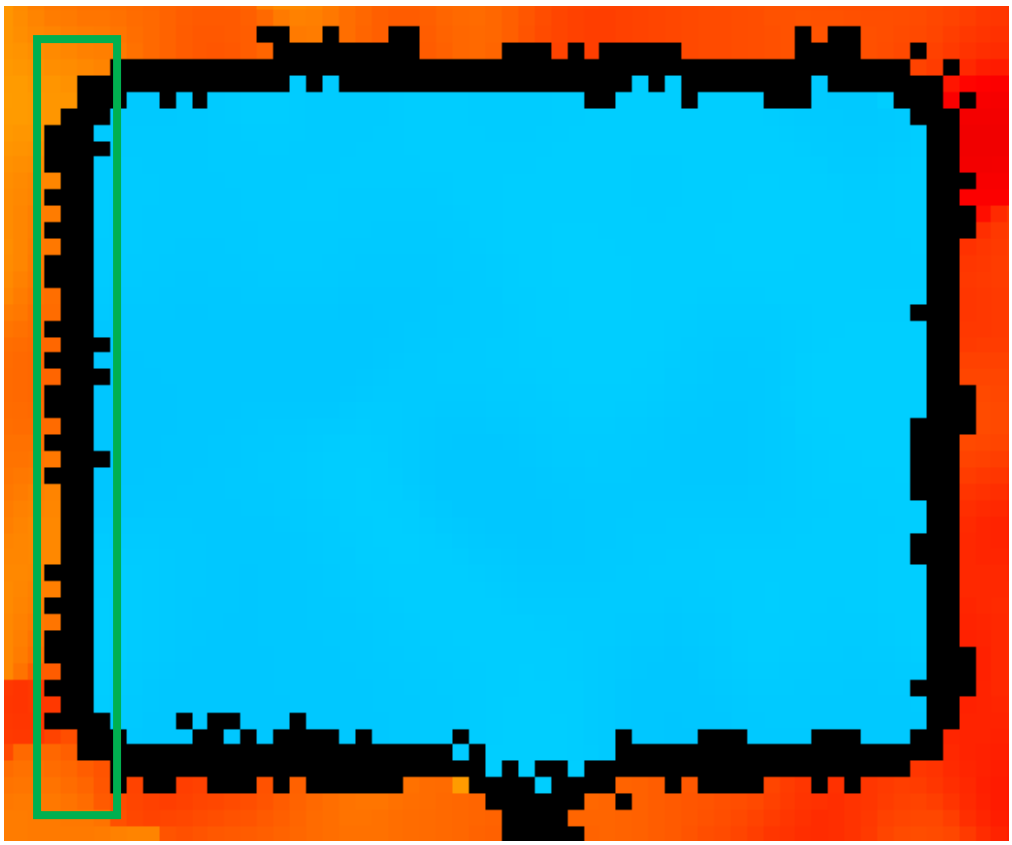


Figure 4.20 Showing Pixel Resolution of a Zoomed Test Placard Placed at 3.0m

The green rectangle to the left side of the image highlights a region within which it is clear that there is a solid line two pixels wide of dropout with further broken lines within and outside of the placard image. A similar pattern is visible along the other edges of the placard and from this simple visual observation an isotropic resolution factor of approximately 2.5 is associated with the pixel uncertainty of the edges of this object. Relating this to a lateral measurement resolution at this distance gives an X-Y resolution of 14.6mm at 3m compared to the ideal (single-pixel) value of 5.8mm thus the sensor is approximately 2.5 times lower in lateral resolution than the theoretical value at this distance. Repeating this test process for a number of placard distances (depths) gave the result shown on figure 4.21 below.

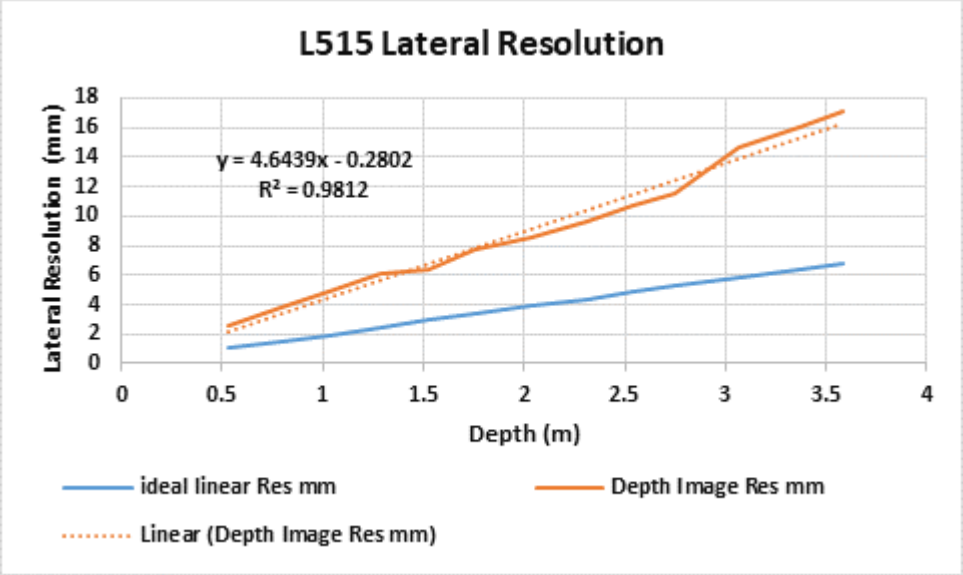


Figure 4.21 Lateral X-Y Resolution of the L515 Sensor with Depth

From this test for the L515 camera, the average lateral resolution is approximately 2.4 pixels or 4.6mm per metre depth and as figure 4.21 above shows, is largely invariant over the depth range tested.

The D435i camera was not tested for lateral resolution in this way as it was clear from earlier testing, including figure 4.18 above, that this device was far less capable and subsequently dropped from further calibration testing and for similar reasons the D455 camera was also not tested further.

4.1.5 Depth Camera and RGB Image Comparison

The depth camera is clearly at a resolution disadvantage relative to the RGB camera, considering the L515 camera with its on-board RGB camera they share a similar Field of View but the colour camera has a horizontal resolution of 1920 pixels compared to 640 pixels in the depth camera. This difference is illustrated on the following side-by-side image comparison shown on figure 4.22. Note that though these images are not exactly aligned, they were taken at the same depth.

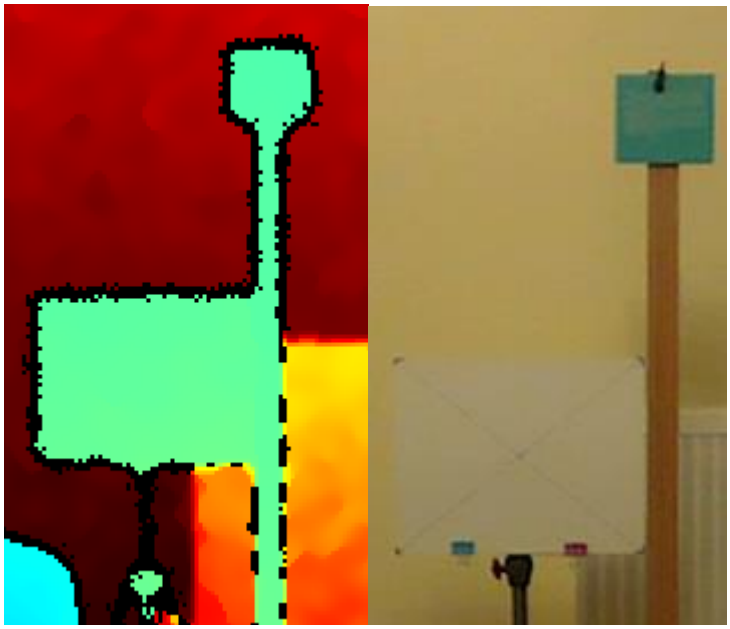


Figure 4.22 Comparison of Zoomed Depth and RGB for the same Region

As can be seen from figure 4.22 a lot of detail is lost in the depth image relative to the RGB camera. The very small gap between the main placard and the small placard support is lost, the two items become one. The small upper placard is also highly distorted at the corners and lower edge.

This observation promotes the idea of using colour image data to enhance the resolution of the depth camera. However, this comes with further image matching complexity in addition to a significant increase in data volume being processed per frame. Since the objective of this project is to perform robot tracking in real-time at 30Hz this option was not investigated further.

4.2 ServoTrack Development and Testing

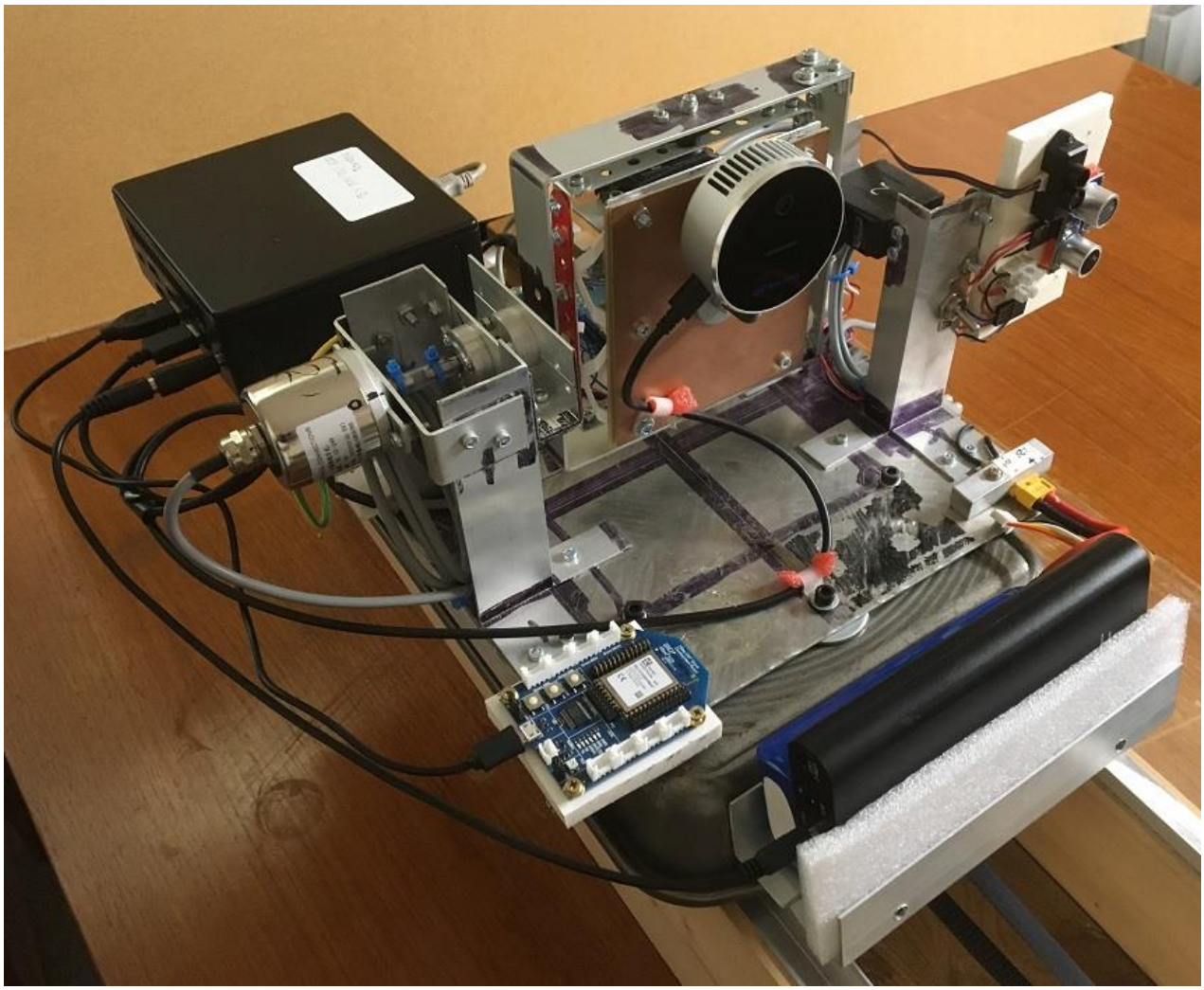


Figure 4.23 ServoTrack Carriage Front View

The ServoTrack test rig, shown in figure 4.23 above, was developed to provide a servo-controlled moving camera base with three axes of control: Pitch, Yaw and translation in the camera Z axis. The system provides motion control of camera and more importantly accurate sensor measurement of the camera position. This data is captured along with the depth camera data at 30Hz and recorded for later post-processing.

The primary motivation for this rig was to aid in the development and validation of the tracking algorithms developed in Chapter 3 above and tested in Chapter 5. This level of motion measurement precision could not be achieved with a freely moving (unconstrained) mobile robot.

Development of the real-time control algorithms to operating this rig at 30Hz was also seen as an interesting area of research within the project and a potential development platform for future autonomous robot designs.

The vision algorithms and the real-time embedded computing methods developed are common to this rig and the ServoTruck test facility also developed within the project and discussed later.

A sketch of the physical layout of the ServoTrack system components is given on the figure 4.24 below with a system schematic given on figure 4.25.

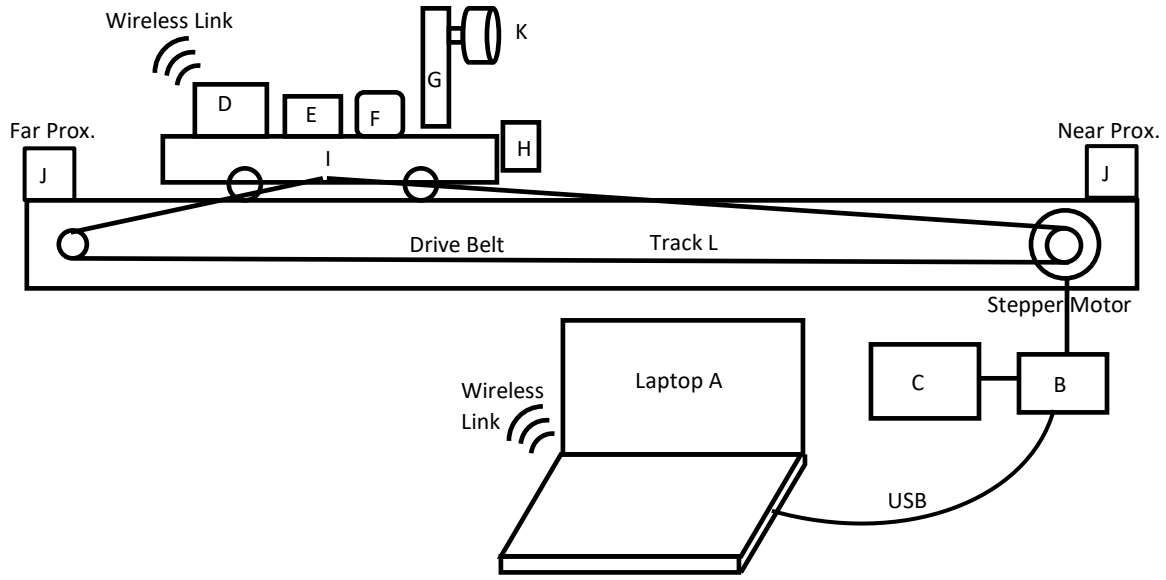


Figure 4.24 ServoTrack Schematic Layout of Hardware Components

Components labelled in the above schematic are listed below.

A. Laptop (provides primary system control interface running TrackHost)
B. Track Electronics Module (with Arduino™ Mega® running TrackDRV)
C. Bench Power Supply (set to output 24VDC)
D. Intel® NUC™ Embedded Computer (running CarHost and CarCam)
E. Arduino® Mega™ (CarDRV)
F. ServoTable Electronics
G. Servo Table and Depth Camera 2-axis Gimbal Mount
H. 12V 5Ah and 19.5V 17.5Ah Battery Packs
I. Carriage Truck
J. Near and Far Track-End (Hall effect) Proximity Sensors
K. Intel® RealSense™ L515 Depth Camera
L. The Track

4.2.1 ServoTrack System Description

Primary user control is via a simple text-based interface (TrackHost) running on the laptop (A). This provides track control via wired USB-2 connection to the Track electronics module (B) running TrackDRV code. The laptop also provides control signals to the carriage computer running CarHost (D) via Wireless (ZigBee) serial link. CarHost also feeds control signals to the Arduino (E) running CarDRV which in turn provides real-time control of the ServoTable via electronics (F) and feedback of sensor data to CarHost.

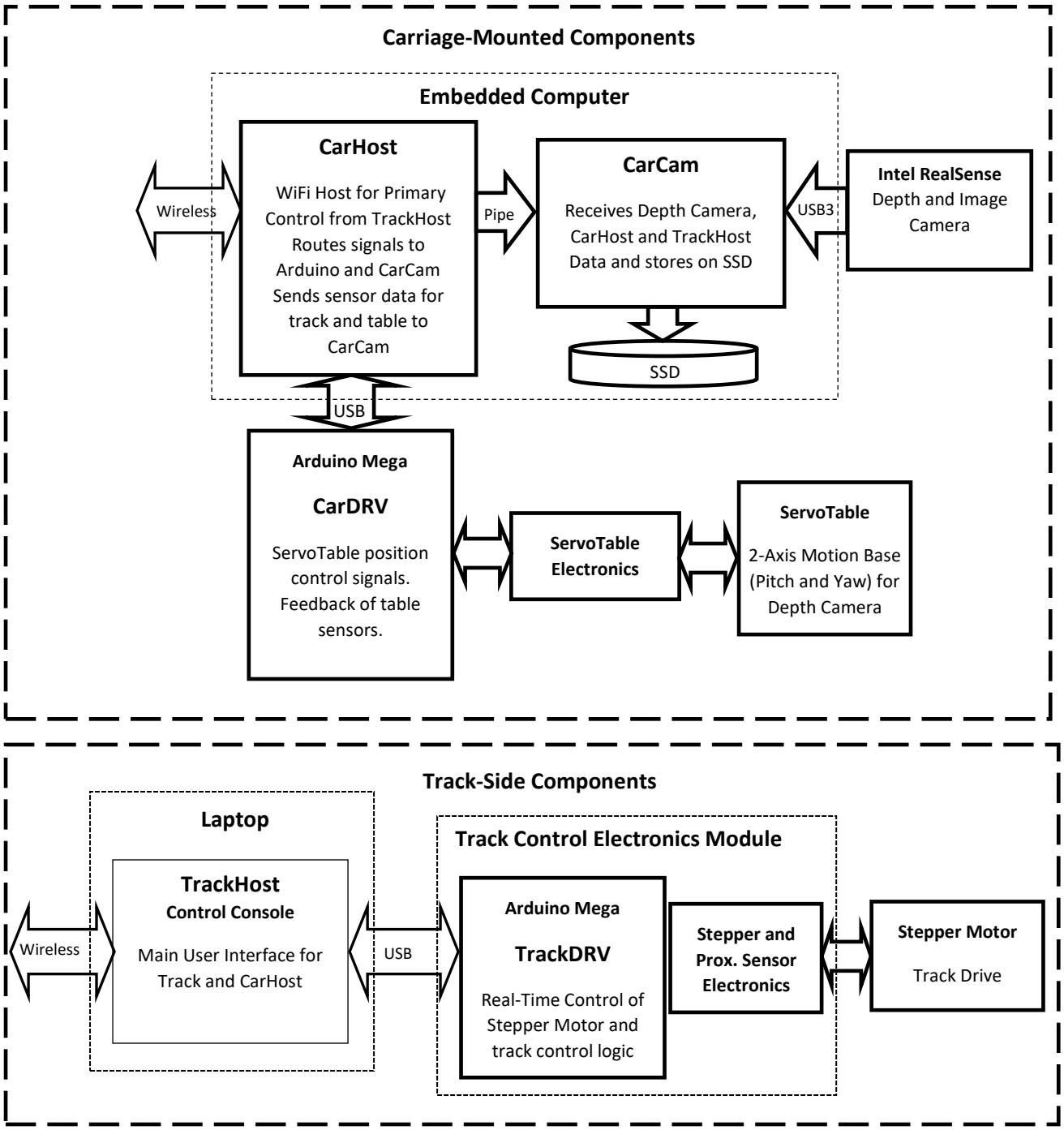


Figure 4.25 ServoTrack System Schematic

CarHost integrates all control and sensor signals (track + camera + table) with the Depth Camera data, recorded using CarCam and stores this data locally to SSD for off-board post-processing. The Track electronics module (component B on figure 4.24 above) is shown on figures 4.26 and 4.27 below.

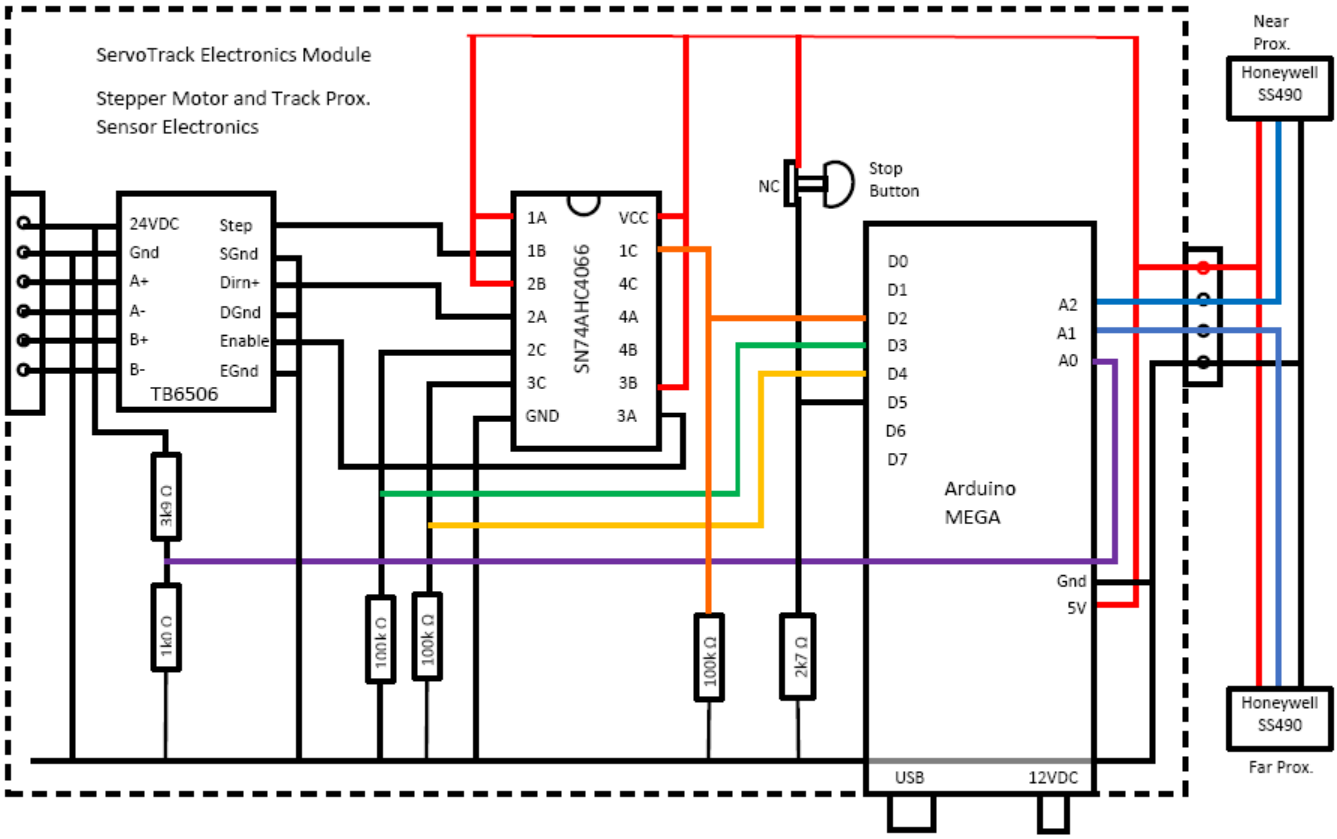


Figure 4.26 Track Electronics Module Schematic

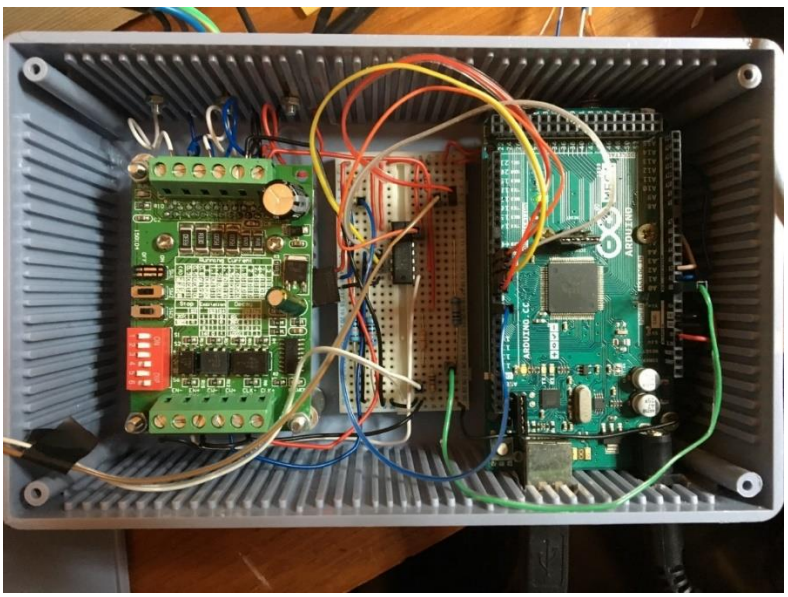


Figure 4.27 Track Electronics Module Internal Details

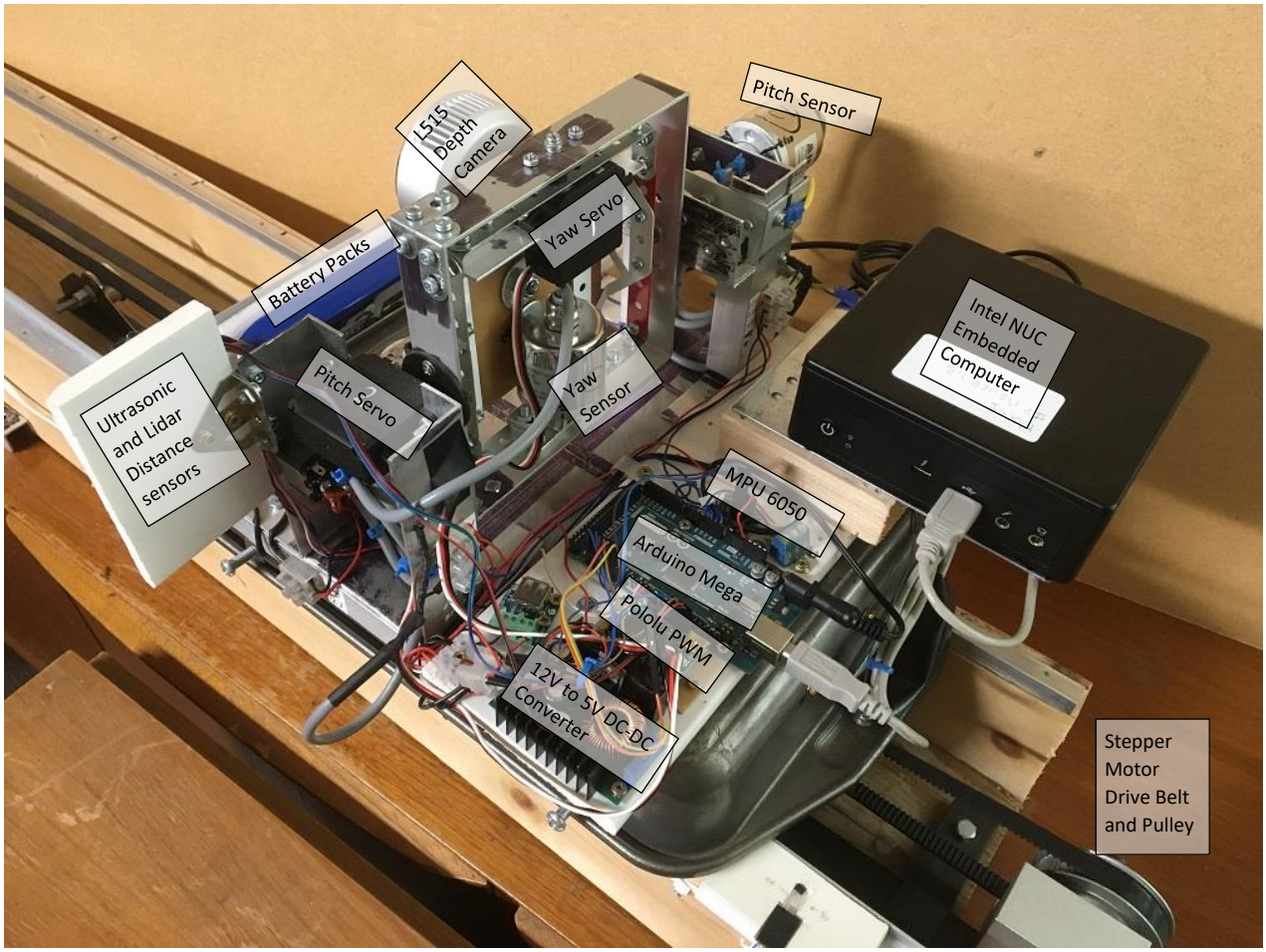


Figure 4.28 ServoTrack Carriage-Mounted Electronics Details

Figure 4.28 above shows a rear view of the ServoTrack carriage highlighting the various electronic and sensor components. The pitch and yaw servos are simple RC aero-model control servos. They are quite powerful but also quite friction bound due to high 150:1 gearing between motor and output shaft. The Miran yaw and pitch position sensors are sufficiently accurate for the project <0.2%FS, and Arduino recording resolution is 10bit, though the analogue measurement electronics within the Arduino were found to be quite noise prone and would have benefitted from an analogue pre-filter circuit. Calibration data for the ServoTrack test rig is given in Appendix B.

Note there are additional ultrasonic and lidar distance measurement sensors mounted on the white board visible on figures 4.23 and 4.28 above. These were ancillary to the main test function (intended to validate position measurements) and not part of the primary test regime and hence not documented further here. A schematic of the carriage electronics is shown on figure 4.29 below.

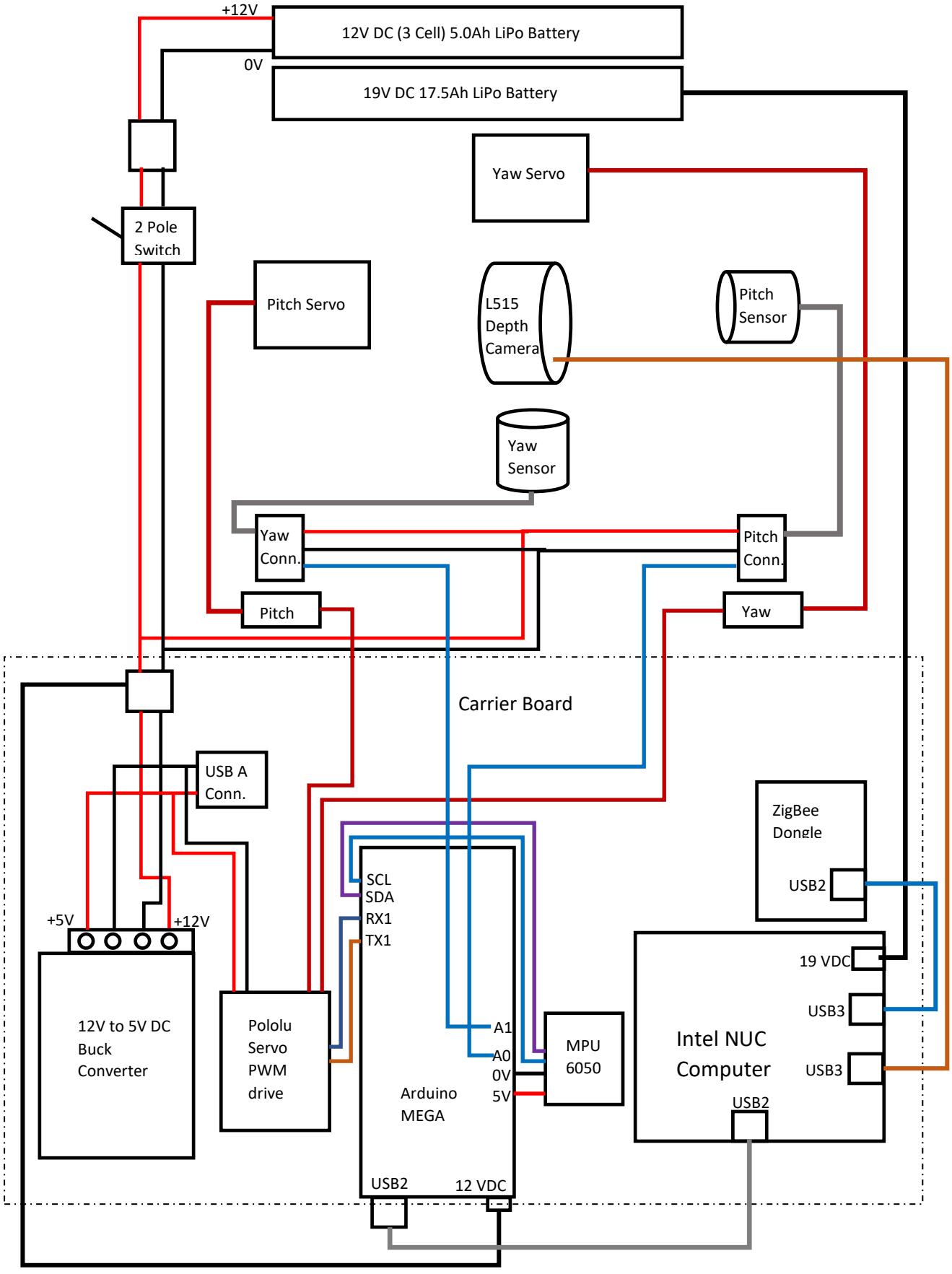


Figure 4.29 ServoTrack Carriage Electronics Schematic

4.2.2 ServoTrack Software Components and Functional Overview

There are 4 computers and 5 software components in the system configuration shown on Figures 4.2.3 to 4.2.7 above. This section defines functionality and communications between the different program components with reference to figure 4.25 above.

Component **Functionality**

TrackHost This component is the principal user-interface to the system controlling both TrackDRV directly via USB and CarHost remotely via Zigbee wireless link. It uses a simple console interface with a limited text-based command system to provide control of system operation. The following commands are supported by both TrackHost and CarHost:

Command	**Description**
init	Initialises the system
mode	mode 1=continuous, 2=step sequence
start	Starts a run and data recording
stop	Stops a run
reset	Performs a reset of the system
end	Ends all programs and closes all recording files
vers	prints out the version data for all software components
para	prints the parametric data for all applicable components

TrackDRV This Arduino program controls the track drive stepper motor providing both simple steps and velocity-profiled trajectories. It also provides control logic for the end-of-track proximity sensors and monitoring of safe state of the track and carriage. On startup the track must be initialised, in this mode the carriage is moved slowly until the far proximity sensor is detected. The system is then available for controlled movement. An emergency stop is also provided on the electronics module.

CarHost This component is the mounted on the carriage and communicates wirelessly via Zigbee for data comms and via WiFi for control functions. It must be up and running before TrackHost can connect and a run started. It can be started via SSH using Wifi in a remote console. CarHost controls CarDRV and CarCam and provides the interface between TrackHost and the other carriage components and is a coordination centre for messaging between all system components. CarHost can be run as a standalone to operate the carriage components independently of the track for testing/debugging purposes.

CarDRV Provides primary control of the Pitch and Yaw servos via the Pololu PWM servo-drive amplifier and output of this and the other onboard sensors including the 9-axis accelerometer/gyro output of the MPU6050. Two distance sensors, one ultrasound and one laser are also mounted on the front of the carriage to provide independent carriage position measurement.

CarCam Under normal operation this program component is automatically started from CarHost using a Linux "fork" operation. CarCam initialises and controls the Intel RealSense depth camera. Depth camera data capture is controlled by the Intel RealSense API. ServoTrack position and CarHost sensor data are recorded along with the corresponding depth data frame to the SSD on the embedded computer.

As discussed in the introduction to this section, the primary purpose of this test rig is validation of robot tracking using the depth camera data and the algorithms detailed in section 3 above. The system is designed to operate in two distinct modes:

Mode 1 is a real-time mode whereby all system components operate and communicate concurrently at a frequency of 30Hz. This frequency is set by the depth camera frame rate of 30FPS. All sensor data and camera image data are then simultaneously recorded frame-by-frame for later off-board post-processing. This mode was challenging to develop requiring that all computers were capable of data throughput at the required 30Hz frequency, whilst minimising latency, jitter and time drift between components. The problem was found to be particularly challenging given the operating system (Linux) is not an optimal choice for real-time application and for this reason mode 2 was implemented first. It also served as a reference for subsequent mode 1 development and testing. The messaging information flow in mode 1 is shown on figure 4.30 below.

Mode 2 is a stepped mode; in this mode a sequenced series of steps are carried out such that each is completed before the next step in the sequence can be started. The advantage of this mode is that each camera image is taken after a motion step has been carried out so the system is stationary at the point of image and sensor data capture. Whilst this mode was more complex to program than mode 1 with a much higher messaging traffic between components, it gives better quality data for image registration due to the minimisation of systematic errors. Since, rather like stop-frame-animation, time is removed from the data acquisition process as are dynamic and time-based errors including motion-blur, sensor-data-latency, jitter and time drift between the system components. The messaging information flow for mode 2 is shown on figure 4.31 below.

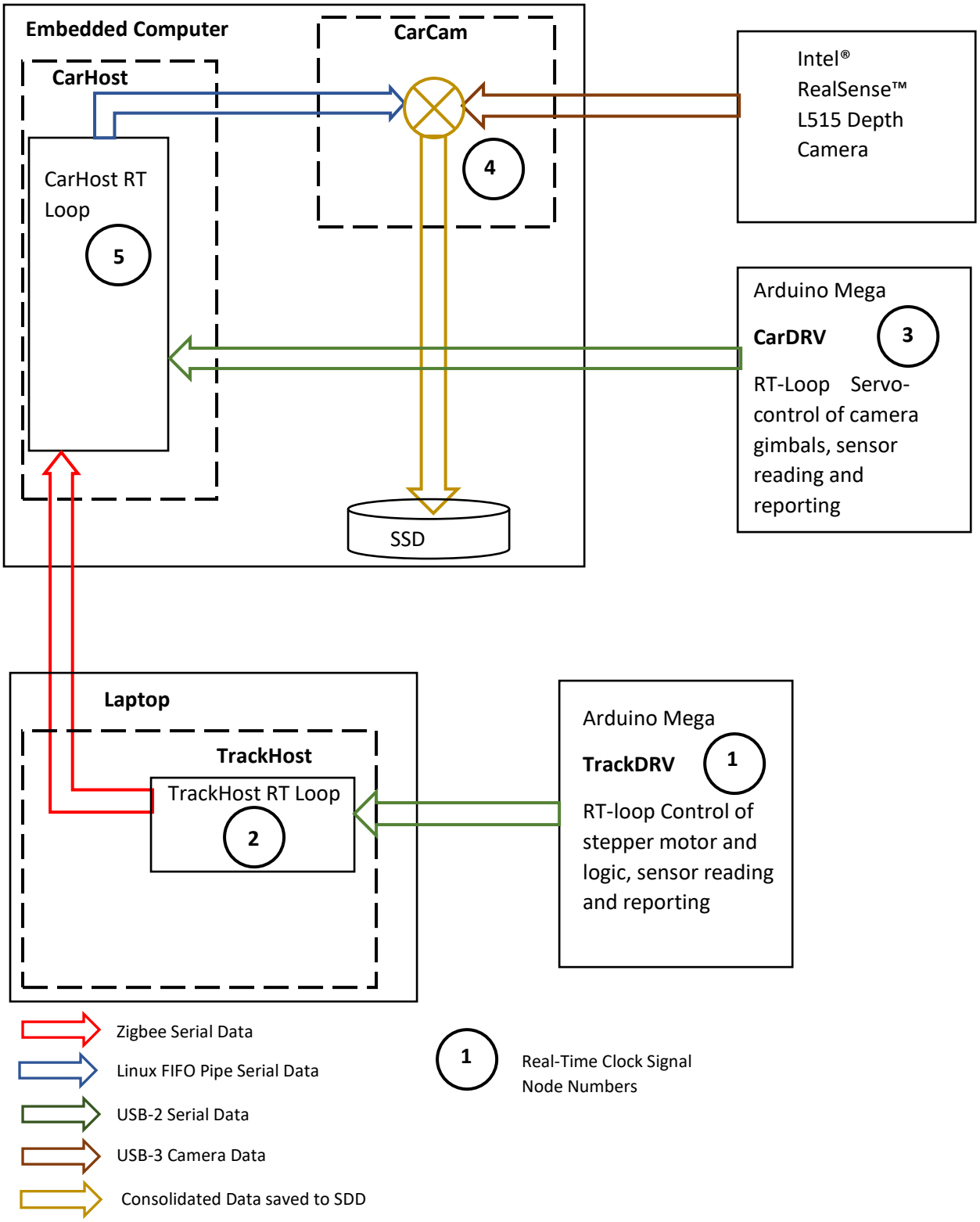


Figure 4.30 ServoTrack Real-Time Data Pathways for Control Mode 1

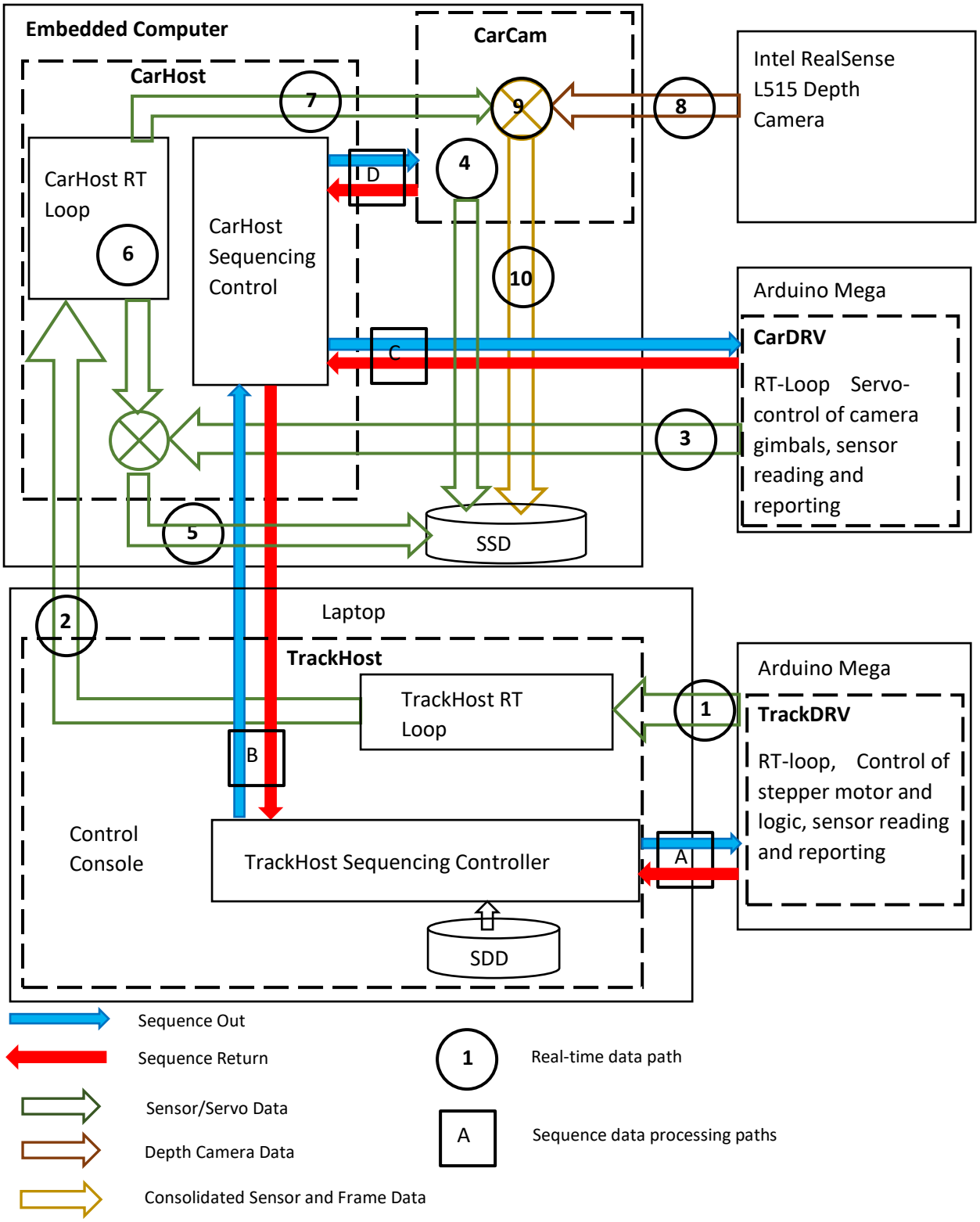


Figure 4.31 ServoTrack Data Pathways for Stepped Control Mode 2

4.2.2.1 Real-Time Control System (Mode 1)

In mode 1 all actions are carried out concurrently in real-time at 30Hz. Data signals at various point around the network shown on figure 4.30 are given below, node numbers are those indicated on figure 4.30. The time signals are also numbered according to the measurement node. These time signals are each maintained by a separate program component and give the time since the start of that component.

Node	Data
1	Time1, TrackPos
2	Time2, Time1, TrackPos
3	Time3, PitchDem, Pitch, YawDem, Yaw, AccX, AccY, AccZ, TFPos
4	Time4, Time5, TrackPos, PitchDem, Pitch, YawDem, Yaw
5	Time5, Time2, TrackPos, Time3, PitchDem, Pitch, YawDem, Yaw, TFPos
7	TrackPos, PitchDem, Pitch, YawDem, Yaw
10	Time4, TrackPos, PitchDem, Pitch, YawDem, Yaw, CameraOn

Sensor data for each main program component including Nodes 2, 4 and 5 are additionally recorded to comma-separated-variable (csv) text files for system performance analysis and debugging during the run. These csv files are quite separate from the combined depth and sensor data binary data file also saved to SSD at node 10.

4.2.2.2 Step Sequence Control System (Mode 2)

The sequencing system forces a sequential step to be read from a script file and directed to the device specified in the step instruction. The action is then carried out to completion by the target device and a return signal is sent back to the sequencer to indicate step completion. The next sequence action cannot be initiated until the previous action has returned a completed message for that globally unique sequence number. This is indicated on figure 4.31 above where blue arrows represents the outgoing steps and red the returned response from the device to show the action is completed.

This system provides a means of building up any programmed sequence of system movements and depth camera frames taken in a step-by-step manner to ensure depth-camera motion is eliminated from the data recording process.

This “stop-frame-animation” approach to data gathering has several other advantages over real-time camera data acquisition.

- Reduces the data accumulation rate (size of the binary file) with each new frame providing fresh data.
- Allows for simulated movement rates far higher than possible with the actual test rig.
- Allows any combination of Pitch, Yaw and Track motion profiles to be tested.
- Removes dynamic vibration and motion-blur effects that may otherwise affect the camera recording process.
- Removes sensor-data-transmission-latency that would lead to systematic errors in the validation process.

Data signals recorded in mode 2 are the same as for mode 1 as shown in sub-section 4.2.2.1 above.

4.2.3 ServoTrack Real-Time Test Results

The results in this section relate to the performance of the test-rig when operating in real-time (mode=1). They measure the ability of the rig to perform the requested actions in real-time. To improve Linux behaviour the Ubuntu Low-Latency Kernel (discussed in Appendix A) is employed along with high priority real-time scheduling SCHED_FIFO (priority=99) process affinity and memory locking are also set on CarHost and CarCam. The mapping between processes and four computer cores on the carriage computer is shown on the figure 4.32 below.

Core 0	**Core 1**	**Core 2**	**Core 3**
Linux OS	CarHost Real-Time Loop	CarCam Camera Capture Loop	CarCam Data Recording Thread

Figure 4.32 ServoTrack Carriage Computer Core to Process Mapping

ServoTrack real-time behaviour is quite complex because it includes 5 components running simultaneously and messaging in real-time at 30Hz.

Thus, overall performance is due to a number of factors including:

1. Latency and jitter in response of the operating system (Ubuntu Linux) to real-time interrupts.
2. Propagation delays due to the transmission media connecting programmes and devices.
3. Synchronisation (or rather lack thereof) between these programs/devices.
4. Loop cycle time within the application layers.

Unfortunately, Ubuntu-Linux does not provide user-space access to hardware interrupts or user-defined Interrupt Service Routines (ISR's) that can be triggered by such hardware mechanisms. The application

programmer is therefore compelled to “busy-loop” for incoming real-time data interrupts. This is achieved by a loop polling mechanism illustrated in the simplified code fragment in figure 4.32.

```
do{
     CurrentTime = Clock.micros();  // time since start of run
     Clock.MuTick()                 // start an independent loop timer
     if(DataPort.available(0)){     // poll for incoming data (without waiting)
          DataPort.read(Data);      // read the data
          ProcessData(Data);        // process data
     }
     usleep(ST);                    // sleep for ST microseconds
     CT = Clock.MuTock();           // CT = Cycle time for loop
} while( RunState!=RUN_END );
```

Figure 4.32 Real-Time Loop Code Fragment

There are a number of features in the above loop that are of interest in understanding Linux real-time behaviour and the following test results.

1. The loop will not do anything (other than loop endlessly) if there is no data on **DataPort**.
2. **CT** measures the loop cycle time achieved for both data and no-data-present conditions.
3. The **usleep** delay prevents Linux from looping at a very high rate and starving all other tasks.
4. **ProcessData** is a gross simplification, it actually represents 3 layers of state machines.
5. Each ServoTrack program component has a very similar loop structure.

Initial testing carried out and reported in Appendix A shows that under ideal, no-load conditions, some of the latency and jitter effects are primarily due to the transmission mechanisms. This proves also to be the case under operational conditions as illustrated in the following time-history plots figures 4.33 to 4.36.

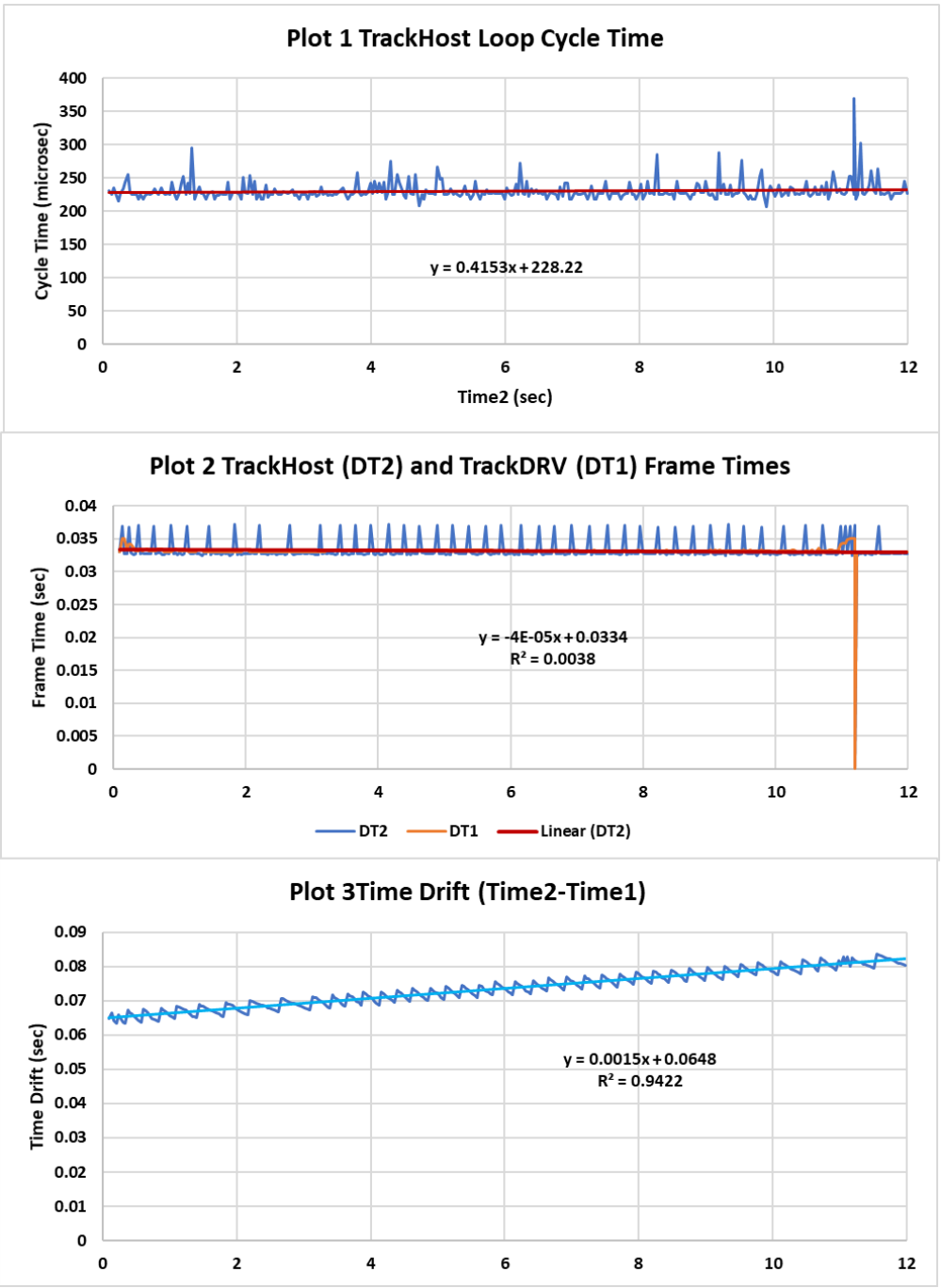


Figure 4.33 ServoTrack Timing Test Results for TrackHost and TrackDRV

Figure 4.33, Plot 1 above shows that the TrackHost loop cycle time has a mean value of 228 microseconds however the usleep delay is 200 microseconds, so the actual loop-processing time is approximately 28 microseconds. Thus, if the usleep (200) call were not present, the loop would actually spin at approximately 35KHz, however this being Linux, everything else would stall.

Figure 4.33, Plot 2 shows the frame rate, this is actually set by the Arduino (TrackDRV) at 33.3 msec, there is some slight variation due to interaction with stepper motor speed control algorithm and hence the

glitch at 11.2 seconds evident at the end of the step. The DT2 response shows that there is a frame-time variation on TrackHost. This is entirely due to the effect of randomly sampling the incoming serial data buffer from TrackDRV. This has transmission delay of between 2 and 4 milliseconds hence the bounded saw-tooth appearance of DT2.

The intercept of figure 4.33, Plot 3 simply shows the difference in the times the clocks were both started and since they cannot be started simultaneously says little else. However, it is also clear that there is an apparent linear time drift between the two clocks Time1 (Arduino) and Time2 (Linux). The code is organised such that TrackDRV (Arduino) is setting the frame rate for both it and TrackHost, there is then a drift buildup in the time events are recorded at TrackHost. These event recordings occur every time TrackHost intercepts an incoming serial clock pulse from TrackDRV. As shown on plot 2 these events vary depending on how much data got into the serial buffer at the time DataPort.available() triggered. This time drift is believed to be due to the gradual accumulation of these variable delays and represents 1.5 milliseconds of error per second of actual real time.

This gradual apparent time drift between processors is neither real nor critical since there are no temporal components or sensor data affected by it. It is retained nonetheless as it is a useful indicator of the health or otherwise of the various communication pathways between computers.

What is critically important is that the propagation delay between a position sample being taken at the track sensor (TrackPos) and it's recording along with the camera data at CarCam. Since, if there are gross time delays, this will lead to systematic errors in the data recording process and invalidation of the test rig as a means of verifying the vision algorithms. Since this was the primary justification and purpose of the ServoTrack test rig, this is the most critical aspect of real-time performance required of the rig. Unfortunately, this propagation delay is almost impossible to accurately measure, though based on evidence it is possible to infer it to a reasonable degree.

Figure 4.34 below shows timing results collected at CarHost during this same test.

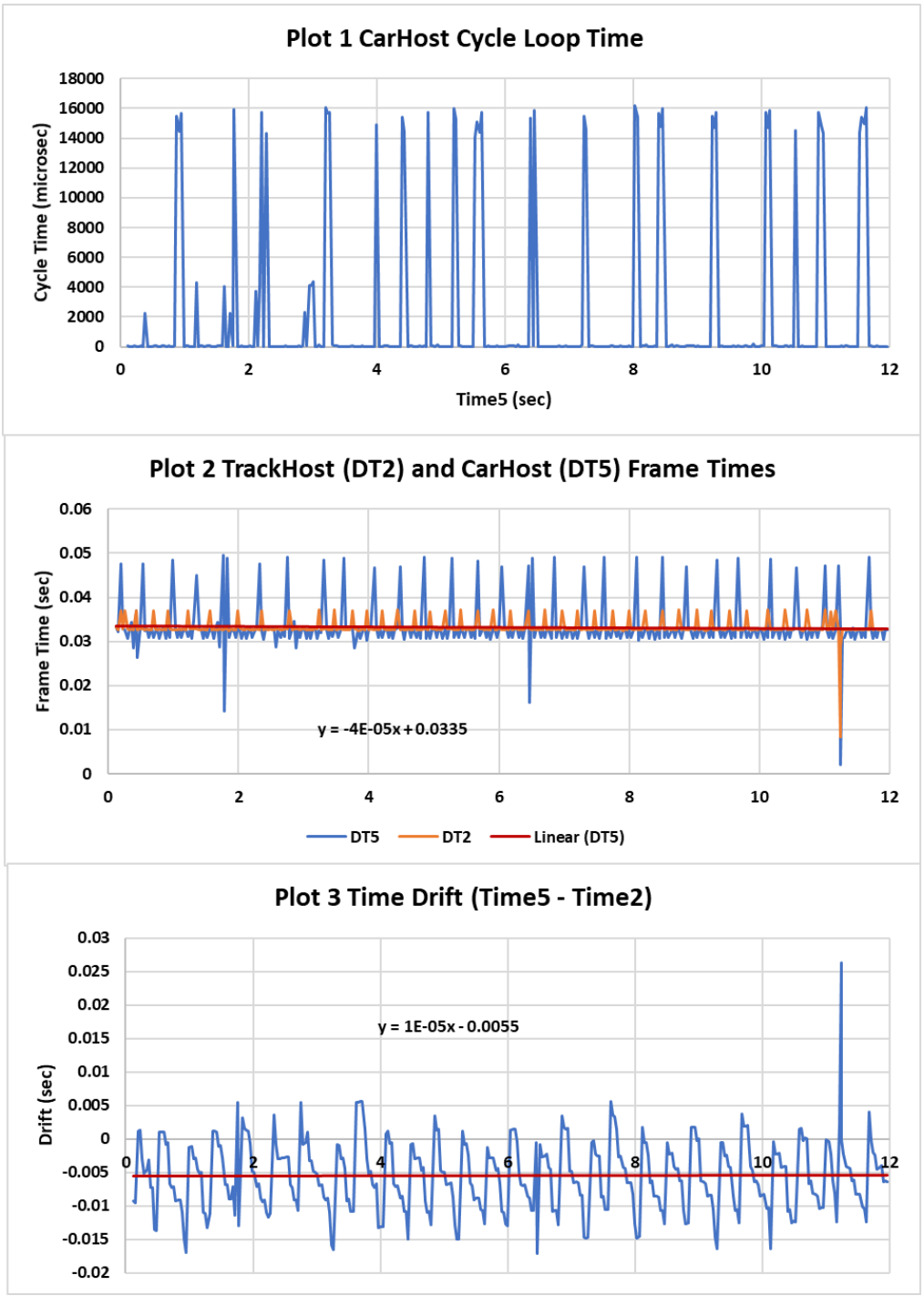


Figure 4.34 Timing Test Results at CarHost over ZigBee

Plot 1 of figure 4.34 above shows that CarHost is considerably more taxed than TrackHost, with cycle-time spikes up to 16 milliseconds. Communication between TrackHost and CarHost is via ZigBee serial wireless interfaces. These devices are not optimal for this role with a transmission delay of around 15 milliseconds. However, they proved to be far more reliable than WiFi with no large variations in transmission times seen during testing with WiFi.

When connected via ZigBee TrackHost position data is communicated over the link and acts as the frame update driver in the same way that TrackDRV provided the time signal to TrackHost.

The large oscillations on DT5 (shown on plot 2 of figure 4.34) bear witness to the variation in serial buffer delay due to the ZigBee transmissions. As can be seen these variations are much larger with spikes up to the ZigBee transmission delay (15 milliseconds). However, the mean frame time is still close to the target at 33.5 milliseconds.

Plot 3 of figure 4.34 above shows that whilst there are large dynamic variations in drift the actual long-term drift is negligible. The large spike at the end of this plot is due to the discontinuity originating in the TrackDRV stepper logic at the end of the 1m step.

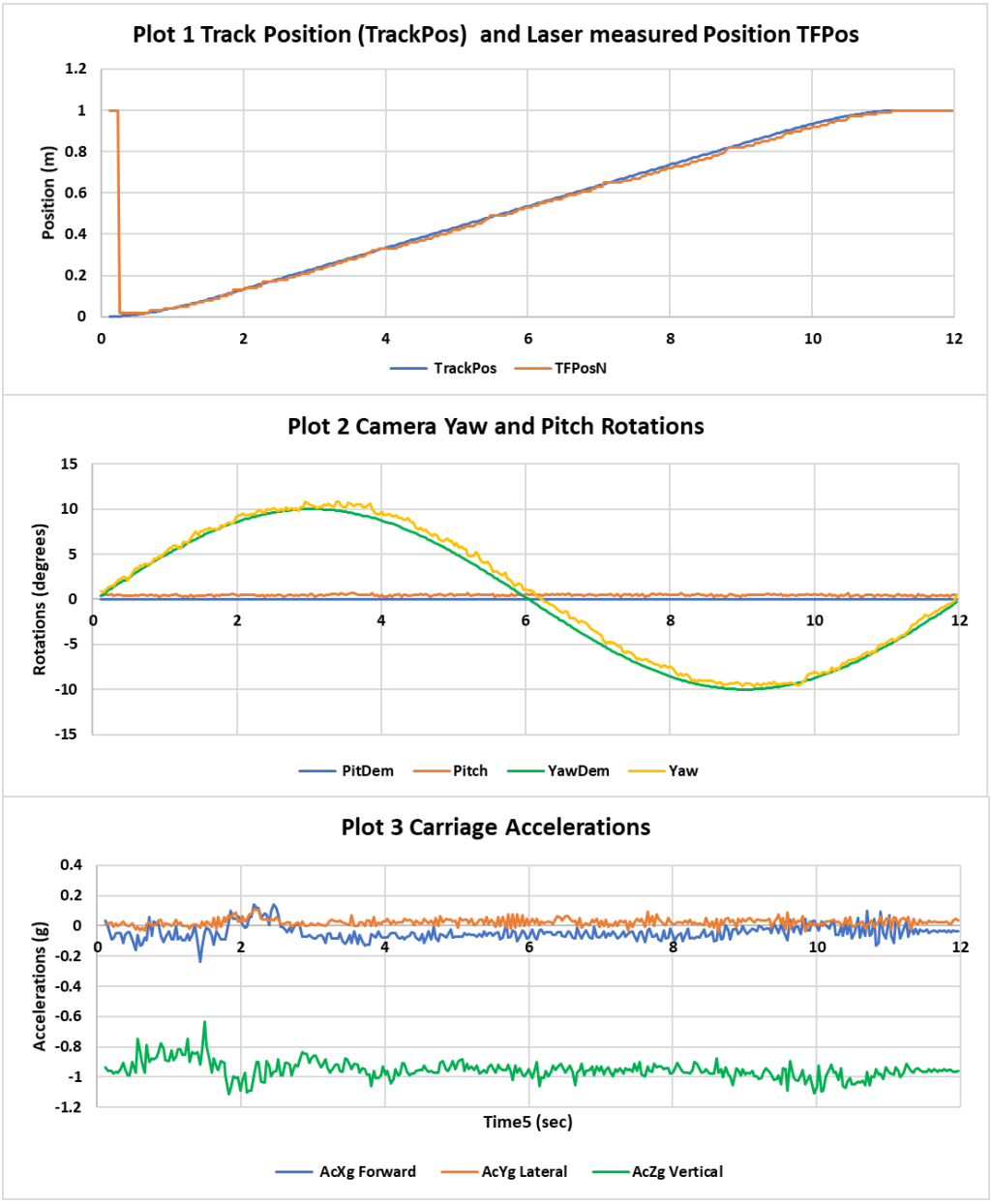


Figure 4.35 ServoTrack Sensor Outputs Seen at CarHost

Plot 1 of Figure 4.35 above shows independent measurements of Carriage Position. TrackPos is taken directly from the stepper motor system and is accurate to within 1mm. TFPos is taken from a time-of-flight laser position sensor mounted on the carriage and read by CarDRV. These two independent measurements, taken by different means and provided by different communication pathways give some confidence of the correct operation of the ServoTrack real-time systems.

The laser was statically calibrated and normalised to provide an accurate measurement relative to the stepper motor system. Any significant dynamic lags would therefore show up as a discrepancy between these two position results. As can be seen there is a small error but it is within acceptable limits for the intended use.

The carriage was stimulated with a sinusoidal yaw signal whilst traversing the track. As can be seen there is an appreciable difference between the demand (YawDem) and the actual servo position (Yaw). This is due to friction in the servo however, only sensor positions (Yaw and Pitch) are used in the validation process and these signals have been calibrated as described in Appendix B.

The carriage also shows some evidence of longitudinal and vertical accelerations due to the stepper motor and belt-drive system. The carriage has grown in weight relative to the original design, mainly due to the addition of the heavier NUC computer and the additional 19volt battery needed to supply it.

The following plot figure 4.36 shows results recorded at CarCam for this same test.

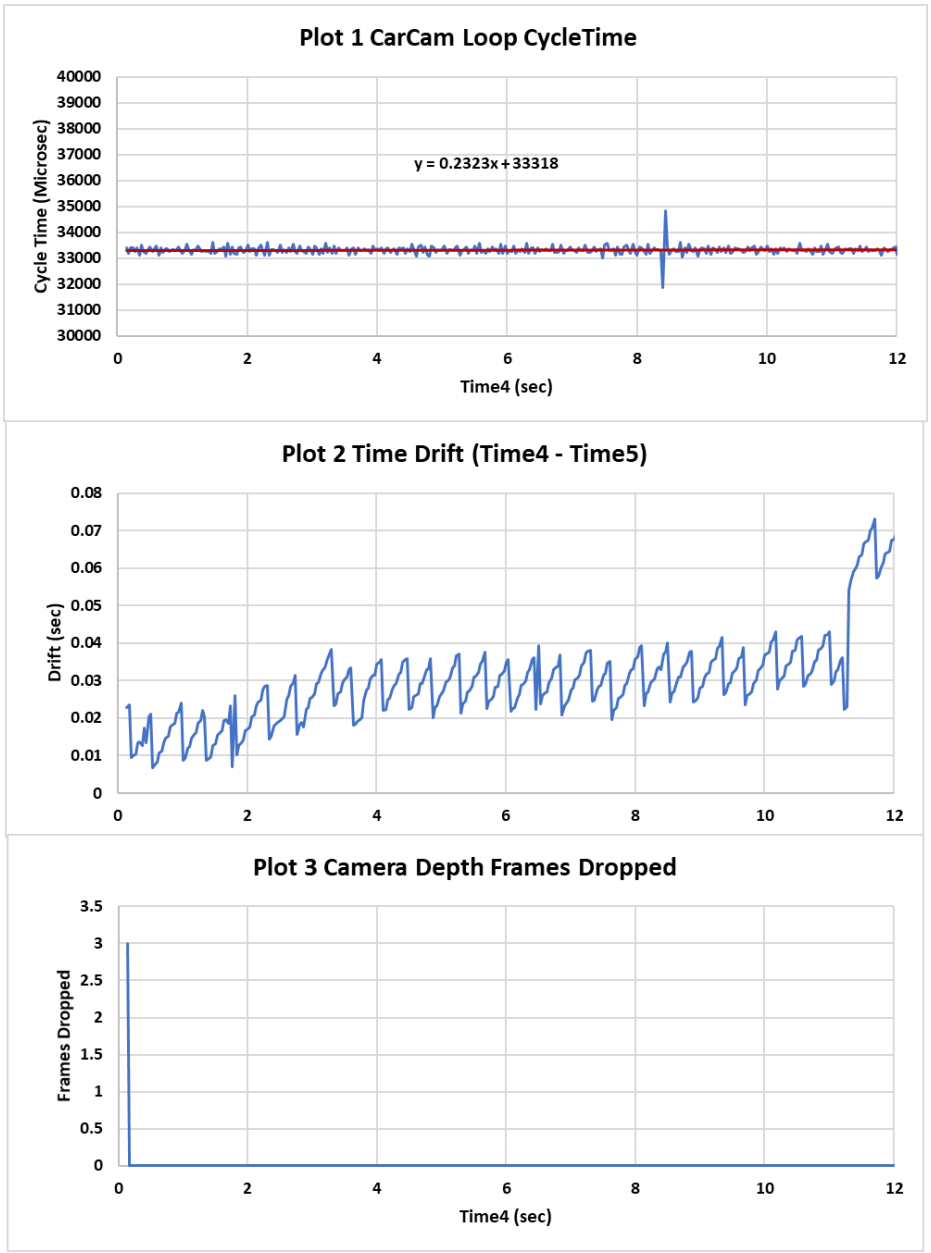


Figure 4.36 Timing Test Results taken at CarCam

Figure 4.36 above shows results taken and recorded at CarCam. In this case the loop is controlled by the Depth Camera API so cycle time and frame time are exactly the same thing. Hence, they are not provided as separate plots as for TrackHost and CarHost.

As can be seen with the exception of a minor glitch at 8.4 seconds (reason unknown, but it is not considered a problem) the camera API keeps very good time and is completely independent of the other ServoTrack timing mechanisms.

Plot 2 of figure 4.36 shows that drift is dynamically varying at approximately the same frequency and amplitude as for CarHost due to Zigbee messaging latency. The glitch originating from the TrackDRV is also in evidence at the end of the run. This glitch only happens at the end of a step and has no impact on

data recording. What these results show is that the system is basically health and functioning as expected.

This fact is confirmed by plot 3 of figure 4.36 which is a count of the frames dropped during the recording process. It took quite a lot of time and effort to achieve this rather boring function validation plot.

CarCam is the end of the line as here (or rather in a separate CPU core/thread) the sensor data is recorded to a binary file together with the depth camera images captured on a frame-by-frame basis for use in later off-line processing and vision algorithm are validation, this is covered in Chapter 5.

4.2.4 ServoTrack Step-Sequence Test Results

Owing to the many issues with developing the real-time system discussed above and in Appendix A, an alternative system based on stepped response (mode 2) and illustrated on figure 4.31 above was developed. Many development tests were carried out, on the mode 2 system, but only a few showing salient features of this stepped mode are shown here.

In the first test the carriage is stepped in 20mm increments along the track with a pause at each step to allow a camera frame to be taken. The time response showing track position and an independent laser distance sensor TFPos (mounted on the side of the carriage with data normalised to match the TrackPos sign and origin) is shown on figure 4.37 below.

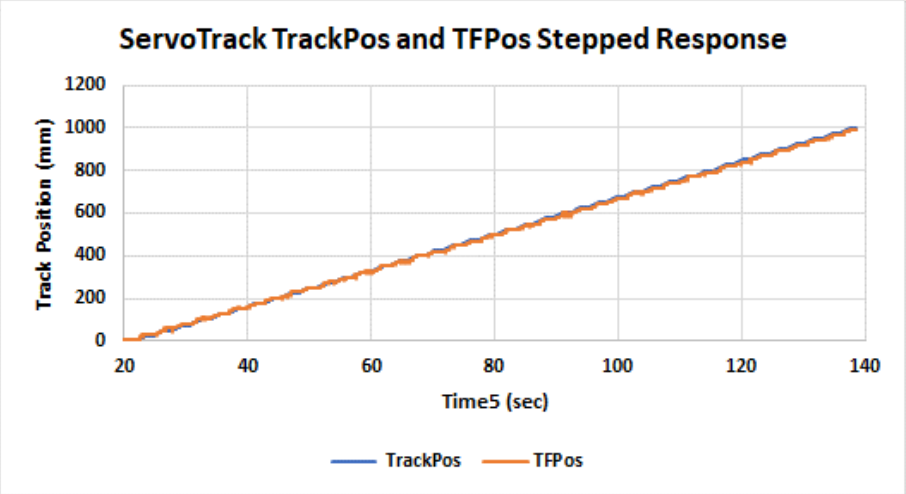


Figure 4.37 Showing XTest2_01 Stepped Position along the Track

This shows that the system performs as expected but some of the detail is lost in the scale so the following figure 4.38 below shows the same result zoomed over a 40 second period.

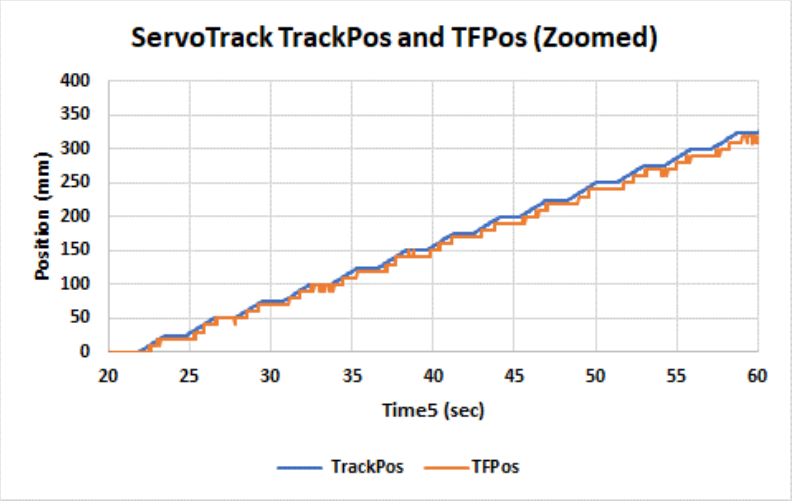


Figure 4.38 Showing XTest2_01 Zoomed

The following further zoomed plots on figure 4.39 show carriage accelerations, TrackPos and the CameraOn signal.

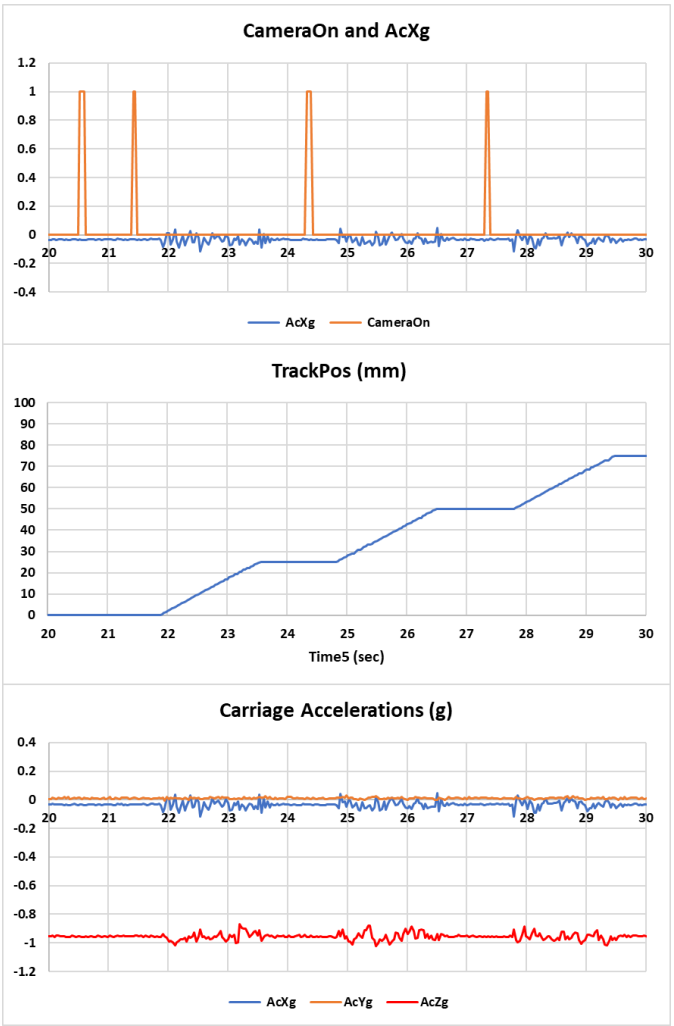


Figure 4.39 Showing XTest2_01 TrackPos, Accelerations and CameraOn

As shown on figure 4.39 above, the sequencing of steps is programmed such that the carriage is quiescent at the point when the depth image is taken (CameraOn=1).

This completes the initial testing of the ServoTrack test rig, the purpose being to verify functionally of system operating in both modes. Whilst some performance concerns exist and particularly the 15 milliseconds latency of the Zigbee communications in mode 1, the rig is considered fit for purpose in that it is meeting the 30Hz frame rate without dropping frames when operating in real-time.

4.3 ServoTruck Development and Testing

4.3.1 System Overview

The truck running gear is based on a high-end, off-the-shelf Radio-Controlled car model. The original brushless 3-phase drive motor (without commutation) was found to be very difficult to control and was far too fast for the intended use. The original motor and electronic drive amplifier were therefore replaced by a 3-phase brushless motor with Hall Effect commutation and an off-the-shelf DC motor control amplifier. The combination was successfully re-engineered into the truck drive-train and provides smooth speed control of the truck over the required speed range. A picture of the assembled truck is shown in figure 4.41 below.

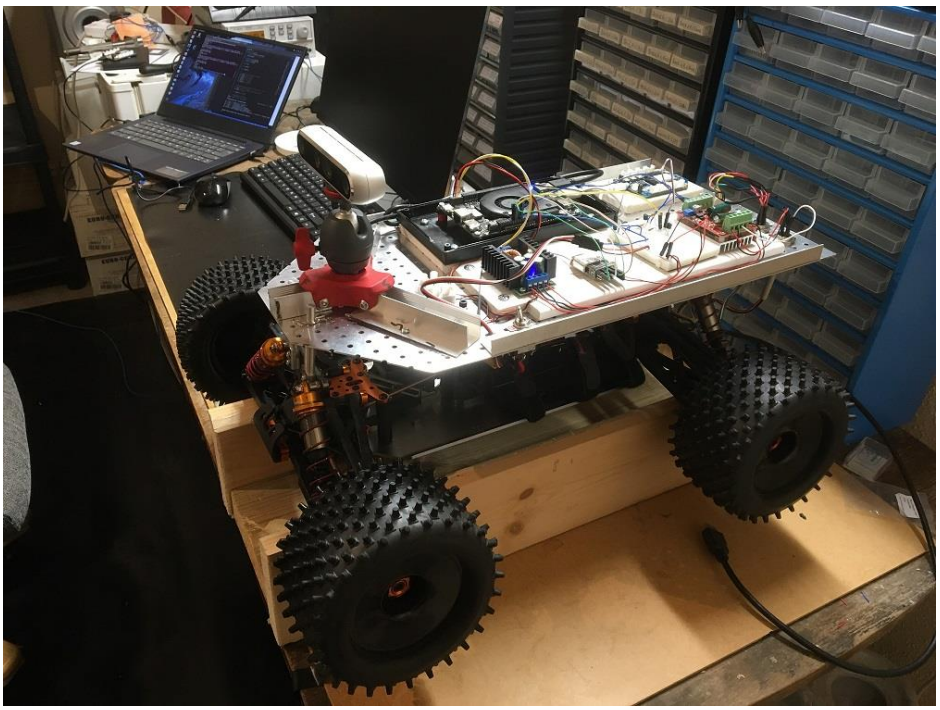


Figure 4.41 ServoTruck

The design of this system is much simpler than ServoTrack and is primarily controlled via a standard radio-control module typically used to control model aeroplanes. ServoTruck is operated like any other radio-controlled vehicle. WiFi serial communications provides Linux headless SSH functionality to start and stop

the recording program TruckCam. The primary speed and direction control of the vehicle is via a RadioLink TF8B control box, this is used for starting, stopping, recording, directional and speed control of the truck. A schematic of the system is shown on figure 4.42 below.

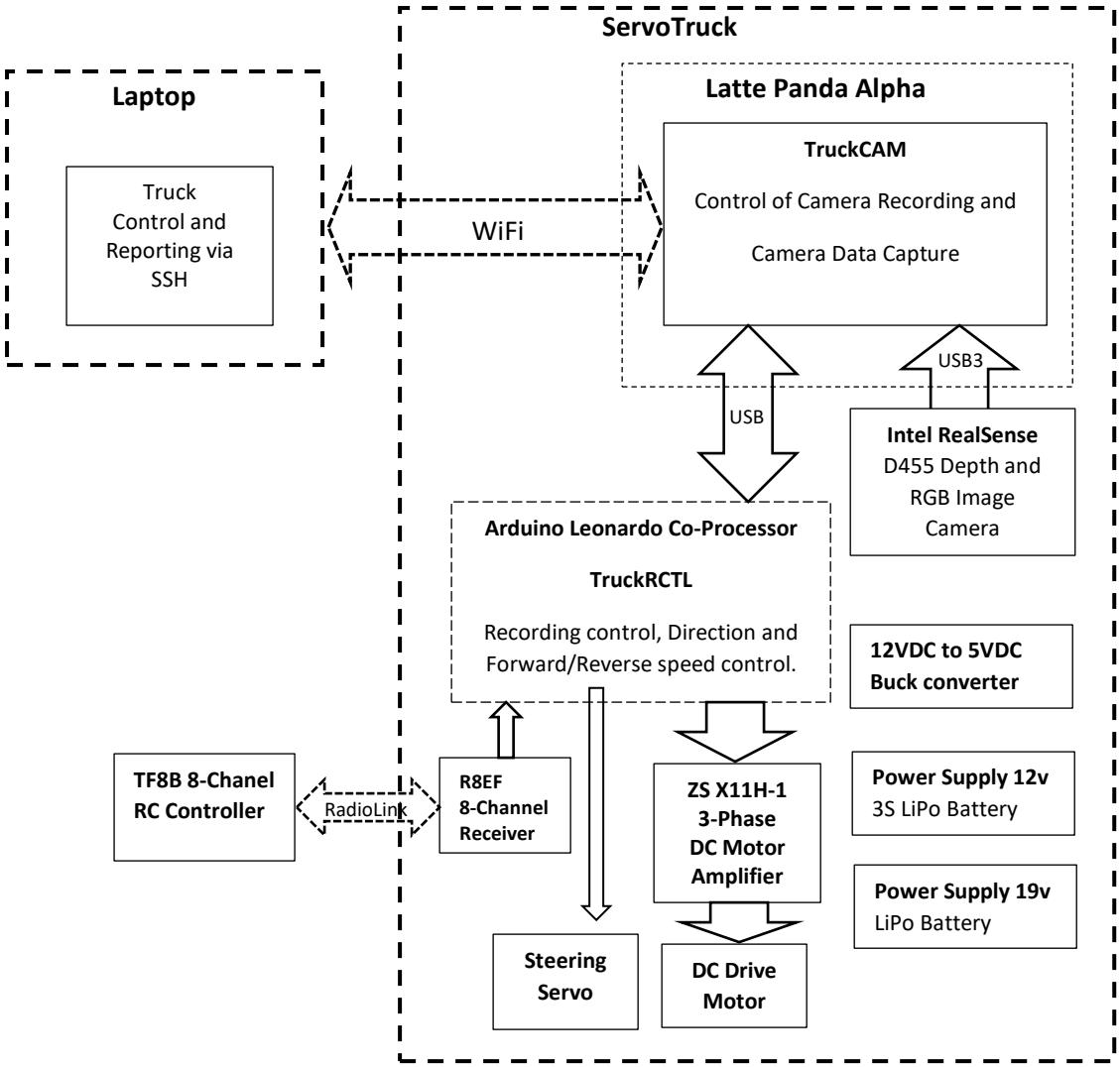


Figure 4.42 Schematic Diagram of ServoTruck Hardware and Software Components

ServoTruck Capabilities

1.1) Capable of manual operation via RC controller.

1.2) SSH (WiFi) Connection to Laptop for TruckCam start/stop and data recovery via NFS

1.3) Continuously record depth camera data at 30Hz for subsequent recovery and post processing.

Constraints and Assumptions

2.1) Operation will be indoors.

2.2) Operation will be on ostensibly flat surfaces.

2.3) The truck will have a maximum forward and reverse speed of 1.0m/s.

2.4) Truck will not exhibit shocks or vibration modes likely to impact recording accuracy.

The radio-control module is shown on Figure 4.43 below with the lower and upper-level electronics of the truck shown on figures 4.44 and 4.45 respectively.

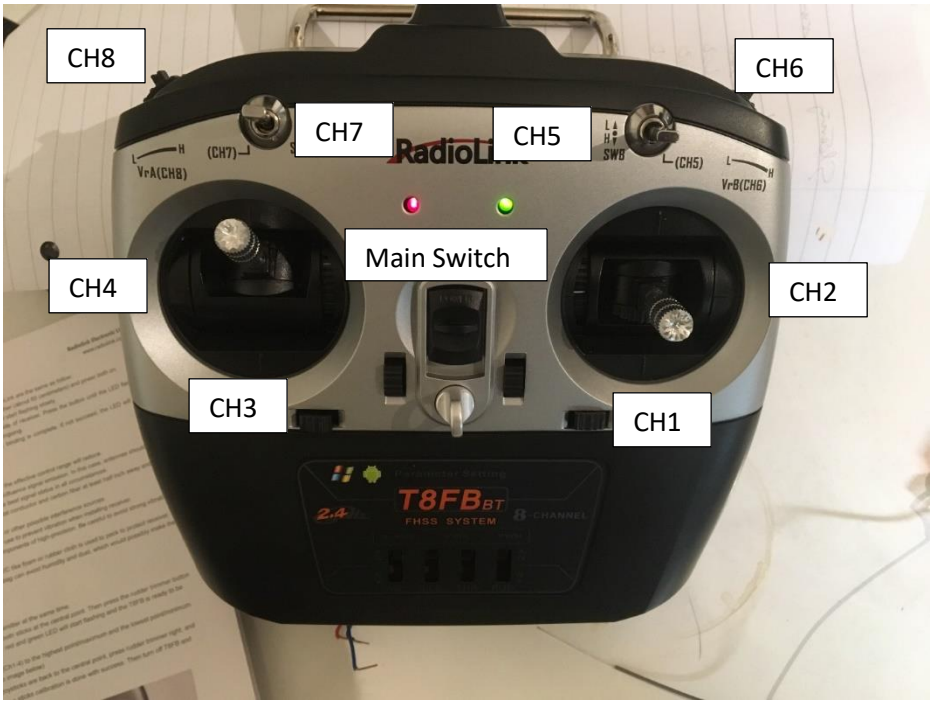


Figure 4.43 T8FB Radio Control Unit (RC Channels as Indicated)

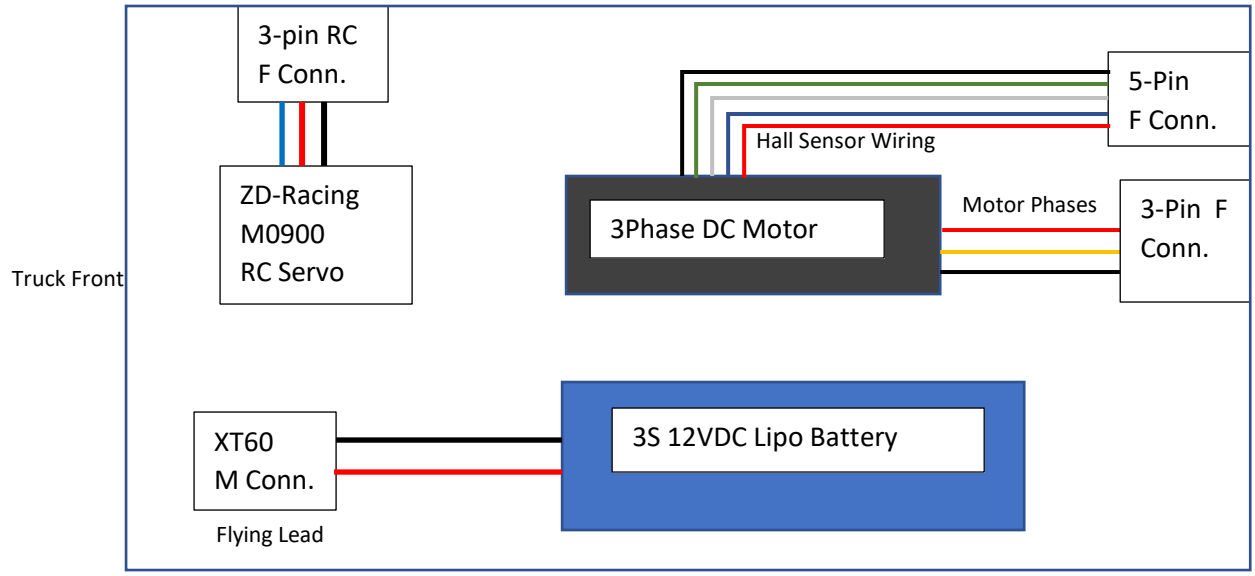


Figure 4.44 ServoTruck Electronic Schematic (Lower-Level)

Figure 4.44 above shows the lower-level electronics schematic. Figure 4.45 below shows a schematic of the upper electronics tray highlighting the various components. Figure 4.46 shows a picture of the upper-level electronics.

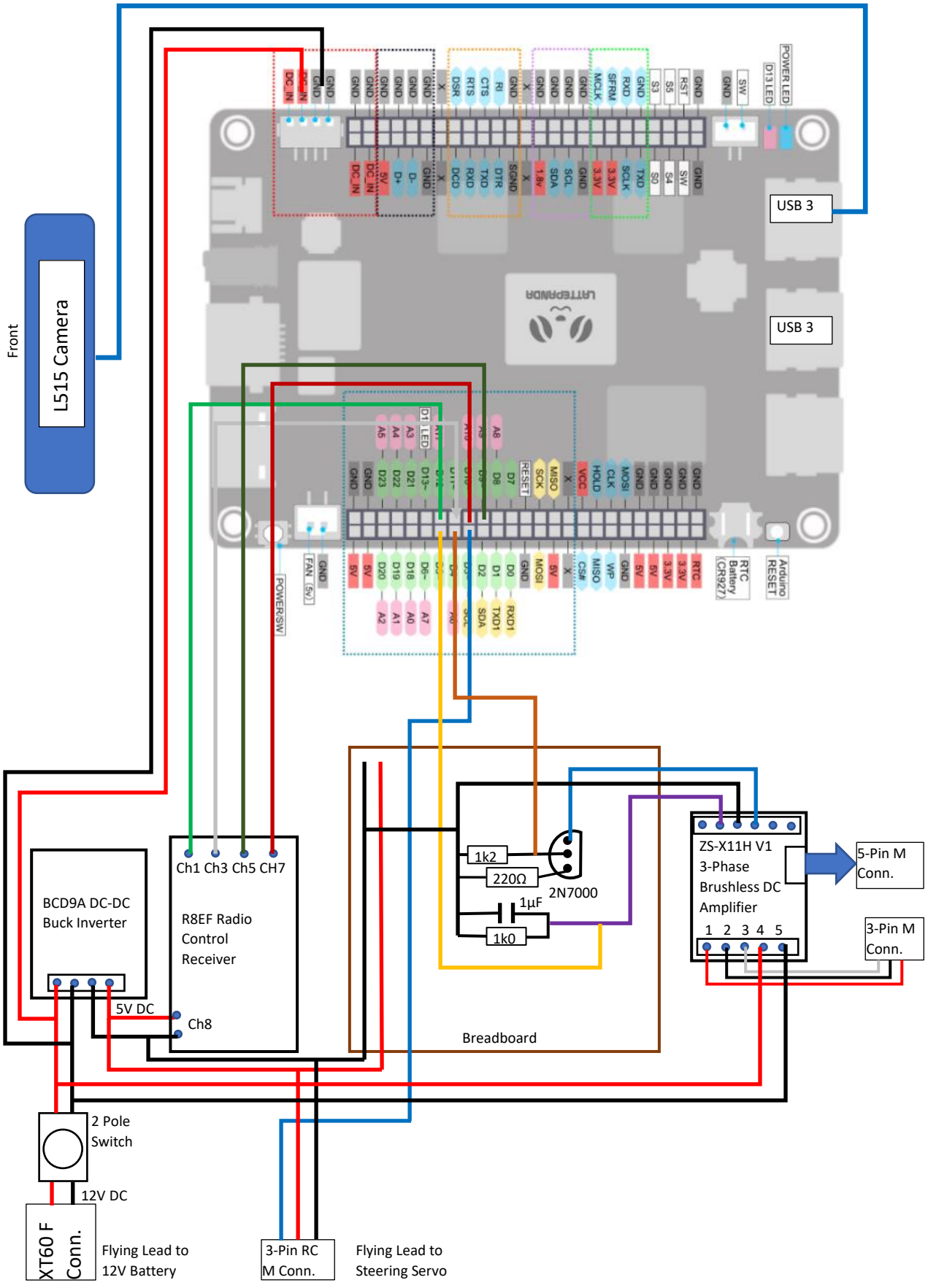


Figure 4.45 ServoTruck Electronics Schematic (Upper-Level)

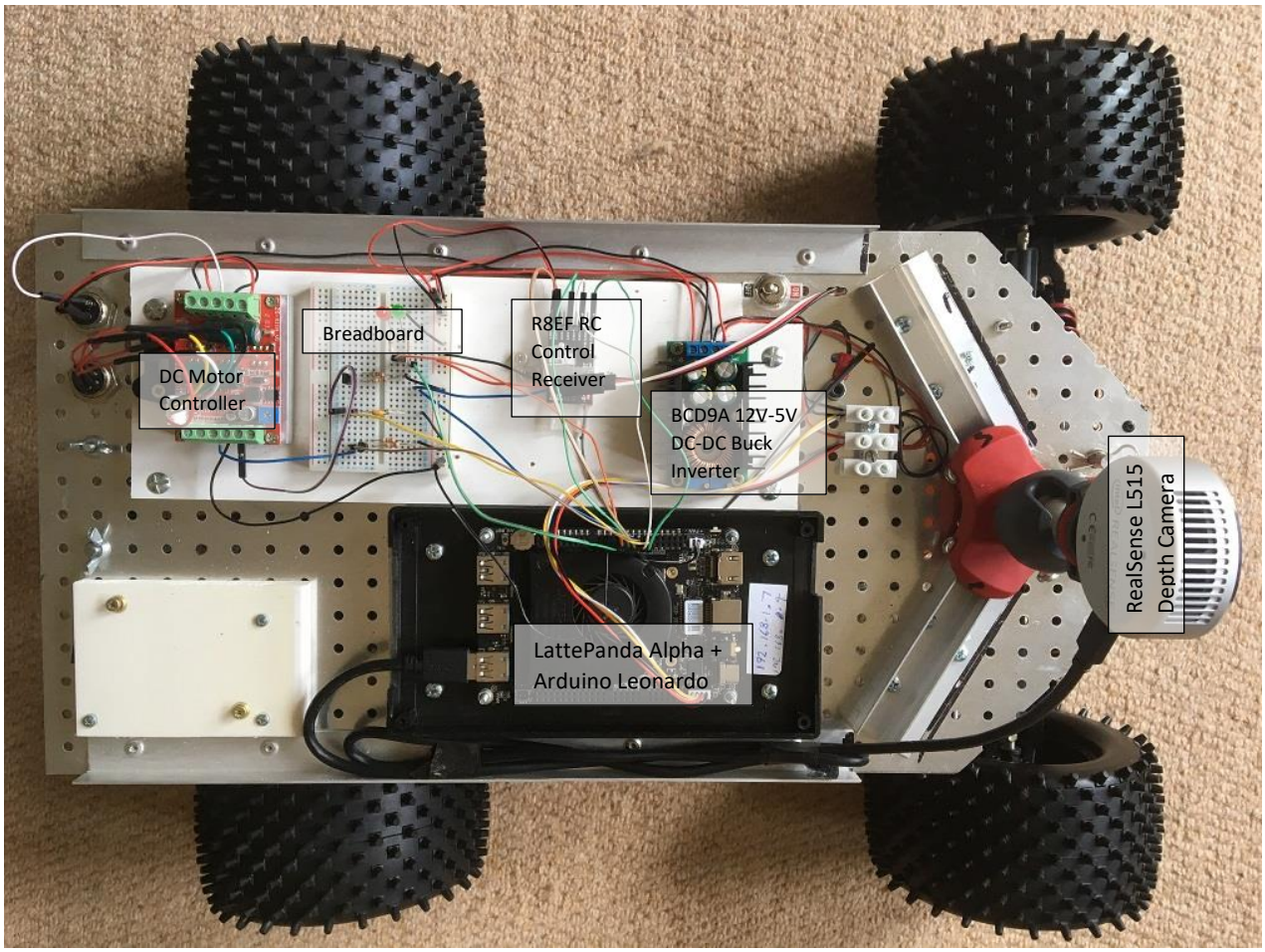


Figure 4.46 ServoTruck Upper-Level Electronics Layout

4.3.2 System Components Functional Breakdown

LattePanda Alpha

This component runs the TruckCam software on Ubuntu Linux 20.04 LTS. It is designed to run as a "headless" computer system with no direct user control and also provides communication with:

- The RealSense D455 Depth camera for recording and data storage on SSD
- The on-board Arduino Leonardo Co-Processor for speed and direction control.
- The TruckSide Laptop via WiFi using SSH

Arduino Leonardo Co-Processor

This component is mounted within the Latte-Panda Electronics and controls of ServoTruck via the TruckDRV program and performs the following tasks:

Interfacing and controlling the DC motor speed and direction with hardwired connections between digital and analogue pins via the breadboard and 3-Phase Amplifier as shown on the circuit diagram (figure 4.45 above).

This component also interfaces with the 2.4GHz R8EF RC receiver for control inputs:

- Steering control via CH2 of the T8FB RC Controller
- Truck Speed and Direction via CH3 of the T8FB
- Truck Start and Stop commands via CH5 of T8FB
- Start/Stop Depth Camera Recording via CH7

Laptop

All of the control functionality for ServoTruck (speed, direction and recording) is controlled directly via the T8FB RC controller input with the on-board computer running TruckCam in “headless” configuration.

The laptop is used to connect to the LattePanda and start TruckCam remotely via SSH. This pathway is used to monitor system operation and as a “health-check” during the code development phase. It is also used to recover recorded data files via the Linux Network File System (NFS) and shutdown the LattePanda when testing is complete.

The Arduino running TruckRCTL acts as a feedthrough for the RC control signals from the T8FB and records these signals along with a time signature to a text “.csv” file for health monitoring.

Further Notes

This initial development step is intended to provide a teleoperated system as a reduced-risk hardware development and as a means of recording live camera data at 30Hz for later post-processing and evaluation of the robot visual tracking algorithms. This is carried out in chapter 5 of this report.

It is not considered practical to send camera data over WiFi in real-time due to the amount of data generated by the camera (18.4Mb/sec) and bandwidth and jitter limitations of the WiFi system. The use of RC control gear provides low latency control signals that would not be possible with WiFi.

In a follow-on phase the Latte Panda computer could be replaced by (or augmented with) a AAEON Boxer 8250 embedded computer. This device comes with a NVIDIA Jetson Xavier NX GPU capability. This allows for on-board processing of the depth camera data and a fully autonomous robot able to navigate in the environment using the vision processing capability of the Jetson.

There are no position or orientation sensors on board ServoTruck and hence tracking and 3D modelling is carried out using the robot tracking algorithms developed in chapter 3 and tested later in chapter 5.

5. Tracking and 3D Model Algorithm Testing

In this chapter the algorithms developed in chapters 2 and 3 will be tested using the ServoTrack and ServoTruck test rigs developed in chapter 4 and further documented in appendix A. The objectives of this work are two-fold, firstly the construction of an accurate 3D model of the environment visited by the depth camera during the test and secondly the computation of an accurate track through that environment.

These objectives are of course tightly coupled since if results from the tracking algorithms are poor, the placement of the images within the 3D model will also not be accurate and this will result in highly visible extrusion distortions of the world within the 3D model. Thus, the degree of model extrusion is one of the methods of accuracy assessment used within in this section.

In addition, where (ServoTrack) sensor-based tracking data is available, this is directly compared to the algorithm-computed track and thus a more direct assessment of tracking accuracy can be obtained. For the ServoTruck test rig there are no independent tracking sensors and only camera data is recorded so the quality of the 3D model is the primary means of assessing computed tracking accuracy in this case.

The Depth camera used for this testing was the Intel RealSense L515 camera discussed in chapter 4. For both ServoTrack and ServoTruck the data was recorded on the test rig computers and then downloaded and post processed offline. Post processing was performed on a HP OMEN games Laptop with:

- Intel i7-8750H 12-core CPU
- NVIDIA GeForce RTX-2070 GPU.

The laptop was running Ubuntu 18.04.6 LTS and all code compiled using the NVIDIA compiler: nvcc version 10.2 which also invoked gcc 7.5.

The offline analysis program VisionOffV5 (shown in outline on figure 5.2 below) is used to test all of the tracking algorithms developed in Chapter 3. This program is controlled by the file: VisionOffV5Control.txt which sets the configuration for the run and allows for parameter variations. This file is quite large, as a result of many development iterations, consequently only extracts of direct relevance to this chapter, with typical parameter settings are shown on figure 5.1 below. Comment lines in the file, indicated with a '#' as the first printable character, these guide the user as to the parameter entries on the following non-commented line, with only one non-comment line, before the next '#' commented line(s).

Primary outputs used to assess the accuracy of the algorithms are screen captures of the 3D model generally at the end of a run and a *.csv text file containing computed and measured tracking data which is then processed to compute the errors between the actual sensor tracking data and the tracking

compute by the algorithms. This operation only applies to ServoTrack runs where rig sensor data is available.

```
#
# VisionOffV5Control.txt
#
# ImageTest      controls the type of Image rendered and registration
# 1=2D | 2=3D+Sensor| 3=PCA Testing | 4=Open Loop ICP| 5=P2L_ICP+Loop Closure
  4
# RegMethod controls the registration used for ImageTest=4
#   1=P2P_ICP,  2=P2L_ICP, 3=Cell_Search
   2
#
# Binary Data Input File
   ../Data/L515/XTest1_05_Chair.bin
#
# Tracking Results file
   ../Results/XTest1_05_Track.csv
#
# CellX Cellular reduction size, allowable sizes are: 32, 16, 10, 8, 5, 4
# KNPoints = 9 for P2P_ICP and 18 for P2L_ICP & Cell_Search (ImageTest=4,5)
# ICPFrames = number of frames between Reference vector refresh
# PreFilter 1  = Bilateral,  2 = Crop, 0 = no pre-filtering
#
#   CellX     KNPoints  ICPFrames  PreFilter
     5            18            1              0
#
# Point-pair matching criteria
# DistMin = minimum distance threshold for a point-pair match (in metres)
# CosMin = minimum value of Cos(Theta) in a dot-product vector alignment test
#               for example CosMin = cos(15) = 0.966, cos(8.1) = 0.99, cos(5.1) = 0.996
# FMThresh Fostner-Moonen covariance matching Threshold (RegMethod=3)
#   DistMin    CosMin     FMThresh
     0.05         0.99          1.0
#
#  Bilateral Filter parameters
#   Ksz     Spatial_Const     Depth_Const
    31          15.0               200.0
END
```

Figure 5.1 VisionOffV5 Control File Extracts

Parameters defined in the comment section of the above file refer to the algorithm parameters shown on figure 5.2 below.

An outline of the principal algorithm components of off-line VisionOffV5 test program is shown on figure 5.2 below.

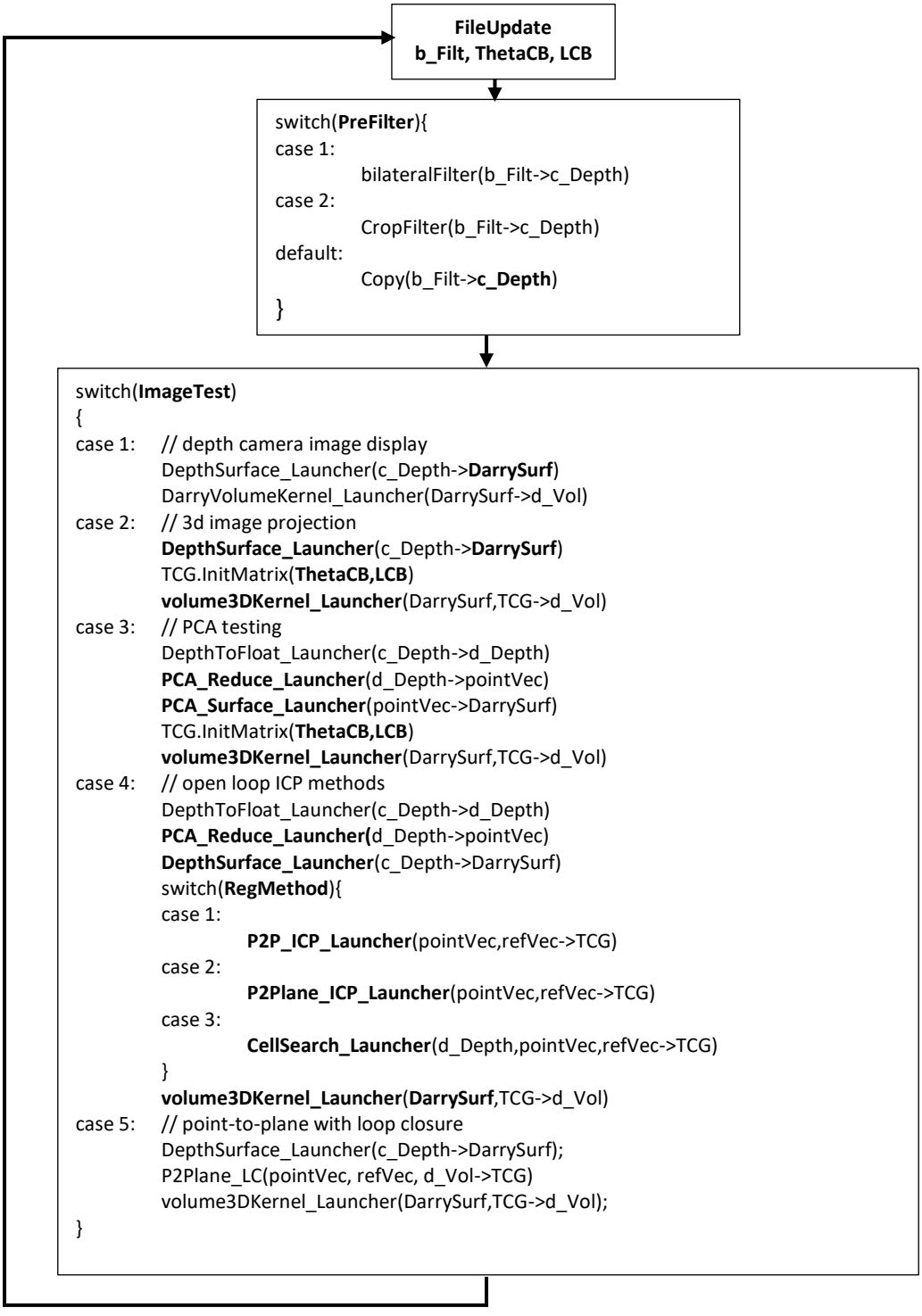


Figure 5.2 Overview of VisionOffV5 Post-Processing Code

The diagram in figure 5.2 above gives a high-level overview of **VisionOffV5** used for development and testing of all main tracking algorithm components.

To aid reading, key variables and functions are highlighted in **bold** and discussed in the following text. The input and output from each function are indicated in the function calls using the convention (input-> output). Functions used in the above diagram call GPU kernels to be launched and typically end with the name "**_Launcher"**.

Each **ImageTest** switch-case branch contains a separate test scenario. These will be discussed and test results presented in the following sections. The loop shown above proceeds using pre-recorded depth camera data on a frame-by-frame basis.

FileUpdate reads a binary file previously recorded using the ServoTrack and/or ServoTruck test rigs discussed in Chapter 4. The binary file contains depth camera image frames in data **b_Filt** and in the case of ServoTrack, sensor data: **ThetaCB** (the camera rotation angles) and **LCB** (the rig displacements) these separate signals are recorded synchronously, frame-by-frame with the camera data.

Pre-Filtering of camera data is controlled by the user parameter **PreFilter**, this is typically set to the default copy option in which case the output is the unfiltered **c_Depth**. If the bilateral filter is selected (**PreFilter**=1) the filter parameters set in the configuration file are used. **PreFilter**=2 was used for debugging only.

The **ImageTest** options are as follows:

ImageTest=1 projects the camera depth image as recorded without further processing (other than coding depth with a false colour to produce a colour bitmap image).

ImageTest=2 transforms the depth camera image data into 3D vertex data using the projection equations defined in section 3.2 (equation 3.3) as a surface in **DepthSurface_Launcher** and projects this surface into the 3D model volume via **volume3DKernel_Launcher**. The camera position data contained in **ThetaCB** and **LCB** are combined in the homogeneous transformation matrix **TCG**, this locates the surface in the 3D model.

ImageTest=3 is a test case designed to validate the working of the PCA reduction process carried out in **PCA_Reduce_Launcher**. This PCA-reduced data is then used to reconstruct a surface model projected into **DarrySurf** by **PCA_Surface_Launcher** and injected into the 3D model volume **d_Vol** in **volume3DKernel_Launcher**. This is intended to demonstrate that the original depth camera surface can be accurately reconstructed from the much smaller quantity of PCA reduced data.

ImageTest=4 investigates three different tracking methods developed in section 3 of the report based on frame-to-frame image registration according to the setting of **RegMethod**. All three methods have the same pre-processing. **DepthToFloat_Launcher** converts from camera depth units (a 2-byte unsigned int) to a float depth in metres. **PCA_ReduceLauncher** performs a PCA reduction of the segmented camera data. The **P2P_ICP_Laucher** and **P2Plane_ICP_Launcher** perform point-to-point and point-to-plane registration as discussed in sections 2.3, 2.5, 3.5 and 3.6 above. **CellSearch_Launcher** performs an additional registration step by matching the covariance and Eigen-structure between cells in subsequent frames based on the methods of Förstner and Moonen discussed in sections 2.9 and 3.12 above. This is a further thresholding on the point-pair matching process and excludes any pair-matches where the FM metric exceeds **FMThresh** set in the configuration file see figure 5.1 above.

ImageTest=5 performs point-to-plane ICP with a loop-closure on the model volume, the first three steps of which are identical to the combination of **ImageTest=4** and **RegMethod=2.** In a subsequent step a camera image **v_Depth** is recovered from the model volume contained in **d_Vol** at the location contained in **TCG**.

The VisionOff5V program produces a number of different outputs including a video recording of the data built in the model volume, frame-by-frame screen images some of which are presented below and tracking data obtained from the ServoTrack test rig and the Tracking algorithms. This enables validation of the tracking algorithms against the ServoTrack test-rig recorded sensor data. Algorithm computed tracking data for ServoTruck is available and presented as screen image captures but there are no equivalent sensor outputs for validation of this test rig. Tracking quality assessment in this case is based on 3D model imaging alone.

5.1 Image Capture and 3D Model Projection

The figure 5.3 below shows a colour image of a beach ball used in the following test scenario.

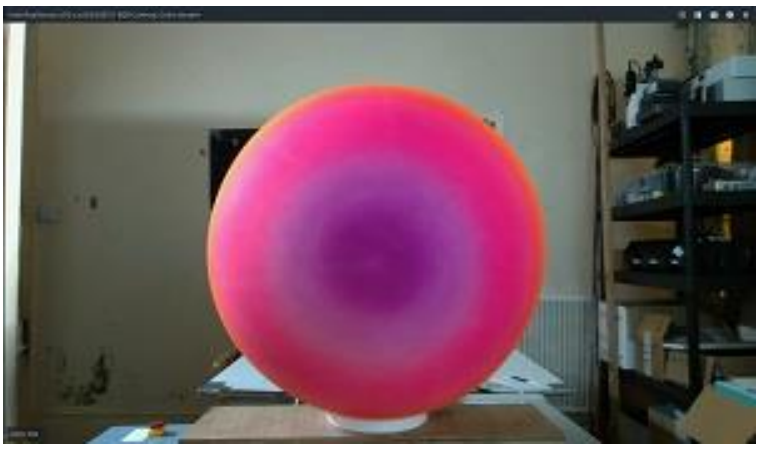


Figure 5.3 Beach Ball Colour Image

Figure 5.4 below shows a depth camera image of the above Beach Ball for **ImageTest=1**. This [640x480] bitmap image is presented as captured by the L515 depth camera with no further processing other than the use of false colour to indicate depth. The colour palette used is discussed below.

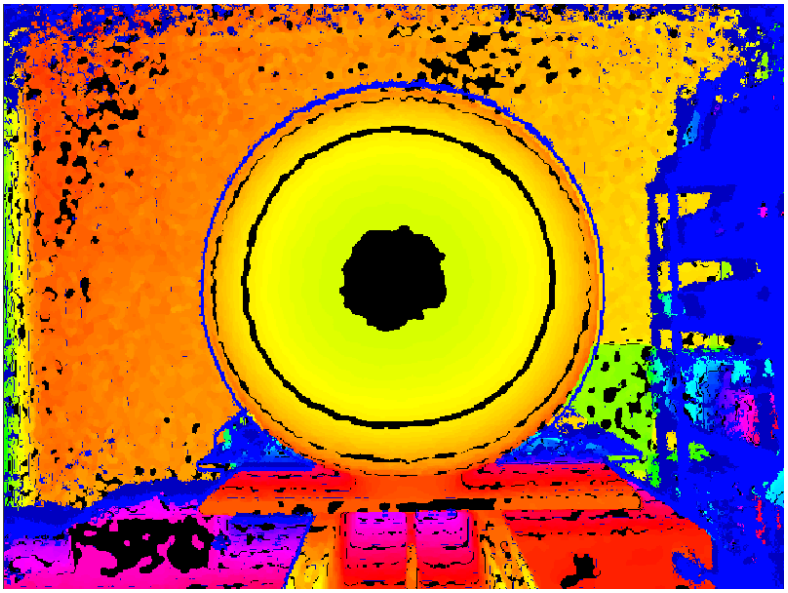


Figure 5.4 Depth Camera Image of Ball, ImageTest=1

With **ImageTest=2** the depth camera image is converted into 3D Vertex data in **DarrySurf** by **DepthSurface_Launcher**. The Transformation **TCG** is set by the ServoTrack rig sensor data and the 3D image is placed in the volume model **d_vol** by **volume3DKernel_Launcher.** The following figure 5.5 shows the projected 3D model image of the same beach-ball.

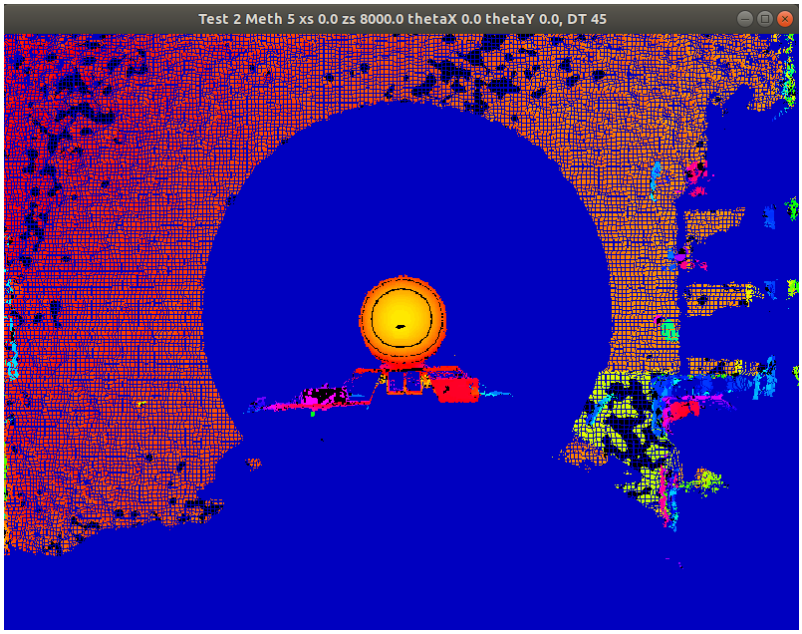


Figure 5.5 Projected 3D Model Image of the Beach Ball

The ball image looks much smaller in figure 5.5 in comparison to figure 5.4 due to 3D projection and its position relative to the 3D model viewport which is displaced outside and in front of the model volume. The dark blue parts of the image represent regions where the camera failed to capture any data and hence remain unpopulated in the model. This is either because they were out of the field of view (most of the ground plane) or in the shadow of the ball (hence the large circular area missing from the back wall). Some regions were sufficiently dark so as not to reflect the laser (for example the shelving to the right of the image). Though it is the most accurate of the devices tested, the L515 camera, along with many Lidar devices, does not respond well to black or other nonreflective surfaces.

Note that there is also a lattice effect evident on the back wall. The 3D model is composed of 5mm cube elements (known as voxels – as in volume pixels). The lattice effect is due to the fact that the camera lateral pixel resolution at 3.2m (the distance to the back wall) is poorer than the 5mm cubes used in the model and as a result some voxels are not hit and hence do not get coloured. Figure 5.5 shows the first depth frame captured and as more images are captured and fused into the model this lattice effect tends to blend away due to camera movement and sensor noise effects.

The 3D volume is used to build a model of the environment by integration (fusing) of depth camera images and represents a physical room-sized volume 4m wide, 3m high and 4m deep consisting of 5mm voxel cube elements. This data is stored as a 3D matrix of float values requiring 1.5936Gb or 20% of GPU memory. Clearly this nonparametric method of model representation requires significant memory resources.

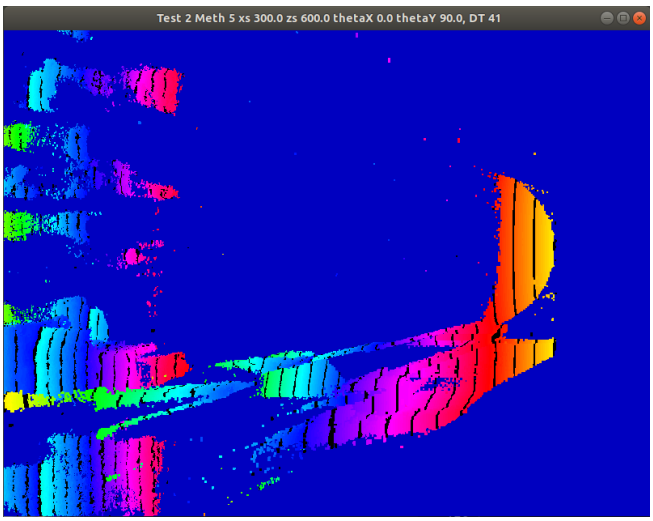


Figure 5.6 Showing Zoomed and Rotated View of the 3D Beach Ball Model

Figure 5.6 above shows a rotated and zoomed view of the shape of the front surface of the beachball and also the false colour scheme used. This is a cyclic depth-to-colour scheme with a more frequent change of colour than the depth-temperature maps shown on figures 4.1.6 to 4.1.14. The cyclic colour palette used

to generate figure 5.6 and subsequent figures, is shown on figure 5.7 below. The black bands represent 50mm intervals into the volume model along the z axis. On this palette zero is at the top left cell, reading positive moving to the right along each row and likewise down the table which does not end after 2 cycles as shown but will continue to cycle as depth increases.

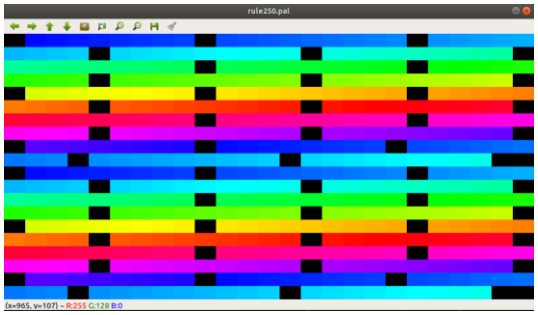


Figure 5.7 False Colour Depth Palette Used

Thus, from the colour palette shown in figure 5.7 and the zoomed figure 5.6 it can be deduced that the ball is placed between 450 and 500 mm in front of the camera, the actual measured distance was 485mm. This cyclic map though not exact, gives a better indication of actual depths and distances involved than a simple linear temperature map would. Note that the colour scheme is applied along the Z-axis of the 3D model volume and does not re-orient as the viewport is moved. This 3D model volume is aligned with the camera initial starting position with all angles zero and the camera placed at a pre-specified location in the model volume, typically at a middle position in X and Y coordinates and at zero Z (or depth into the volume). The initial camera 3D image is projected into the model from this initial position.

5.2 PCA Reduction

With reference to figure 5.2 above, for the case **ImageTest**=3 the PCA reduction process developed in section 3.6 is verified by test. This is achieved by decomposing the camera image contained in **d_Depth** into a PCA-reduced set of vectors held in **pointVec**. This is achieved with:

PCA_Reduce_Launcher(**d_Depth->pointVec**)

This data-reduction is then used to reconstruct the original dataset using the equations defined in section 3.7 of this report. This is achieved with:

PCA_Surface_Launcher(**pointVec->DarrySurf**)

The GPU implementation of the PCA method is discussed in detail in section 3.6. The image in figure 5.8 below demonstrates recovery of the depth camera data camera data from the PCA reduced data contained in **pointVec**.

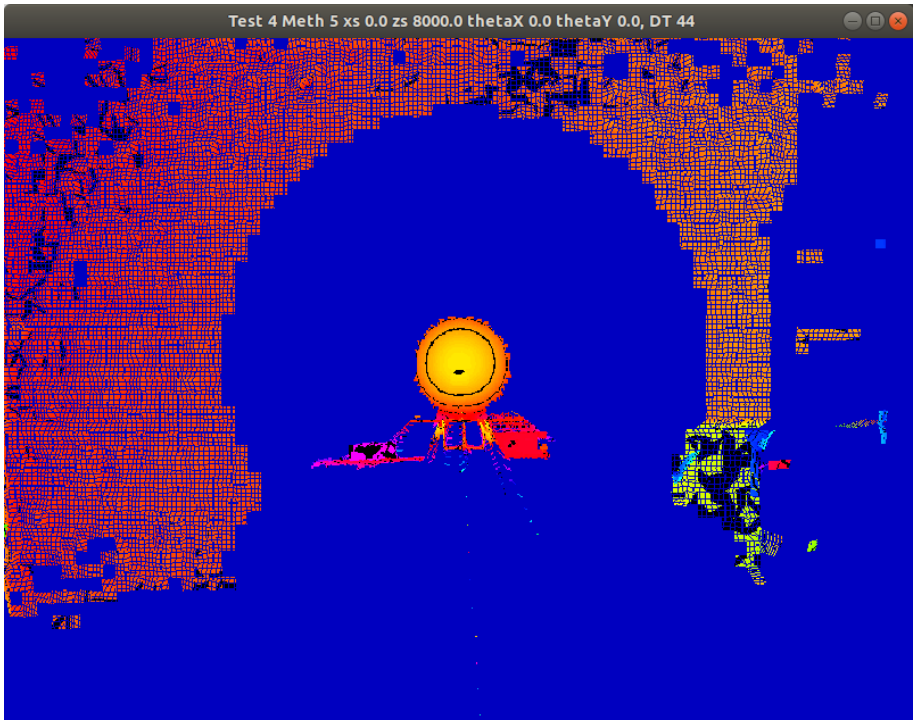


Figure 5.8 PCA Reduced Image of Beachball (CellX = 8)

This image compares directly with the unreduced image in figure 5.5 above. As can be seen there are clearly missing segments and evidence of the discretising effect of segmentation, particularly in the shadow edges at the back wall. This image is composed of a series of segmented rectangular planes centred on the mean vertex for each segment with the rectangular surface oriented by the surface normal obtained from the PCA reduction process. The cell size used for this segmentation was 8x8 pixels, so the original 307200-pixel image was PCA reduced to a **pointVec** vector containing 4800 data points (this vector includes vertex and surface normals for each of the segments). The reduction process also prunes segments which contain zeros indicating no return from the laser. Pruning is important as it removes noise effects due to edge discontinuities discussed in section 3.1. This further reduces the image size; the amount of reduction is critically controlled by cell size **CellX** and the effect of cell pruning. The following table 5.1 shows the effect of the PCA reduction process, the actual values are for the beachball scene in figure 5.9 above. Where Reduction Factor = Retained Points/(640x480)

Cell Sizes (pixels)	Total Points	Retained Points	Pruned Points	Reduction Factor
4	19200	13684	5516	0.04454
5	12288	8474	3814	0.02758
8	4800	3023	1777	0.00984
10	3072	1847	1225	0.00601
16	1200	645	555	0.00210

Table 5.1 Showing Effect of Cell Size and Pruning on PCA reduction

PCA reduction combined with cell pruning, provides a very effective means of reducing the large number of data points (307200) in the original depth image to a much smaller set of points. There is however a balance to be struck, if the cell size chosen is large the camera image is reduced to a small number of large planes. These planes will be robust to camera noise being based on averaging a large number of data points, but there will also be a loss of critical detail within the image. Choose a cell size too small and the PCA-reduced points will be prone to camera noise. The following figure 5.9a to 5.9c clearly shows the effect of cell-size on PCA reduction of a chair 2 metres from the camera, the images are zoomed.

Figure 5.9a No PCA reduction

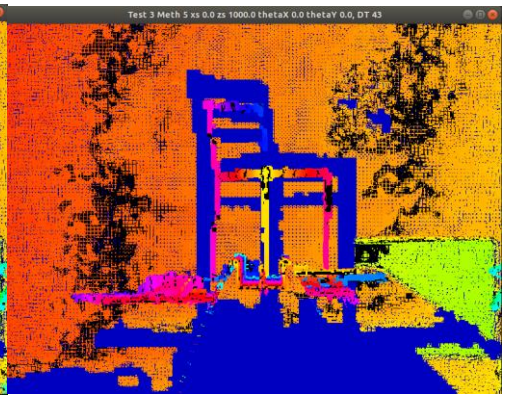


Figure 5.9b PCA cell size 4 pixels

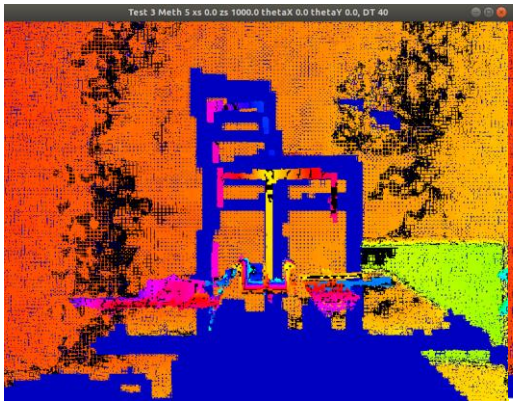


Figure 5.9c PCA cell size 5 pixels

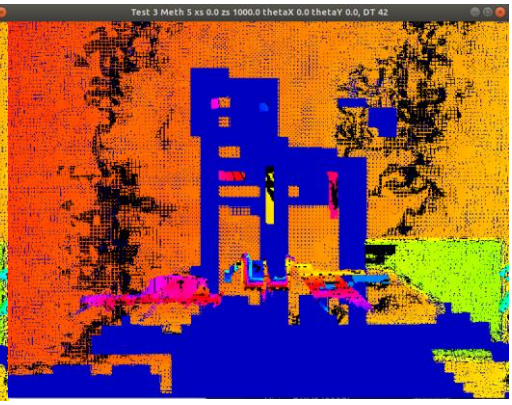


Figure 5.9d PCA cell Size 8 pixels

Clearly the cell size used in PCA reduction can lead to a significant loss of image detail when compared to the unreduced image in figure 5.9a. Figure 5.9c shows that most of the chair is removed from the image only the shadow and small fragments of the upper front legs remaining. For this reason, the full camera image (and not the PCA reduced image) is fused into the 3D volume model. For ImageTest=4 (ICP tracking cases) there are two paths through the code presented in figure 5.2 above. This is schematically illustrated in figure 5.10 below.

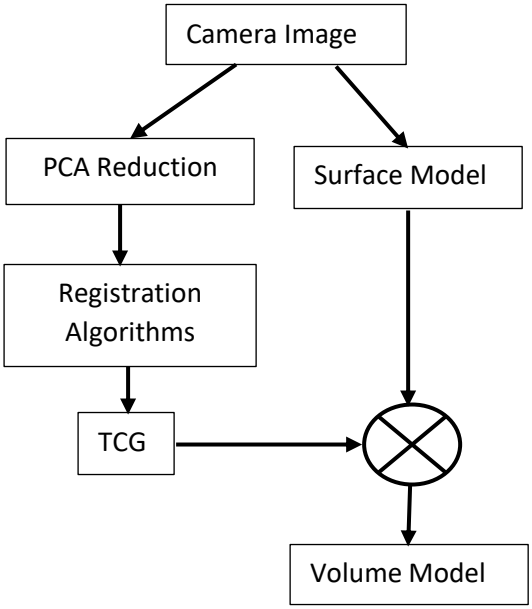


Figure 5.10 Showing Primary Code Paths for Tracking and 3D Modelling

The objectives of the above algorithm are two-fold, firstly to tracking camera movement. This is achieved by the registration algorithms in the left-hand path, the output of which is camera pose encapsulated in the camera-to-global transformation TCG. This is then used to fuse the full camera surface data into the volume model in the right-hand path.

5.3 ICP Tracking and 3D Model Development – Initial Testing

As with any testing such as this, the combinatorics of different methods and parameter variations can, if not managed, quickly become overwhelming. To simplify the problem somewhat, testing will rapidly focus in on the methods seen to be of most interest, whilst others, seen to be less promising, will be discarded.

There is of course a risk in such a complex, multi-dimensional search of missing an optimum combination. However, simple consideration of the reality of testing shows than the range of test scenarios is infinite, given this is the case, it is pointless to optimise the parameters of any given method for any given test scenario. What is sought is a tracking algorithm that is inherently robust without the need for complex parameter optimisation cycles.

Therefore, in this first round of testing, the goal is to eliminate those methods that do not give improved tracking and modelling performance for the given common scenario, before taking the winning combinations forward to more exhaustive and rigorous testing and methods development.

In this section the ability of the ICP algorithms documented in chapter 3, with various combinations of parameters, are evaluated when performing a standard ServoTrack test case known as: XTest1_05_Chair.

The chair was chosen as good subject for testing as it contains features at multiple scales combining long limbs with thin support members. This test is one of the simplest possible being a stepped test (ServoTrack Mode=2) and consists of 3 stationary camera frames, followed by stepping forward along the track in 10mm steps to one metre. There are no inputs on the pitch and yaw axis during this test and a camera frame is taken after each 10mm step.

To assess the accuracy of the system, tracking results are presented for the algorithm and the ServoTrack sensors on the same plots for direct comparison. As the camera is the only thing moving, the 3D model of the scene should remain static. The placement of camera images into the 3D model utilises the computed tracking data to place them, thus if the tracking algorithm is in error, the 3D model image will be extruded or rotated such that the imaged scene is distorted when viewed at the end of the run. Thus, both the following tracking plots and the 3D model images shown, illustrate the accuracy and quality of the tracking algorithms tested.

5.3.1 Point-to-Point ICP Testing

The theory for this method is discussed in section 2.3. A discussion of the algorithms used is given in sections 3.5 to 3.9. This method is the simplest to implement and is seen to work well under “ideal” conditions. The following conclusions have been previously established by test:

- P2P does not require a large volume of data to perform accurate registration. Conclusion: as few as 6 data point-pair matches will allow accurate registration.
- P2P will correctly register in one step (no iteration needed) when two frames are created from the same image and one frame is then synthetically rotated and displaced relative to the other. Conclusion: Data volume is not an issue, the method it is scalable.
- P2P will correctly register without drift buildup when looking at the same scene repeatedly with a stationary camera. Conclusion: Camera noise on a stationary image is not an issue.
- P2P will also do the above stationary test at 30 frames per second. Conclusion: Frame rate is not an issue, though frame rate will lead to rapid drift buildup where noise does exist for other reasons.
- For accurate P2P tracking, it is essential that the same image data is present in both the reference and test frame images. Conclusion: the method is not robust to data differences.

This method has proven to be quite sensitive to occlusion. Whilst PCA reduction of image data, combined with point-pair matching is designed to reduce the data in both registration images to a common sub-set, inevitably camera motion causes a change in viewpoint and consequently partial image occlusion.

In the following test scenario, the ServoTrack carriage is traversed along the track for 1 metre towards the chair. For this test RegMethod=1, cell size CellX=4 and all other settings are as per figure 5.1.

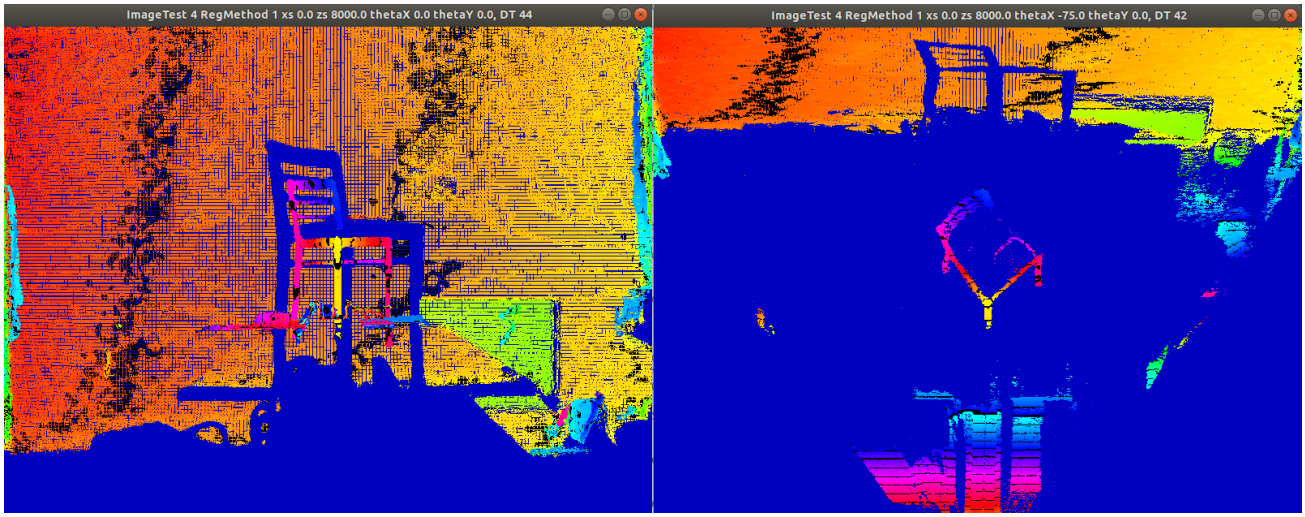


Figure 5.11 3D Model of Chair Front (left) and Plan (right) Views at Start of P2P Tracking Test

Figure 5.11 above shows the 3D volume from front and at 75 degrees (plan) at the start of the run (actually frame 3) before carriage motion starts. The following table 5.2 defines the tracking plot parameter labels for figure 5.12 and all subsequent tracking test plots in this chapter.

Parameter	Description	Units
Tx	ServoTrack Measured Pitch (θx)	Degree
Ty	ServoTrack Measured Yaw (θy)	Degree
Tz	ServoTrack Roll (Fixed=0) (θz)	Degree
pTx	ICP Computed Pitch	Degree
pTy	ICP Computed Yaw	Degree
pTz	ICP Computed Roll	Degree
errTx	Angular Error (Tx – pTx)	Degree
errTy	Angular Error (Ty – pTy)	Degree
errTz	Angular Error (Tz – pTz)	Degree
Dx	ServoTrack Lateral Position (Fixed)	m
Dy	ServoTrack Vertical Position (Fixed)	m
Dz	ServoTrack Horizontal Measured Position	m
pDx	ICP Computed Lateral Position	m
pDy	ICP Computed Vertical Position	m
pDz	ICP Computed Horizontal Position	m
errX	Position Error (Dx – pDx)	m
errY	Position Error (Dy – pDy)	m
errZ	Position Error (Dz – pDz)	m

Table 5.2 Showing Test and Tracking Parameter Notation

As can be seen from figure 5.12 below tracking of carriage Dz is quite good with a maximum error of around 12mm. However, lateral and vertical tracking errors (for the two axes that are constrained) are

worse with errors in excess of 40mm. Whilst yaw tracking limits at around 1 degree, pitch-tracking error (errTx) seems to be drifting off scale.

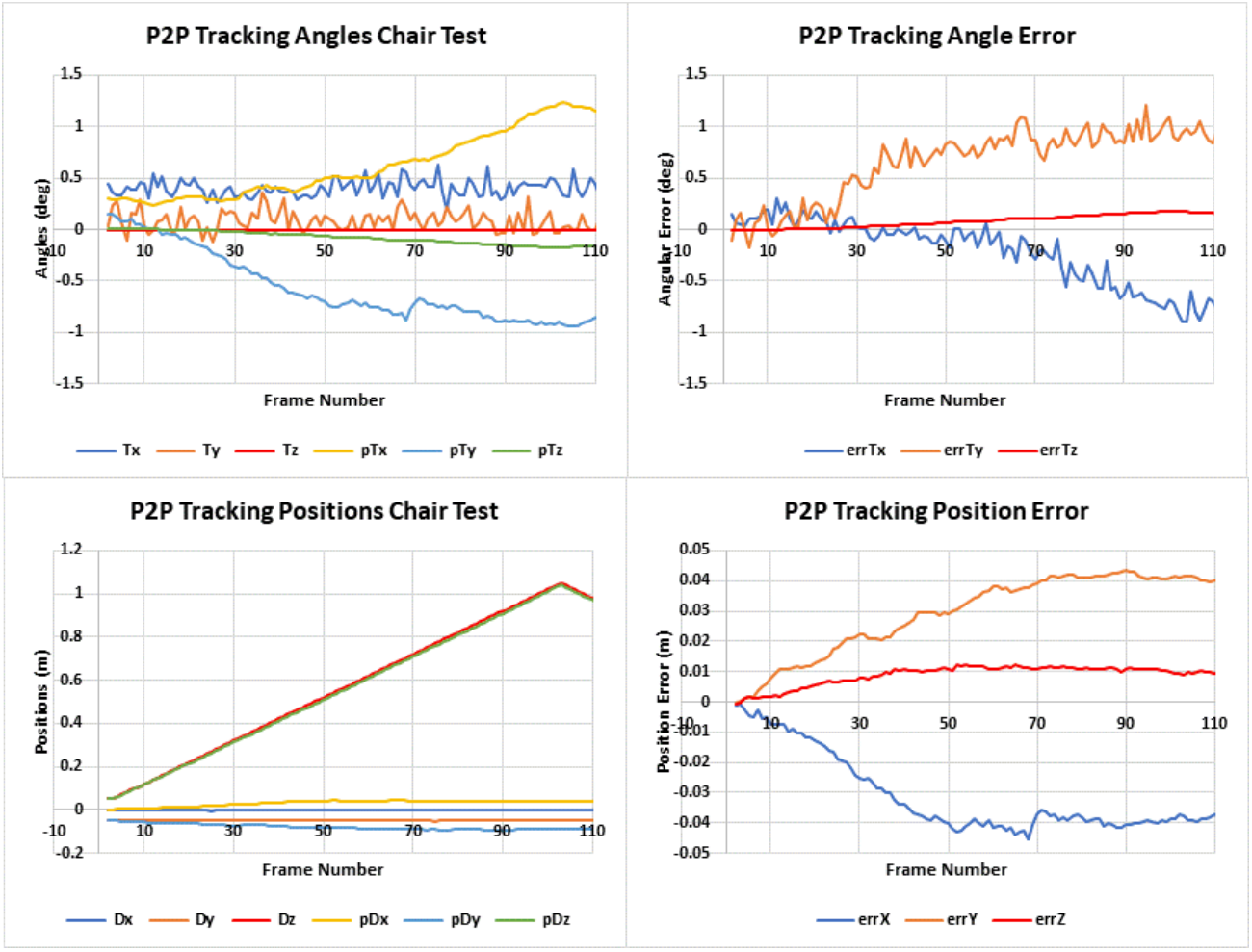


Figure 5.12 Point-to-Point Tracking Test XTest1_05_Chair

Figure 5.13 below shows the final result of the 3D model integration process after the tracking test. These views correspond to the same images at the start of the run shown on Figure 5.11 above.

It is notable in the plan view, that the 3D model (particularly the table surface and the back wall) are significantly developed during the tracking process, this is due to fusion of depth data into the 3D model. There is also some evidence of noise in the form of small isolated speckles of colour against the blue background in this image.

However as can be seen form the horizontal view, the 3D model of the chair is visibly extruded vertically due to the cumulative effect of tracking errors. This would not be the case if the angular and lateral drifts were small. Clearly using this method for tracking and model building requires further work. However, the tendency to drift under what are essentially ideal test conditions (no camera rotations and stepped longitudinal motion) promotes alternatives methods such as point-to-plane which is discussed in the following section.

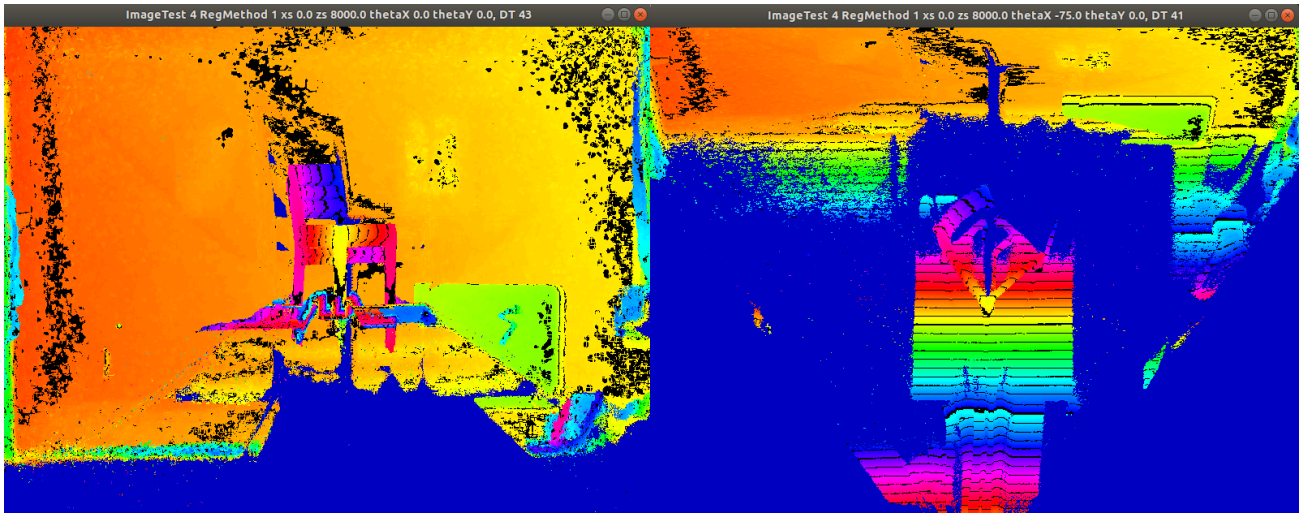


Figure 5.13 3D Model Images of Chair at End of P2P Tracking Test

5.3.2 Point-to-Plane ICP Testing

Figure 5.14 below shows tracking results for point-to-plane tracking test when run for the same test case (XTest1_05_Chair) and conditions as the point-to-point test shown on figure 5.12 above.

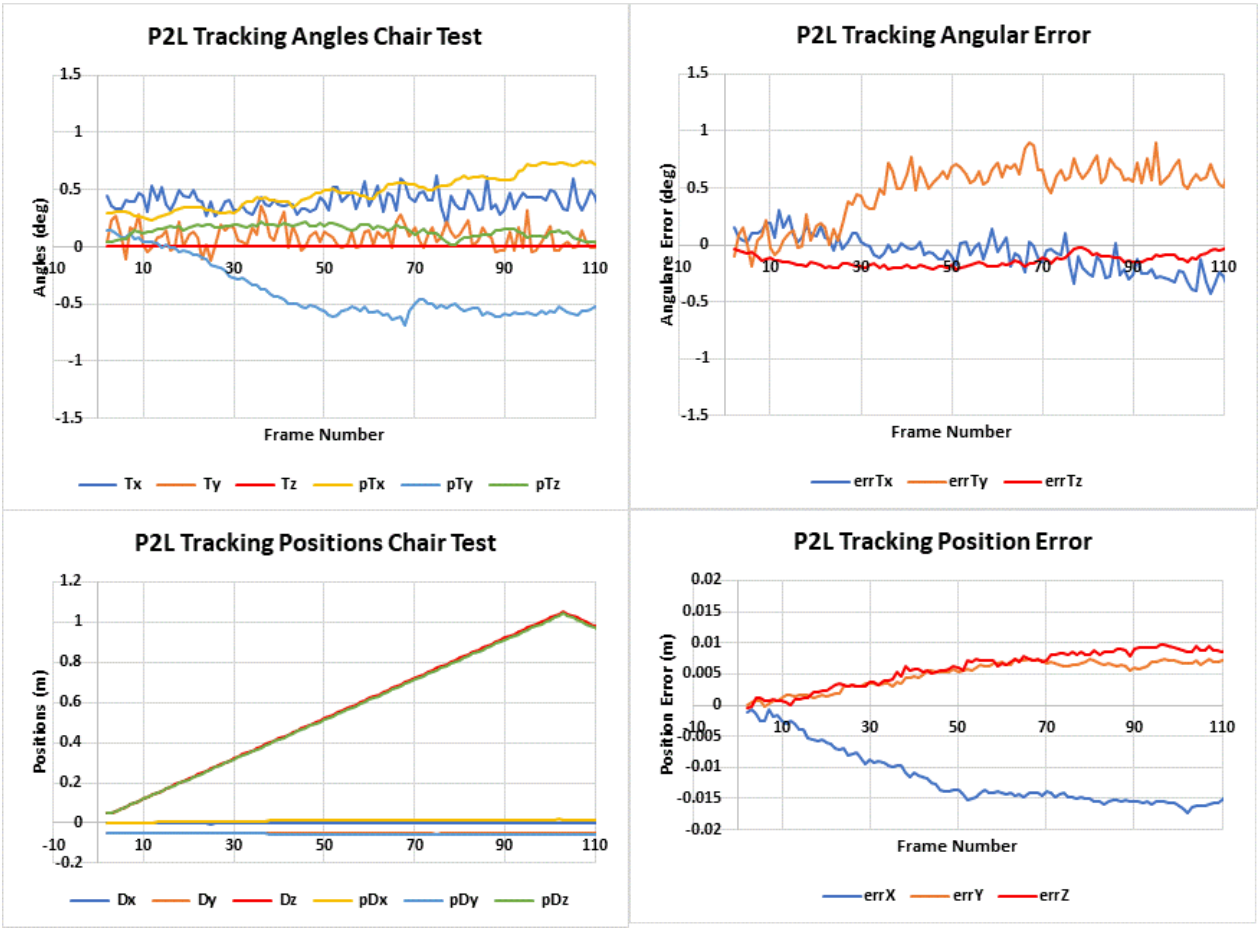


Figure 5.14 Point-to-Plane Tracking Test XTest1_05_Chair

Figure 5.14 above shows that, though not perfect, this method provides superior tracking accuracy when compared to the point-to-point tracking results for the same test case. All conditions for this test were identical with the P2P test, the only change being the switch to P2L tracking algorithm (RegMethod=2).

Drift is still evident on figure 5.14 though the magnitudes are now smaller, with a worst-case lateral position drift error (errDx) of ~15mm and a worst-case angular yaw error (errTy) of ~0.8 degrees. Though small, these errors do cause some extrusion of the 3D chair model shown in figure 5.15 below. This is evidenced as a thickening of the horizontal bars on the chair when compared with the original chair at the start of the run (see figure 5.11) above.

Note also that the dark-blue shadow of the chair on the back wall, visible in figure 5.11 is almost eliminated. This is due to motion of the camera uncovering some, though not all, of the occlusion from the initial model view in figure 5.11.

Occlusion due to viewpoint change is one of the main sources of unavoidable image difference during frame-to-frame registration and the only mitigation for this is thresholding on point-pair matching. However, it also demonstrates that the 3D model integrates new information as the camera moves and this is a key benefit of this image-fusion model building process.

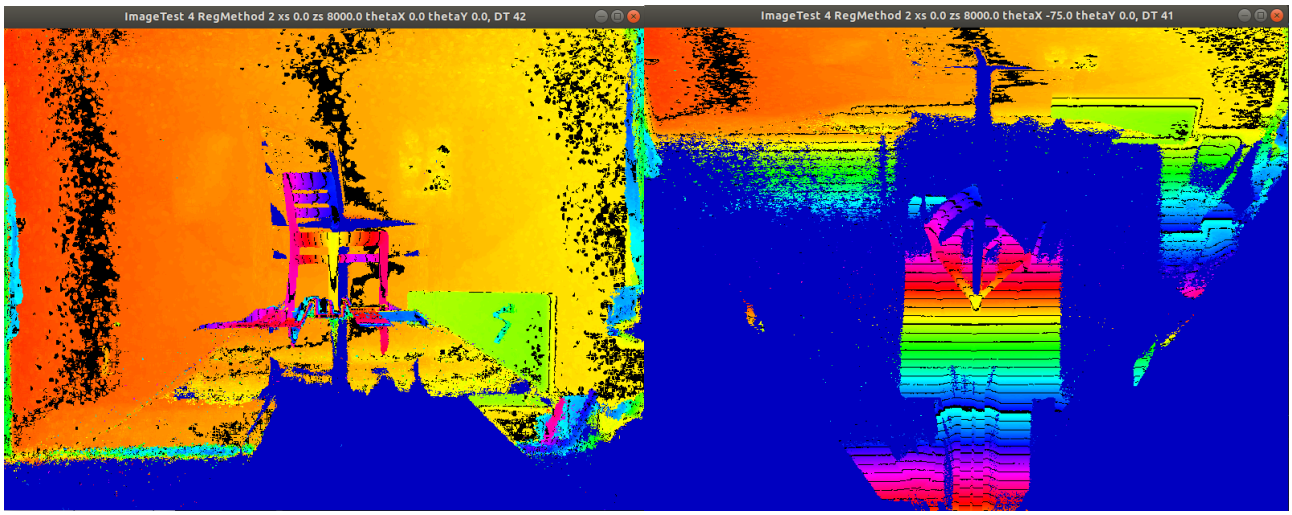


Figure 5.15 3D Model Image of Chair at End of P2L Test

5.3.3 Point-to-Plane with Bilateral Pre-Filtering

A discussion of Bilateral filtering of depth data is given in section 3.4. Here the use of bilateral prefiltering is re-appraised when implemented on a full-scale point-to-plane tracking test. The figures 5.16 and 5.17 below show tracking and 3D model image results for the same chair test as shown in figures 5.14 and 5.15 above with the addition of Bilateral pre-filtering (PreFilter=1) and filter settings as given on figure 5.1.

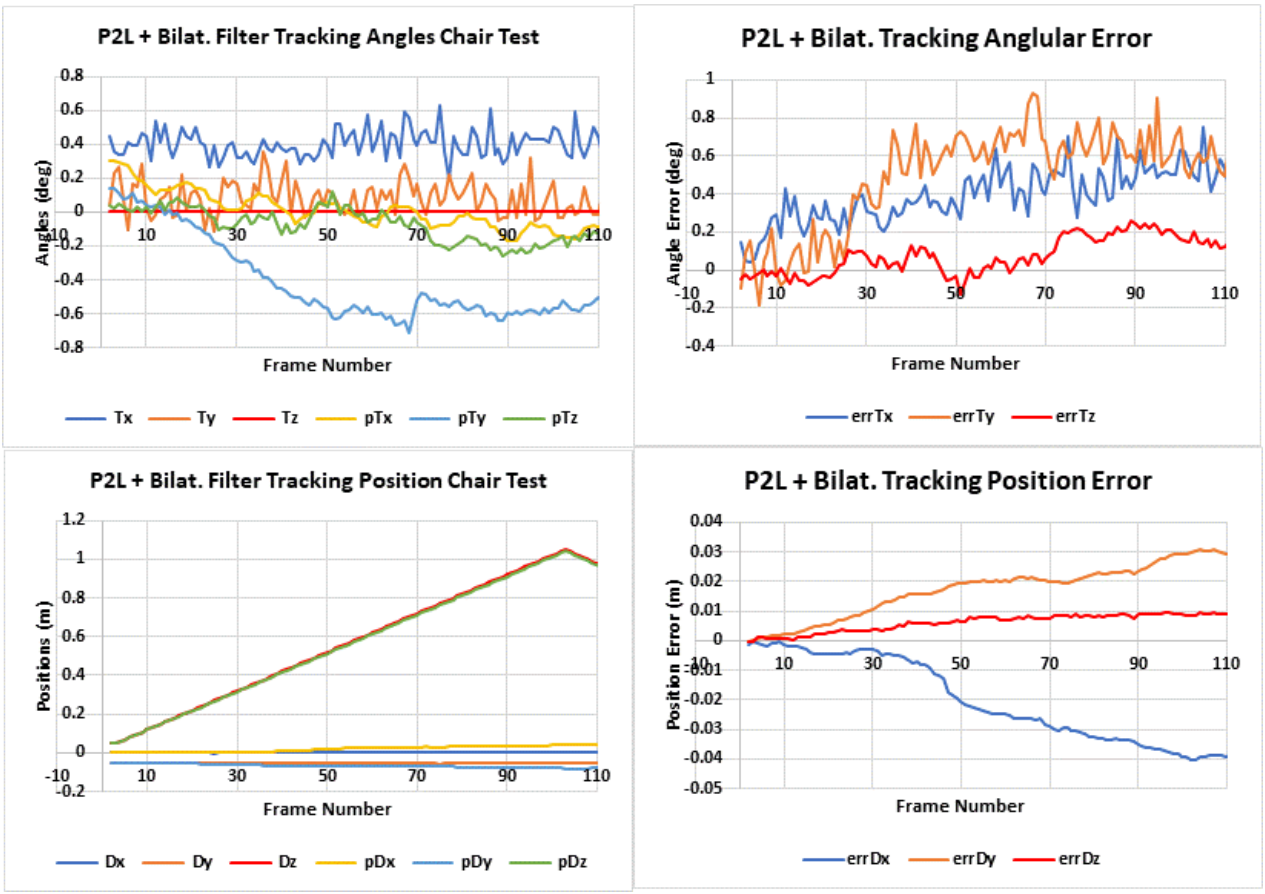


Figure 5.16 Point-to-Plane + Bilateral Prefilter Tracking Test XTest1_05_Chair

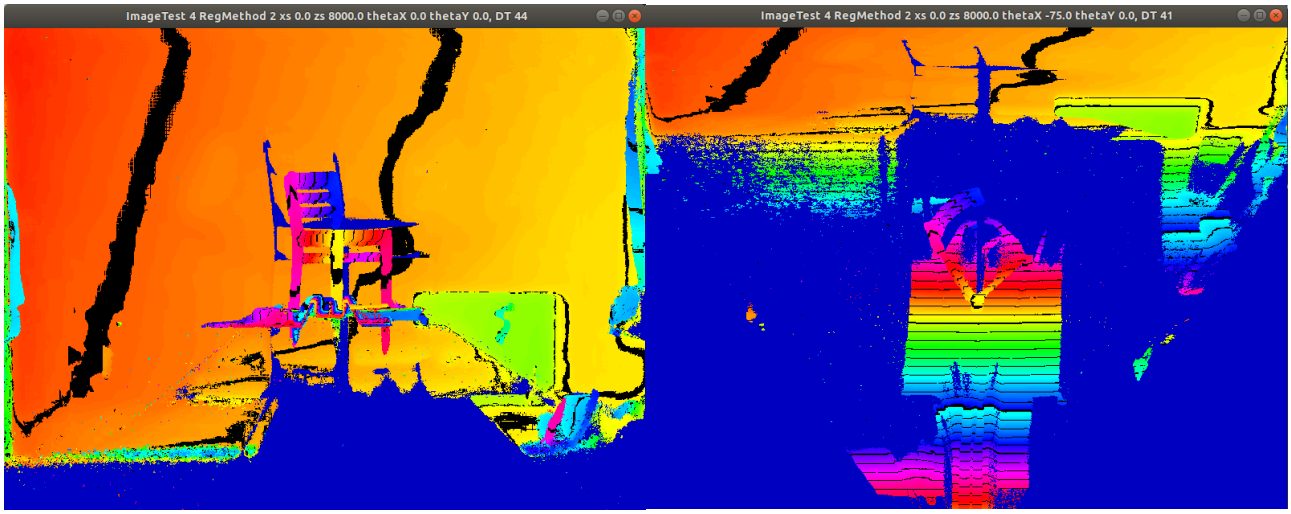


Figure 5.17 3D Model Images of Chair at End of P2L + Bilateral Prefilter. Test

The above 3D model images in figure 5.17 look considerably cleaner when compared with the same images in figure 5.15 above. This filter clearly reduces camera noise in the 3D model. However, by comparison of tracking errors on figures 5.14 (no filter) and figure 5.16 (filtered) it is clear that the addition of the Bilateral Prefilter has, in this case slightly degraded tracking accuracy.

5.3.4 P2L with Deferred Update to Reference Frame

For all the previous tests, the current frame becomes the reference frame for the next step. This mechanism can enhance the effect of drift. This is due to the fact that small incremental changes in movement ensure that camera noise and other minor effects have a larger overall effect on tracking accuracy due to the incremental nature of the tracking update equations.

A recognised counter to this effect is to postpone replacement of the reference image so that the ICP algorithm is working with the same reference frame for a number of subsequent steps each of which uses a larger difference between reference and current frame which may help to supress noise-induced-drift by improving signal to noise ratio.

Figures 5.18 and 5.19 below show results when update to the reference frame is deferred to every fourth frame (ICPFrames=4, PreFilter=0 on Figure 5.1 above). However, to improve assessment of 3D model development, a zoomed image of the central chair is compared with the same zoomed image for the baseline P2L case in figure 5.15 above.

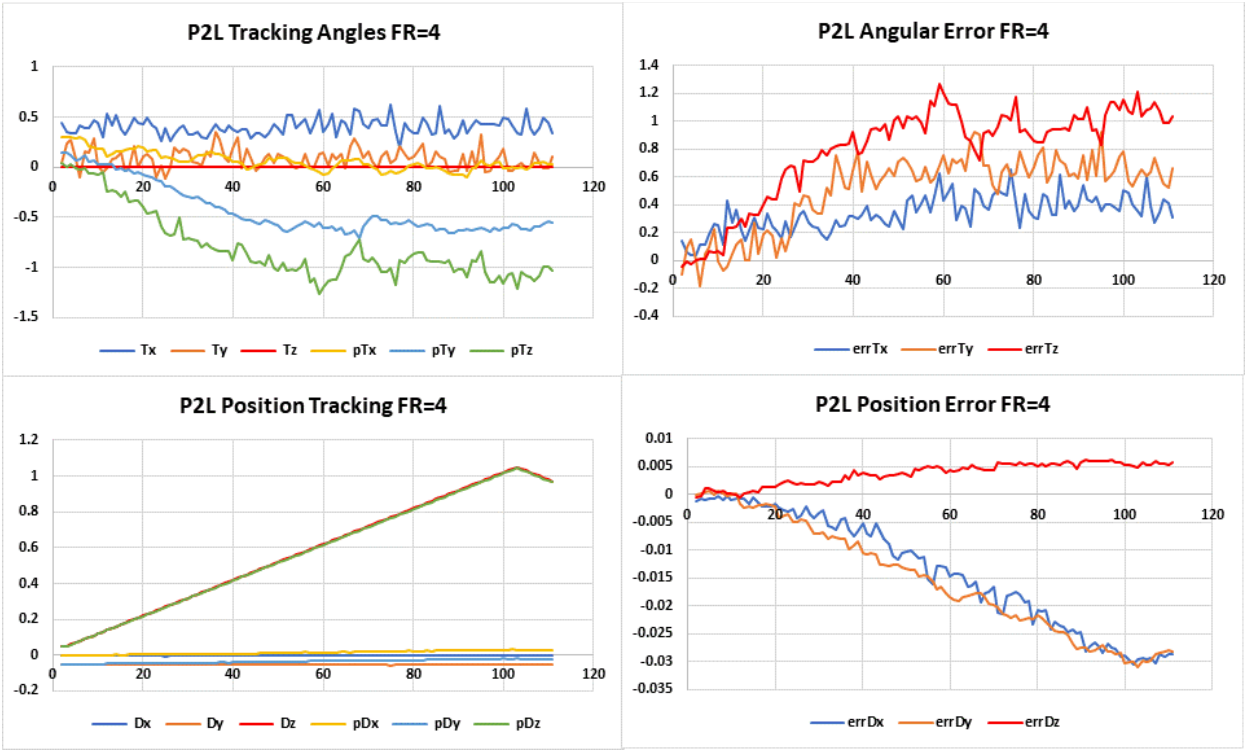


Figure 5.18 P2L Tracking with Reference Frame Updated every 4th Frame

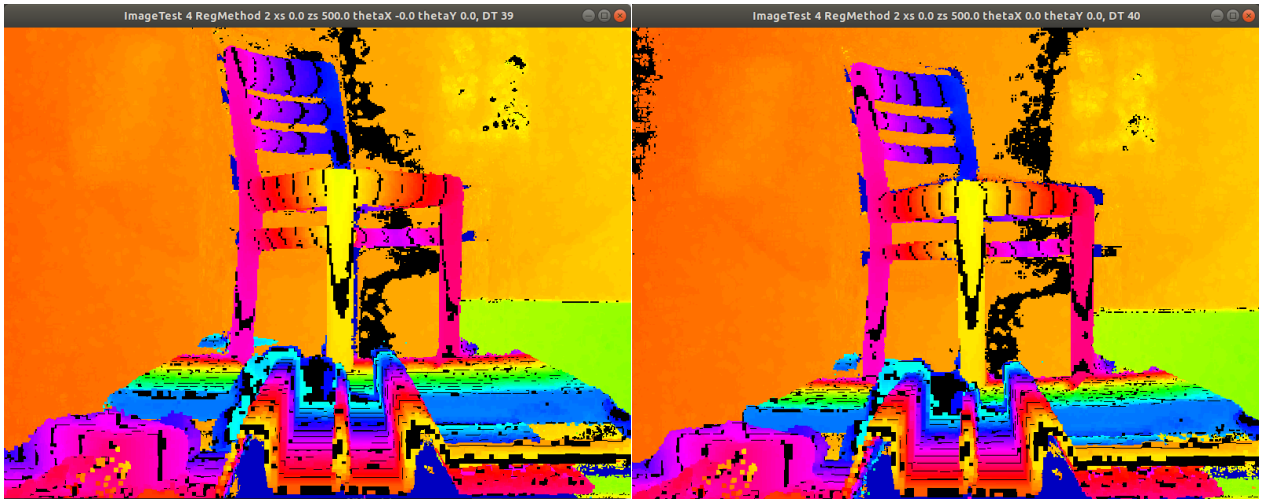


Figure 5.19 Zoomed P2L Baseline (Left) and with Reference Update Every 4th Frame (Right)

The above result show there is definite benefit in deferring update of the reference frame for this case. As evidenced by reduced thickness of the horizontal chair members. Position tracking error (errDz) is also reduced to ~5mm compared with ~10mm for the P2L baseline (figure 5.14 above).

Interestingly lateral and vertical errors errDx and errDy have both increased relative to baseline and show an undesirable constant rate of increase. However, the 3D model results are still far from perfect and the lateral and vertical drift are a concern.

5.3.5 Cell Search Algorithm Testing

This algorithm, discussed in detail in section 3.12, is an extension of the point-to-plane method with the additional modification of utilising the Förstner-Moonen (FM) metric to further qualify the point-pair-matching process. This method should in theory select point-pairs that are not only geometrically close, but also come from regions that are geometrically similar within a given threshold on the FM-metric values. Results for the chair tracking test (for RegMethod=3, ICPFrames=4, CellX=4, PreFilter=0) are given on figures 5.20 and 5.21 below.

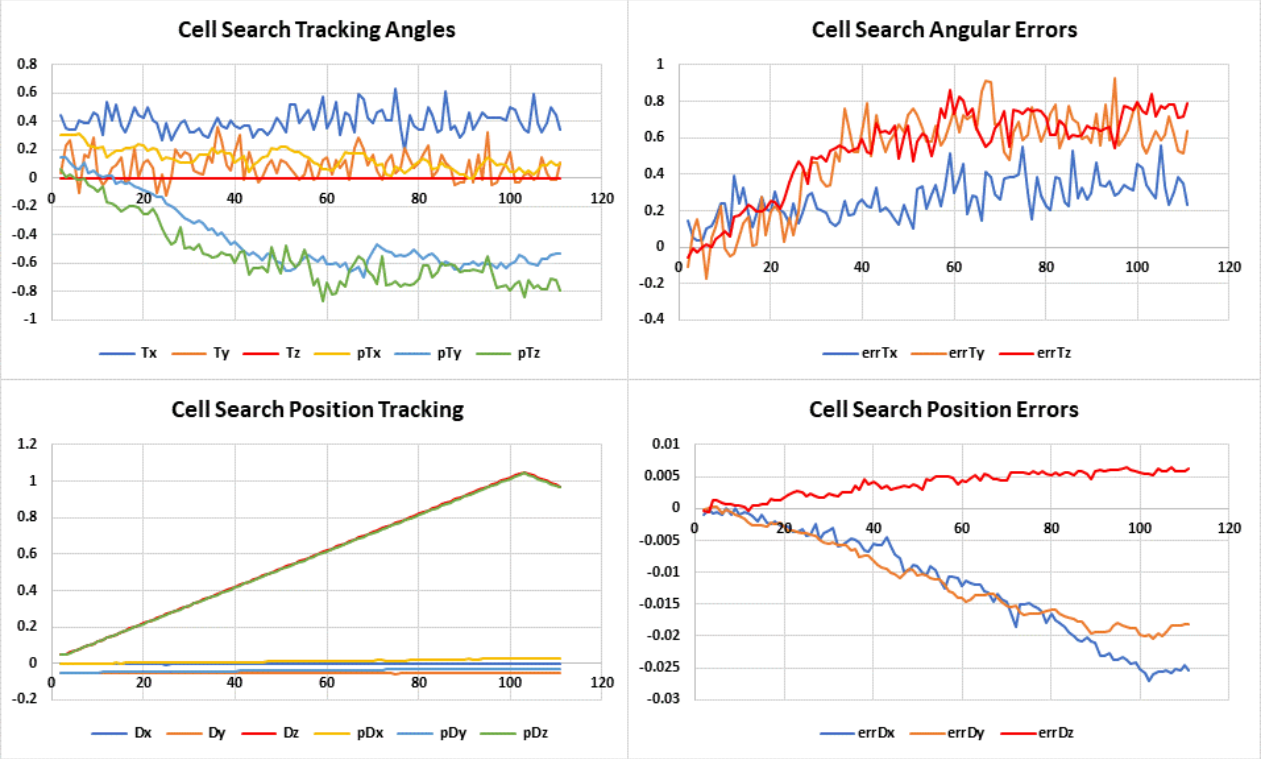


Figure 5.20 Cell Search Tracking Test XTest1_05_Chair

Since the Cell Search method is an augmentation, all be it an elaborate one, of the basic point-to-plane method and the reference update is also deferred to 4 frames in both cases. Thus, the above tracking results in figure 5.20 can be compared directly with figure 5.18. From Figure 5.20 it can be seen that angular tracking errors are all within 1 degree and that position tracking errors are in broad agreement with a similar ~5mm tracking error on errDz. However, the undesirable lateral and vertical drift trend seen on errDx and errDy in figure 5.18 are also present on figure 5.20.

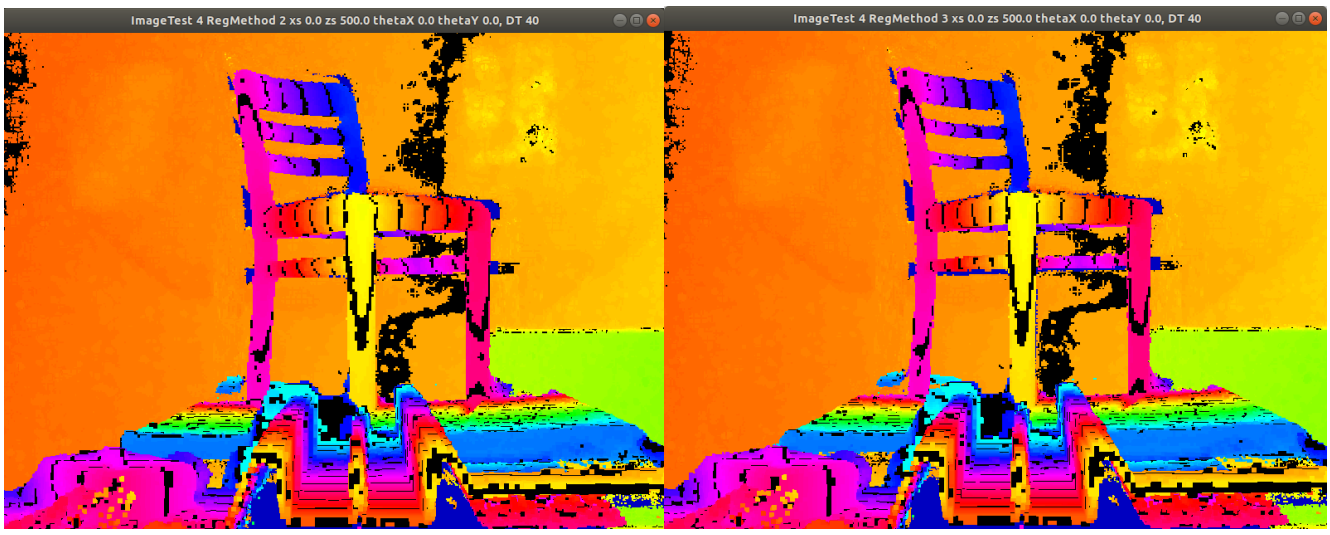


Figure 5.21 Comparing P2L with FR=4 (Left) with Cell Search FR=4 (Right)

From the 3D model comparison shown on figure 5.20 above it is evident that the Cell Search algorithm does improve 3D model accuracy. The width of the horizontal bars on the chair back (figure 5.20 right) are seen to very closely match those of the real chair. Given the 3D model has a voxel size of 5mm, this result is considered as good as can be expected for this simple test.

Summarising ServoTrack (mode=2) testing, the table 5.3 below shows the maximum absolute error values for each test case. This metric is reflective of the worst-case tracking errors during the run.

Test parameters for table 5.3:

- ImageTest=4, CellX = 4
- P2P = Point to Point, P2L = Point to Plane, P2LCS = Point to Plane + Cell Search
- FR = number of frames between Reference updates
- PreFilter = 1 = Bilateral Filter, 0=No Prefilter

Test Condition	Reference Figure	errTx (deg)	errTy (deg)	errTz (deg)	errDx (mm)	errDy (mm)	errDz (mm)
P2P, FR=1, PreFilter=0	5.12	0.90	**1.21**	0.17	**45.3**	**43.2**	**12.2**
P2L, FR=1, PreFilter=0	5.14	0.43	0.90	0.22	**17.4**	7.4	9.8
P2L, FR=1, PreFilter=1	5.16	0.75	0.93	0.26	**40.5**	**30.8**	9.6
P2L, FR=4, PreFilter=0	5.18	0.65	0.93	**1.27**	**30.5**	**31.0**	6.2
P2LCS, FR=4, PreFilter=0	5.20	0.56	0.92	0.86	**27.0**	**20.0**	6.4

Table 5.3 Maximum Absolute Tracking Errors for ServoTrack (Mode=2) Tests

From 5.3 above, results for Point-to-Point (P2P) tracking scores least well on all counts except roll error (errTz). The baseline P2L algorithm performs best on all counts with the exception of surge (errDz) tracking. P2L tracking with bilateral filtering does not improve tracking and gives considerably worse results for lateral errors Heave (errDy) and Sway (errDx). Delaying update of the reference by 4 frames appears to introduce errors in Heave and Sway which negatively correlate with position along the track (Dz and pDz on figure 5.18) the reason for this is not understood. However, this result shows marked improvement in the final 3D model over the P2L baseline as shown by the comparison on figure 5.19.

The Cell Search algorithm (P2LCS) also does not fair particularly well by this measure. However, the 3D model results are visibly improved over the equivalent P2L test for this case as shown on figure 5.21.

The above highly constrained stepped (mode=2) testing gave sufficient confidence in the test equipment and algorithms developed to proceed to the more challenging Real-Time cases given in section 5.4 below.

5.4 ServoTrack Real-Time Testing

In a real robot data scenario, it is assumed that data is gathered continuously at 30 Hz whist the robot/camera is also moving. This scenario is facilitated by the ServoTrack Test Rig when operating in real-time (mode=1). Six real-time test cases are presented in table 5.4 below.

Test	Track Speed (mm/s)	Run Time (sec)	Servo Input	Amplitude (deg)	Frequency (Hz)	Total Frames
RT1	200(95)	12	None			360
RT2	200(95)	12	Yaw	10	1/12	360
RT3	60(42)	25	Pitch	20	1/12.6	750
RT4	60(42)	25	None			750
RT5	200(95)	12	Yaw	15	1/24	360
RT6	200(95)	12	Pitch	15	1/24	360

Table 5.4 ServoTrack Real-Time Test Cases RT1 to RT6

The set speed of the track and the actual speed achieved are different due to the complexities of stepper motor speed control and real-time operation. Therefore, the above table shows the achieved speed in mm/s in parenthesis. In all cases the ServoTrack Carriage traversed over 1m of the track during the test period whilst the camera was also cycled with pitch or yaw input as indicated in the above table.

5.4.1 Fusing the 3D Model with ServoTrack Sensor Data Alone

To illustrate some of the difficulties when operating in mode 1, the six tests shown in the above table were carried out in two different ways firstly the ICP-computed tracking data were replaced by sensor data and this used to place the camera images directly into the 3D model volume. Since this data provides the ground truth it should, in theory, give the “best” model results. The final view of the chair used in section 5.3 is shown in each case at the end of the run. The final view of the chair in the 3D model for the 6 cases are shown on the composite image on Figure 5.22 below.

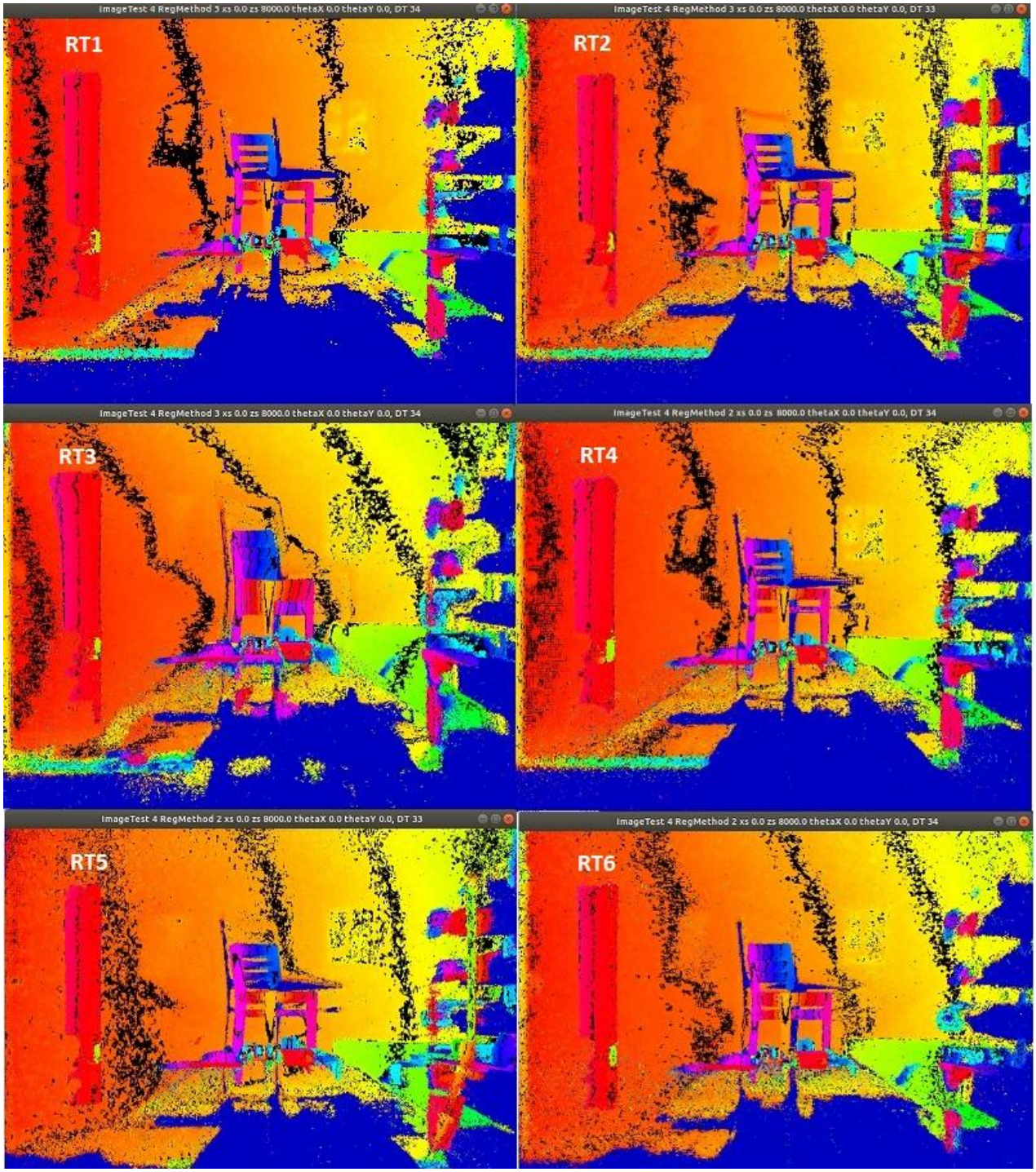


Figure 5.22 ServoTrack Real-Time Tests RT1 – RT6 3D Model Results Using Rig Sensor Data

The only difference between RT1 and RT4 is speed, the RT4 test being much slower with 750 frames compared to 350 frames for RT1. As can be seen by comparison between these two results, there is some blurring of the definition of the chair in the RT4 image indicating that buildup of sensor noise over many images is affecting model quality. RT2 and RT5 are similar yaw tests, though the frequency and amplitude are different. It is notable that the chair legs are slightly thicker in both cases relative to the none-yawed

RT1. RT3 and RT6 are both Pitch input cases and the chair is clearly extruded vertically in both cases. The effect is notably worse in the RT3 case, which was higher in amplitude and frequency.

These results were initially surprising as they highlighted an unexpected issue. Since the camera images are being placed in the model volume using the recorded sensor data one might expect that the 3D model at the end of the test would be of exemplary quality and this is clearly not the case. However, since both the sensors and camera are real devices, one would expect noise. That these "noise" effects lead to coherent image distortion was not expected and required further investigation.

Camera motion whilst the image is being captured would cause motion blur, this effect would be common to all tests though more pronounced on the faster tests. The camera sensor noise results in dynamic variations from frame to frame and again this effect would be common to all tests. A further effect may be due to vibration of the ServoTrack carriage, this is measured and shown on the following figure 5.23 for the three principal axes of the Carriage, note AcYg (gravity) is inverted on this plot.

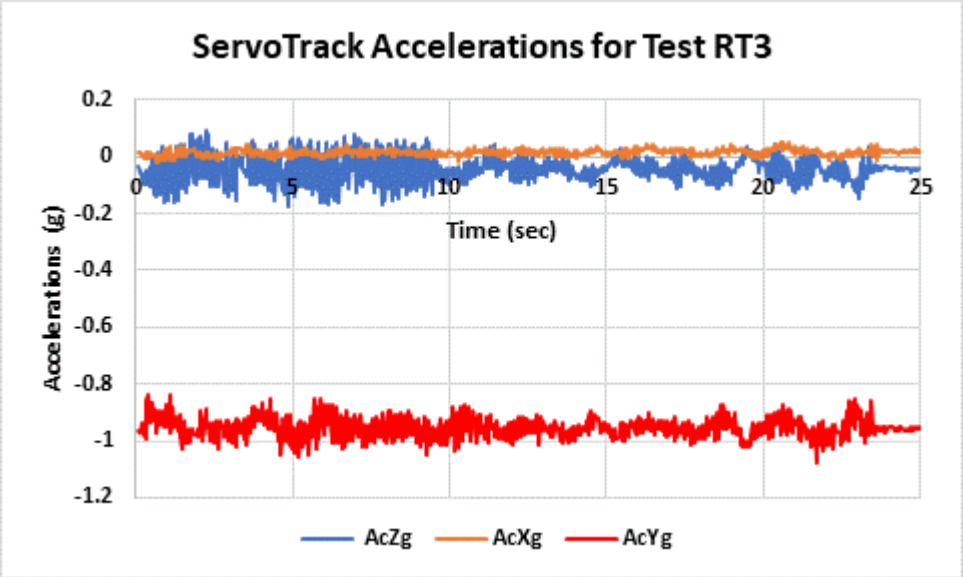


Figure 5.23 ServoTrack Carriage Accelerations (in Camera Axes) During Test RT3

There is clearly acceleration in the camera longitudinal (AcZg) acceleration and some very minor lateral (AcXg) acceleration. The dominant frequency is approximately 9 Hz and is thought to be due to the mass of the carriage on the elasticity of the stepper motor drive belt. This acceleration effect if significant, would also cause a general blurring (similar to motion blur) on all tests.

The most likely hypothesis as to cause of the extrusion of the chair image in the 3D model, particularly pronounced in test RT3, is believed to be due to latency between the time the camera image is captured and the sensor data is being recorded along with it in the database. If these two events are not well synchronised, then one would expect that event timing discrepancy between image capture and sensor

data would lead to image positioning errors within the model and this effect will be most visible on the moving axis direction, hence vertical extrusion of the chair for a pitch test.

The following plot, figure 5.24 shows the actual pitch data recorded together with a derived pitch rate, where the derived signals were obtained by simple numerical differentiation of the 30Hz pitch and pitch demand signals.

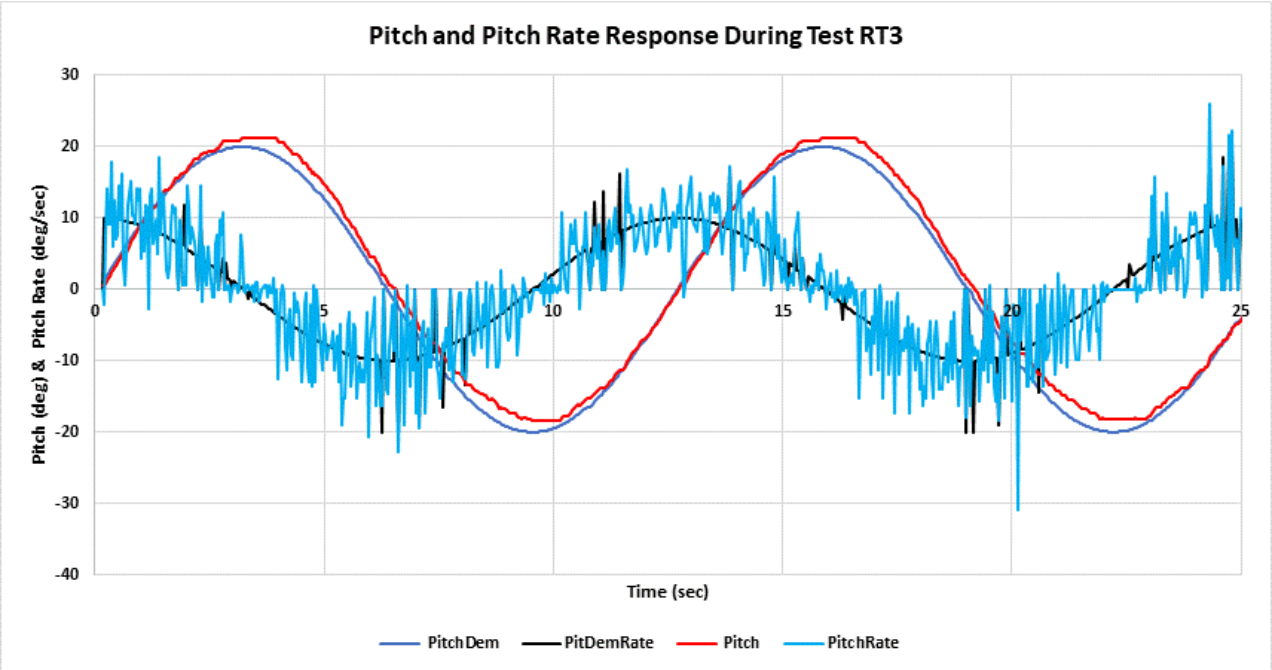


Figure 5.24 ServoTrack Test RT3 Pitch Sensor and Pitch Rate Response

There is some discontinuity evident in real-time sensor data recording. This is evident as a nonlinear response on PitchDemRate, most noticeable between 10 and 12 seconds and elsewhere on figure 5.24 above, where spikes are evident. This result highlights the one of the difficulties of real-time recording over multiple computers. The pitch sensor signal is first captured on the Arduino (running CarDRV) and fed via USB to the Intel® NUC™ embedded computer running CarHost which in turn passes this data over a Linux FIFO pipe to the CarCam program for final recording along with the captured depth frame.

The assumption is that the sensor data is concurrent with the image frame but these separate computers are not time-synchronised and though they are all operating at 30 Hz, due to slight variations in latency, it is occasionally possible for a data frame update to be missed. What happens then is that the data for the subsequent frame compensates with a jump in position, leading to the nonlinear spikes seen on the PitchDemRate response. However, this is not the cause of the very noisy response seen on PitchRate which is believed to be due to stick-slip-friction on the pitch signal due to friction in the pitch servo motor. This stop-start behaviour on pitch will further amplify any mismatch between camera frame capture and sensor signal due to latency.

This latency/jitter effect combined with slip-stick friction on the pitch servo somewhat undermines the ServoTrack test rig's ability to act as a ground-truth reference for real-time tracking accuracy testing.

These issues were in fact the main reason that Mode 2 (stepping mode) was developed first, as it effectively removed time and rig response variations from the equation and allowed development of the tracking algorithms on a more stable basis. However, Mode 2 does not include the real-time effects, such as demonstrated above. It is clearly important to understand these effects when developing algorithms for a real robot operating at 30Hz using inexpensive consumer electronics.

To understand the effect of latency on image placement within the 3D model it is possible to re-synchronise the data between sensors and depth image capture. This is achieved by buffering the time history data for the sensors and varying the time advance/delay between the sensor data and the depth camera images used in building the 3D model. This should give an indication of the effect of latency and confirm or disprove the hypothesis that this is the main cause of the extrusion seen in test RT3 and elsewhere. This is done for a 4 frame delay shown on figure 5.25 below.

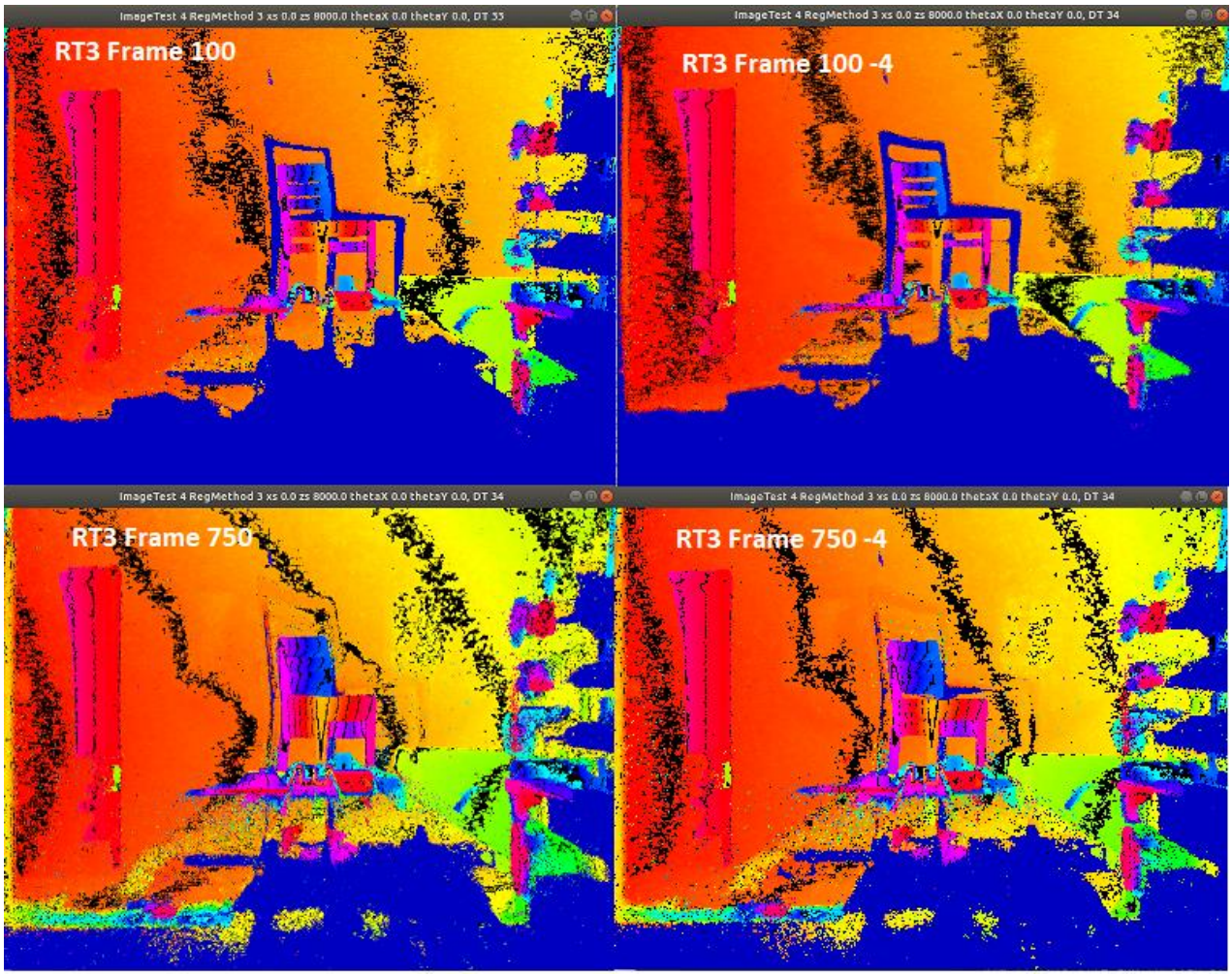


Figure 5.25 Effect of Slipping Sensor Data 4 Frames, Test RT3 after 100 and 750 Frames

The results of figure 5.25 above show comparison of RT3 test results after 100 and 750 frames, with no modification (on the left) and the same results when the sensor data is shifted by -4 frames relative to the camera image data (on the right). Whilst none of the above provide an accurate final image of the chair, it is clear that delaying the data by 4 frames does improve the chair images on the right in figure 5.25. However, the complex friction-dominated behaviour of pitch-servo, combined with the evident real-time latency variation (jitter), leads to the conclusion that the sensor data recorded is far from an absolute ground truth.

However, it would be a mistake to disregard this data. As will be shown in the following tests, sensor data provides good information to validate tracking, the proviso being, that this is within the accuracy limits of the sensors and recording process. It was always understood that the use of cheap radio-control servos and latency limitations in recording would lead to accuracy limitations. To achieve better would require equipment costing many times more and further (extensive and time consuming) computer hardware development.

5.4.2 Point-to-Plane Tracking in Real-Time

The following testing uses a combination of final 3D model image quality and a comparison between computed tracking output and ServoTrack test results to assess the tracking accuracy of the algorithms tested when supplied with real-time data gathered from the ServoTrack test rig.

Since the depth camera images are the only input to the ICP tracking algorithms and this image data is then used to fuse camera images into the 3D model at the computed pose, the accuracy limitations of ServoTrack sensor data have no influence on this process. Shortcomings in the tracking algorithms will produce distorted images and good tracking will result in crisp images at the end of each run, as shown in the final images presented below.

When comparing computed tracking output history from the ICP algorithms with that of the ServoTrack sensor data, it should be born in mind that the ServoTrack sensor data is not definitive especially in the time axis and for this reason the difference between model and test results are considered discrepancies rather than errors.

The objective of this section is to look at what accuracy and performance can realistically be achieved in real-time at 30 Hz with a typical games laptop actually used for this testing.

As for section 5.3 testing was carried out off-line using data gathered on the ServoTrack test rig but processing time is critical to real-time performance and so is also monitored.

The following composite image figure 5.26, shows the 3D model at the end of point-to-plane tracking tests for the same 6 ServoTrack real-time test cases defined in table 5.4 above. For the following test:

- CellX = 5
- ICPFrames = 1
- PreFilter = 0

This combination was found to give the best compromise between accuracy and GPU process time per frame.

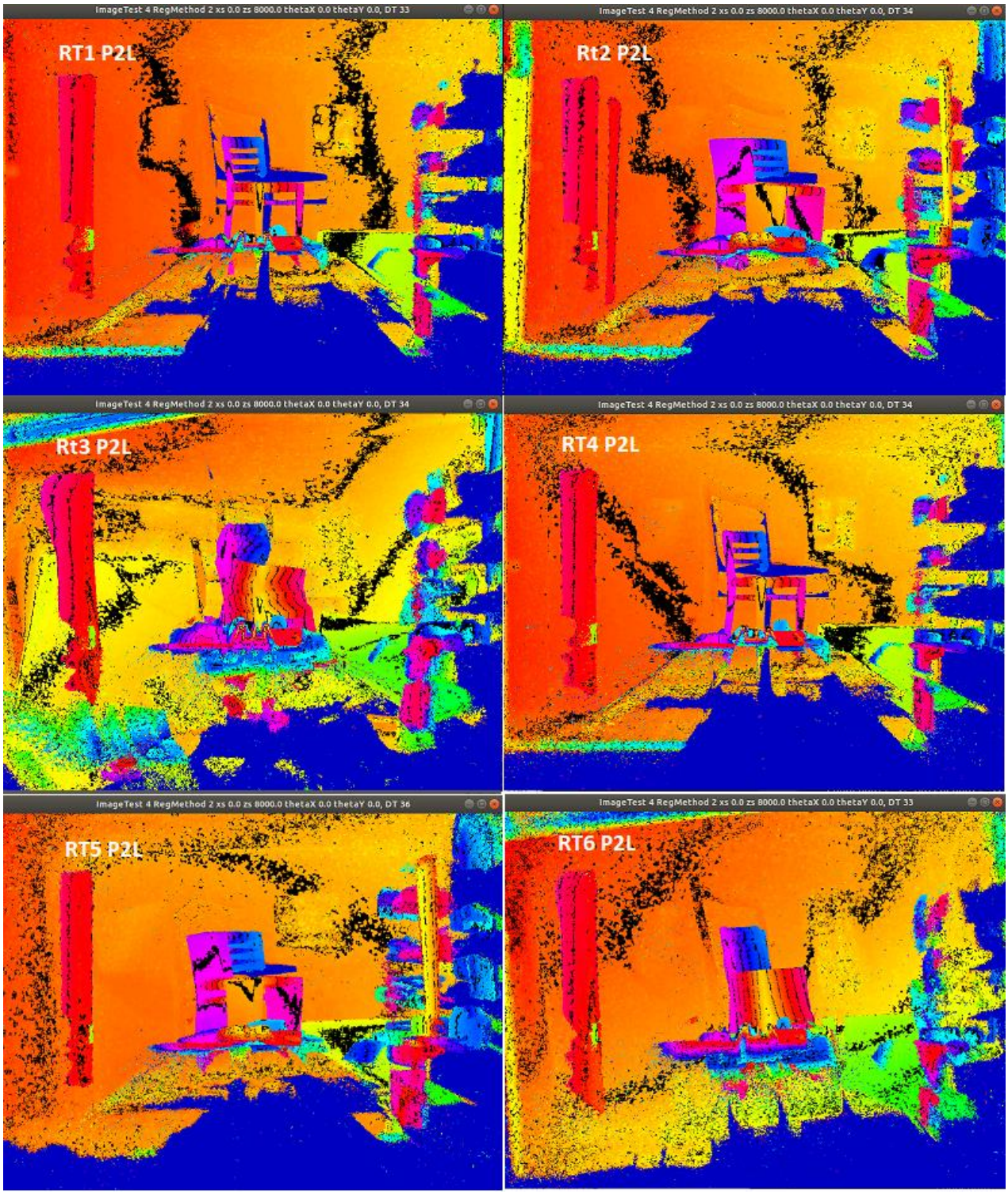


Figure 5.26 3D Model View at End of Point-to-Plane Tracking Tests

From the above results of Figure 5.26 it can be seen that tests RT1 and RT4 compare well in point-to-plane testing in comparison with the equivalents using rig data on figure 5.22. It is noticeable that they do not suffer noise to the same extent with quite clean final images. This cannot be said for any of the rotation cases RT2 and RT5 both show lateral distortion during yaw testing, whilst RT3 and RT6 show combinations of extrusions during pitch testing. Tracking results for test RT4 are shown on figure 5.27 below.

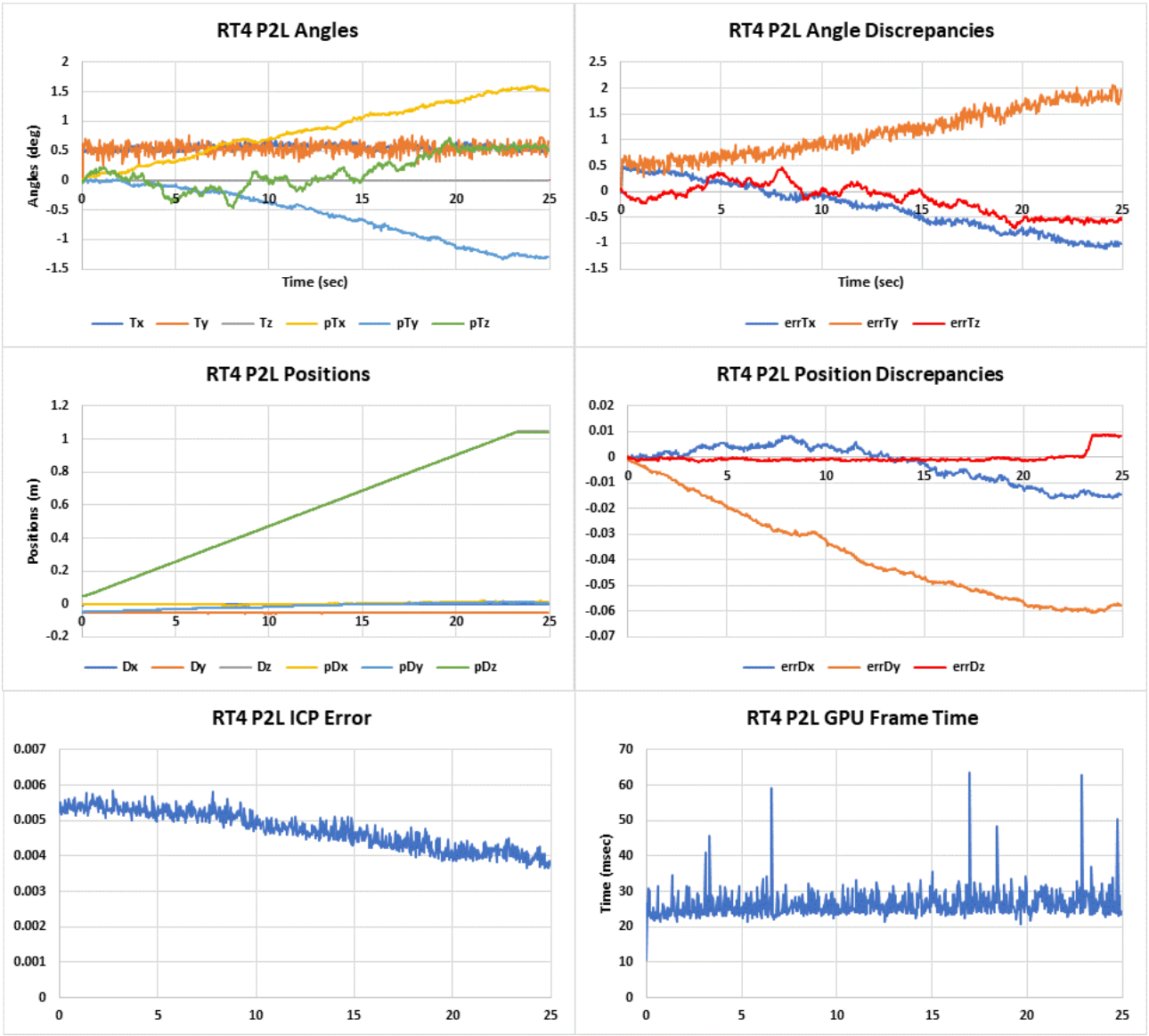


Figure 5.27 Point-to-Plane Tracking Results for Test RT4

This slow, 25 second traverse, shows good overall tracking performance. There is obvious evidence of drift on Pitch (pTx, errTx) and Yaw (pTy, errTy) but excellent tracking of longitudinal motion (Dz, errDz) though rather a lot (60mm) of lateral drift (Dy, errDy). The GPU frame time has a mean of 26.4 milliseconds, well within the target 33.3 milliseconds for a 30Hz frame rate, though it also shows the occasional dropped frame (time ~60msec). ICP Error is included only as an indicator of algorithm health, it

does not relate closely to computed tracking accuracy, though it often indicates problems if it is excessive. For comparison with later results the poor tracking response for the challenging test RT3 is show on figure 5.28 below.

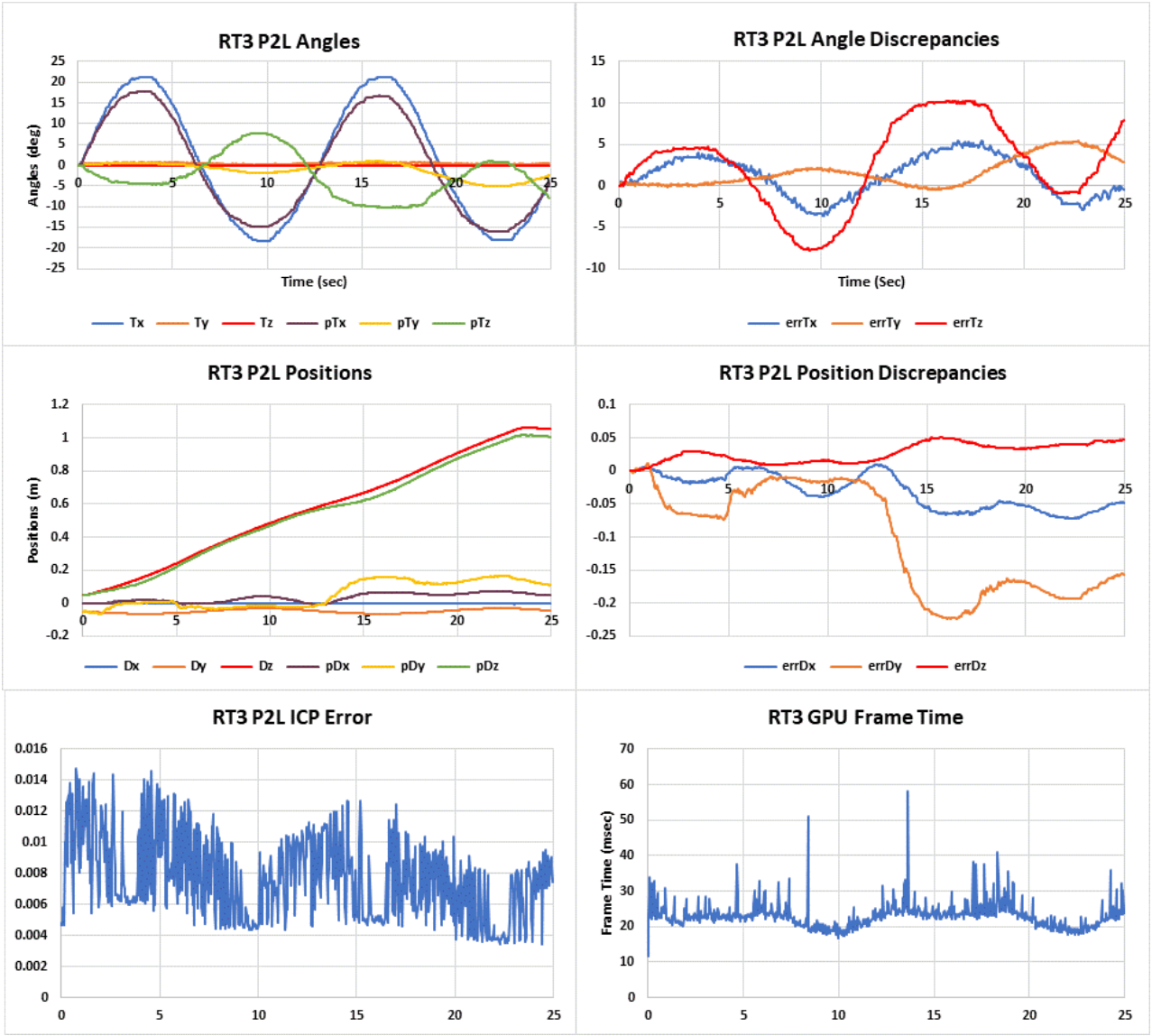


Figure 5.28 Point-to-Plane Tracking Results Test RT3

As can be seen this test fails in every way with poor tracking of pitch (pTx, errTx), roll (pTz, errTz) and shows a very large (0.2m) excursion in the vertical direction (pDy, errDy).

To investigate the effect of deferred reference frame update and bilateral filtering on the above test the following test shown on figure 5.29 was repeated with the following parameters:

- CellX = 5
- ICPFrames = 5 Reference Updated delayed 5 frames
- PreFilter = 1 (Bilateral Filter On, parameters are as given on Figure 5.1)

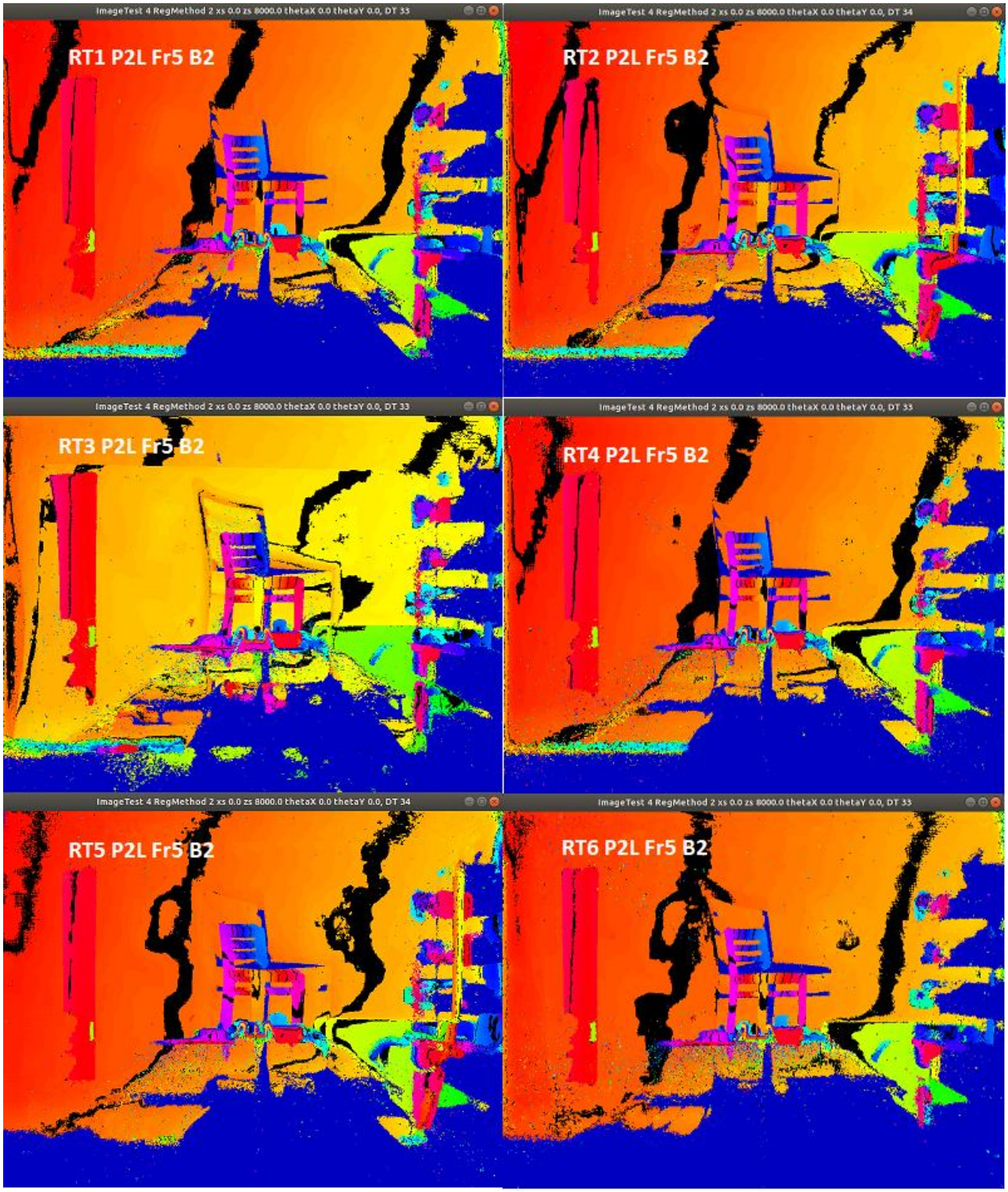


Figure 5.29 3D Model View at End of Point-to-Plane Test with Deferred Update and Bilateral Filter

Whilst the above results are by no means perfect, they do show a marked improvement over the base test case shown on figure 5.26. The chair remains recognisably a chair with no massive extrusions though it is clearly distorted most notably in tests RT3 and RT5. The tracking results for test RT3 are shown on figure 5.30 below.

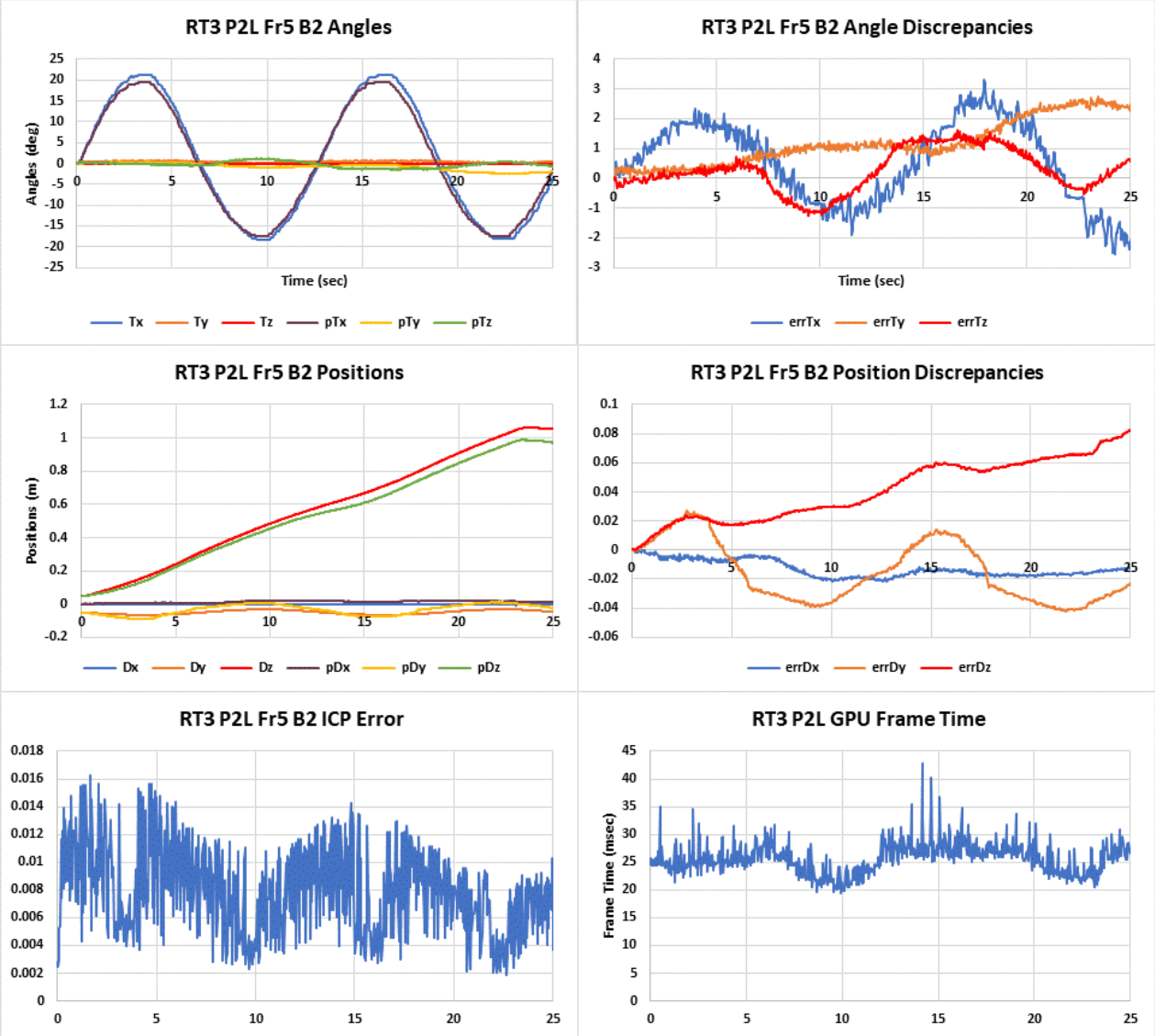


Figure 5.30 Point-to-Plane Tracking Results Test RT3 with Deferred Update and Bilateral Filtering

Whilst tracking of pitch (pTx, errTx) and roll (pTz, errTz) errors have both significantly reduced, tracking of position (pDz, errDz) has actually degraded from 50mm without additions to 80mm with. Thus, although the final 3D model image and tracking has improved significantly in many respects, it still falls short of what would be desired of a depth-camera-based tracking system.

The following table 5.5 shows absolute maximum tracking errors for each test for the Point-to-Plane algorithm with deferred reference and bilateral filtering.

Test	Servo Input	errTx (deg)	errTy (deg)	errTz (deg)	errDx (mm)	errDy (mm)	errDz (mm)
RT1	None	0.98	1.32	0.50	17.64	14.98	6.79
RT2	10 deg Yaw	1.02	1.52	0.95	15.16	12.01	24.94
RT3	20 deg Pitch	**3.314**	**2.73178**	**1.59383**	**21.96**	**42.45**	**84.04**
RT4	None	1.03	1.80	0.55	23.08	13.39	8.76
RT5	15 deg Yaw	0.83	2.28	1.08	22.29	17.92	16.11
RT6	15 deg Pitch	**2.06494**	**2.04345**	**1.87773**	**31.59**	**22.09**	**12.44**

Table 5.5 Showing Maximum Absolute Errors for Point-to-Plane Testing

From the above it is clear that pitch input tests represent the most challenging tracking cases. One possible explanation is that the camera vertical field of view (55 degrees) is less than the horizontal (70 degrees), since pitch motion will displace the camera image vertically, each pitch increment will result in a bigger fraction of pixel occlusions between frames than would be the case for an equal increment in the yaw (horizontal) axis.

Test case RT3 is clearly very challenging due to the large +/-20-degree pitch input combined with motion along the track. To illustrate this more clearly the following image figure 5.31 is a composite of 15 depth camera images taken at intervals during the RT3 test. The sequence reads from left to right and top to bottom starting and finishing with the first and last frames respectively.

From static images it is difficult to appreciate how much the depth images are changing frame-to-frame and although both the starting image and final image show the chair, there are many intermediate frames with very little of the subject in view. Tracking from depth camera image data in real-time at 30Hz is seen to be quite challenging.

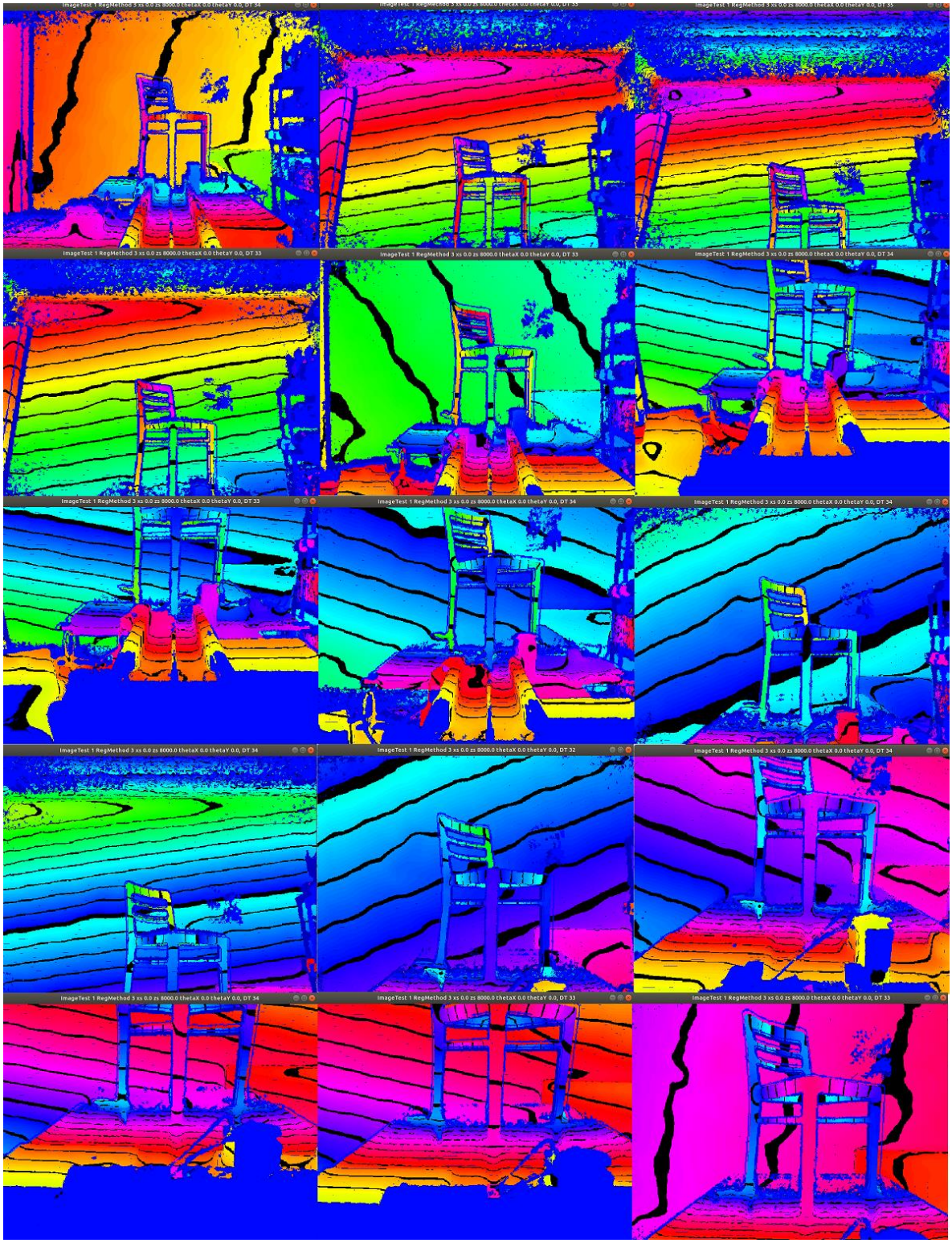


Figure 5.31 Composite of Depth camera Images taken During the Test RT3

5.4.3 Cell Search Tracking in Real-Time

The following figure 5.32 shows results for Cell-Search Tracking tests for:

- CellX=5
- ICPFrames=5
- PreFilter=0
- FMThresh = 2.

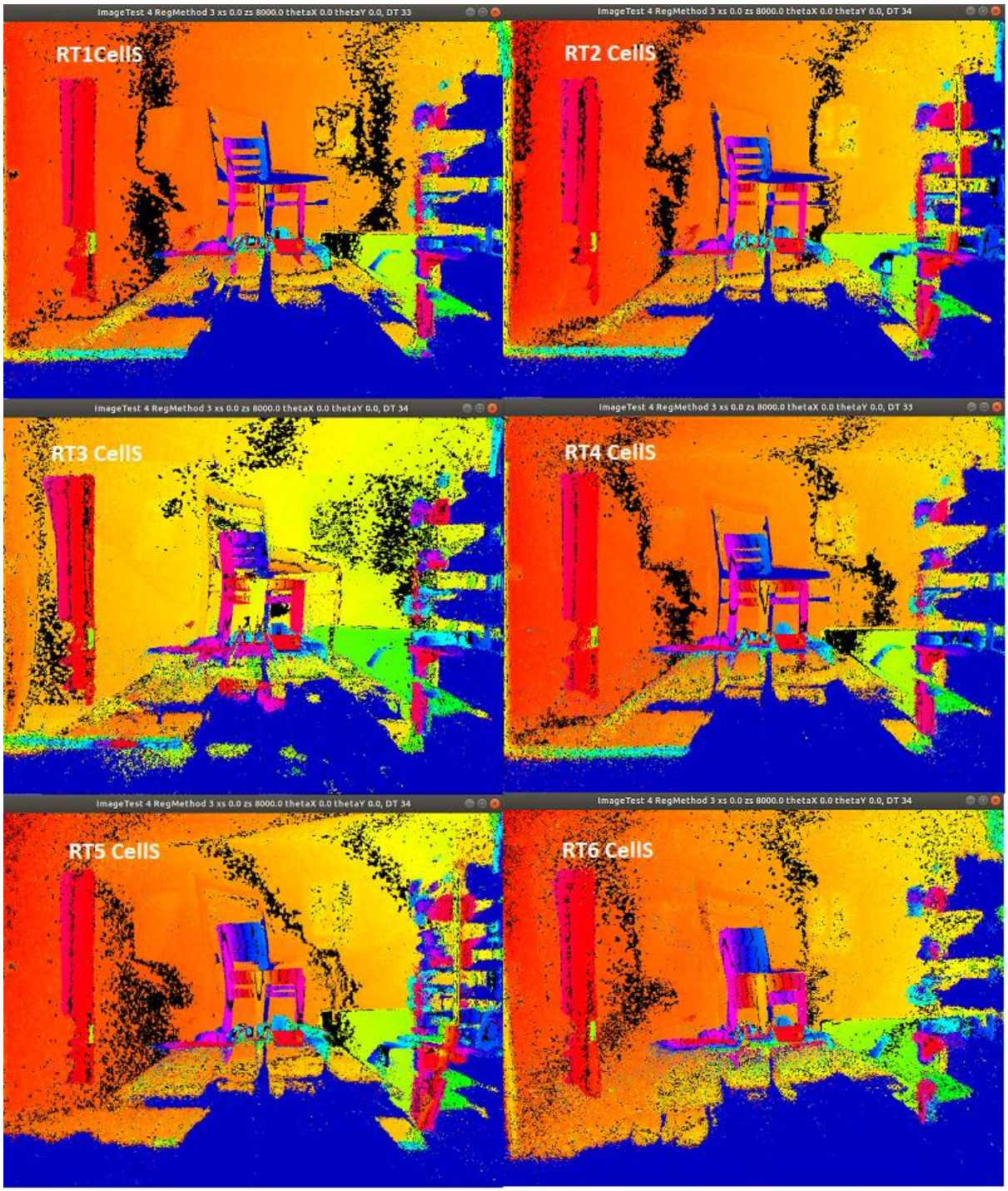


Figure 5.32 3D Model at End of Cell Search Tracking Tests with Deferred Reference Update

The Cell Search method clearly improves tracking response to camera rotations when compared to P2L (in all cases except RT4 shown on figure 5.32 above). However, it is also clear that tracking in tests RT3(Pitch), RT5(Yaw) and RT6(Pitch) are sub-optimal with visible extrusions of the chair in all cases. Cell Search tracking results are shown on figure 5.33 below for the Yaw input case.

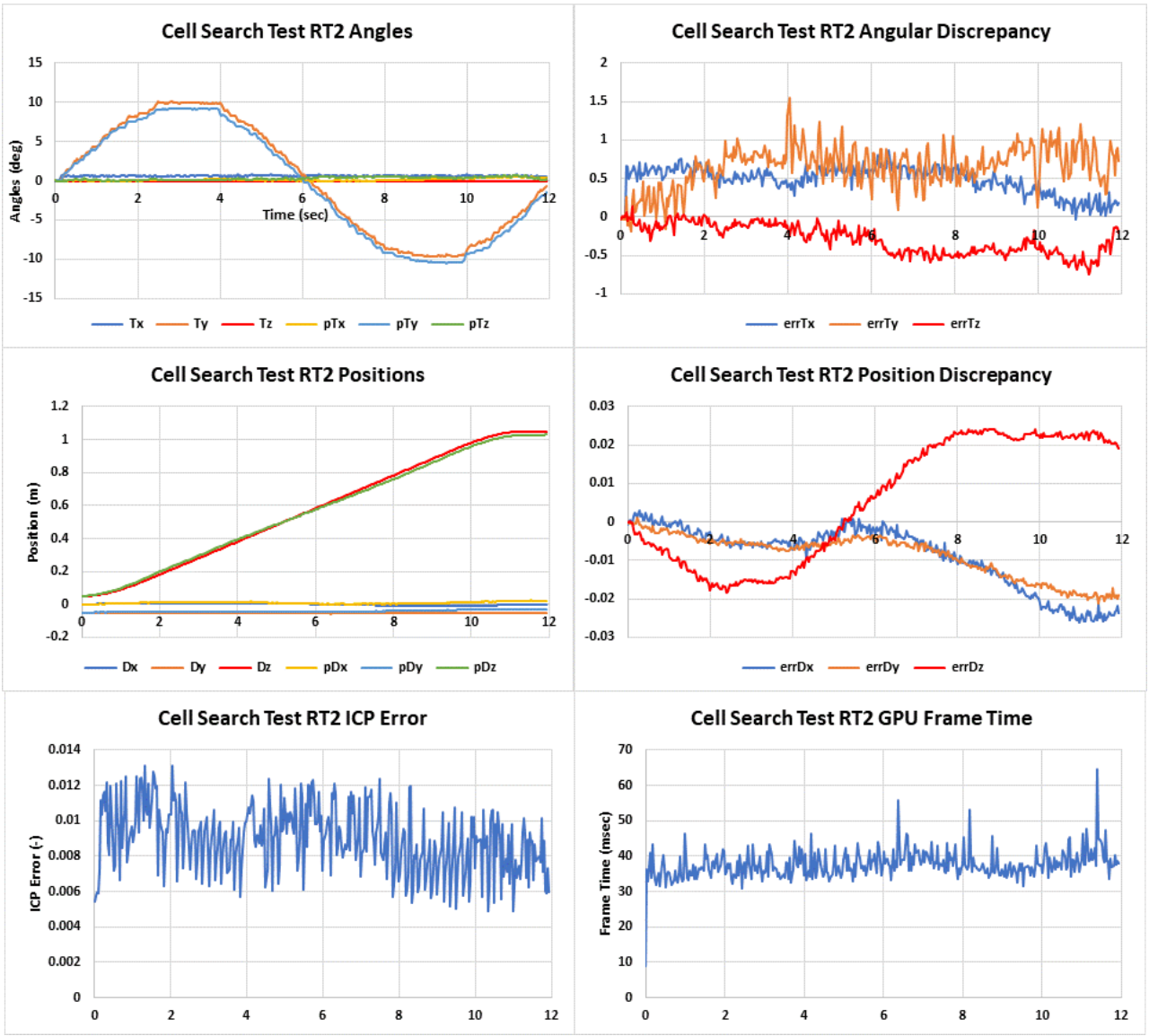


Figure 5.33 Cell Search Tracking Results for Case RT2

The mechanism by which the Cell Search algorithm works is essentially the same as Point-to-Plane but with a more exacting point-pair-matching process. In the above test the total unfiltered cell points per frame was 12286, the culled (null) points reduced this to 9400 (76% of total). This was subsequently further reduced to a PCA-point-pair matched set of 5900 (48% of total). Application of the Forstner-Moonen (FM) metric further reduced this to a set of just 660 FM-pair-matched points, 11% of the PCA-pair-matched set (5% of total). This illustrates the selectivity of the FM metric. It can be pushed further by reducing the FM threshold, but robustness then becomes a concern due to lack of data-points

and it was found that this can cause a brittle failure mode. Figure 5.33 above also shows a mean GPU processing time of 37.6 milliseconds, somewhat shy of the target 33.3 milliseconds. This method has a higher workload than vanilla P2L method due to the GPU-thread-level, Eigen-value-processing required to compute the FM metric for each matched cell. Figure 5.34 below shows the 3D image results for cell search when bilateral filtering is also employed PreFilter=1.

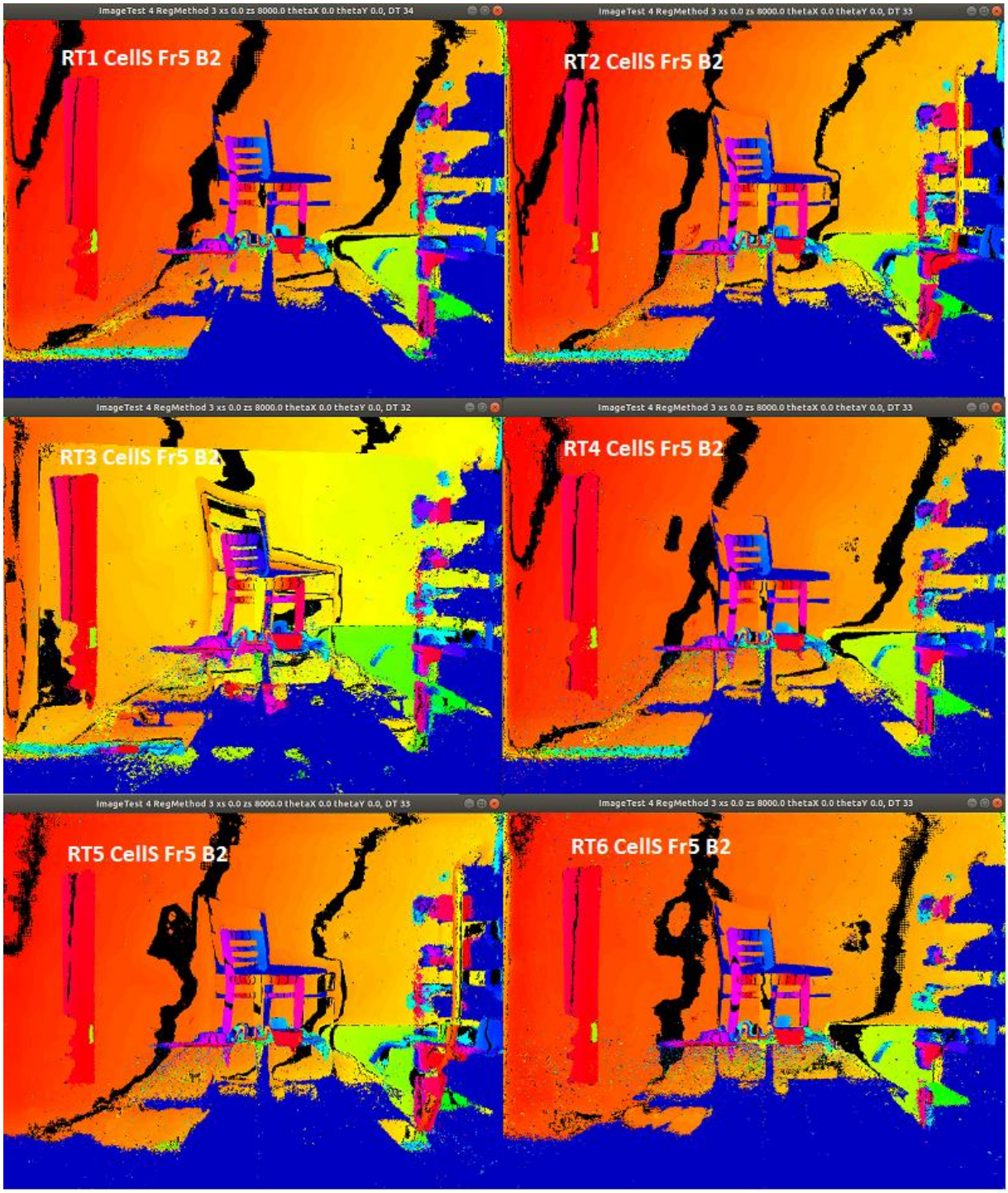


Figure 5.34 3D Model for Cell Search Tests with Deferred Update and Bilateral Filter

As can be seen from figure 5.34 the addition of bilateral filtering significantly improves 3D model image for tests RT3, RT5 and RT6 in comparison with the unfiltered case on figure 5.32. It also improves overall image quality and removes a lot of speckling most noticeable on the back wall of the unfiltered images, however this is largely cosmetic. Tracking results for test RT3 are shown on figure 5.35 below.

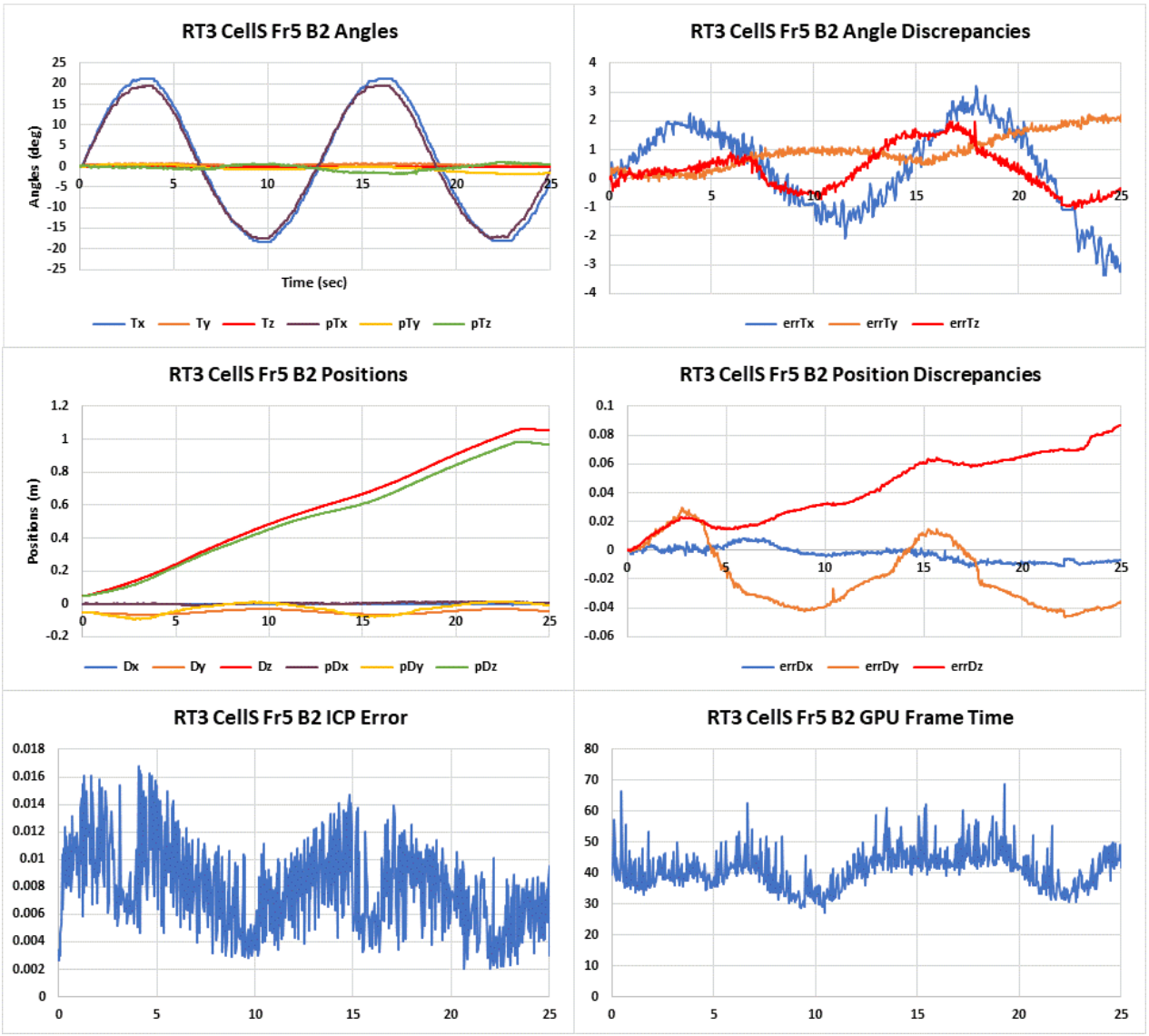


Figure 5.35 Cell Search Tracking Test Results for Test RT3

This test compares most directly with figure 5.30 for the same case when using point-to-plane. It is not notably improved in fact pitch tracking (pTx, errTx) is slightly worse as is position tracking (pDz, errDz). The most noticeable difference is the GPU Frame Time which has a mean of 41 milliseconds now well in excess of that required for a 30Hz frame update. However figure 5.36 below shows tracking results for test RT6 with bilateral filtering and deferred update.

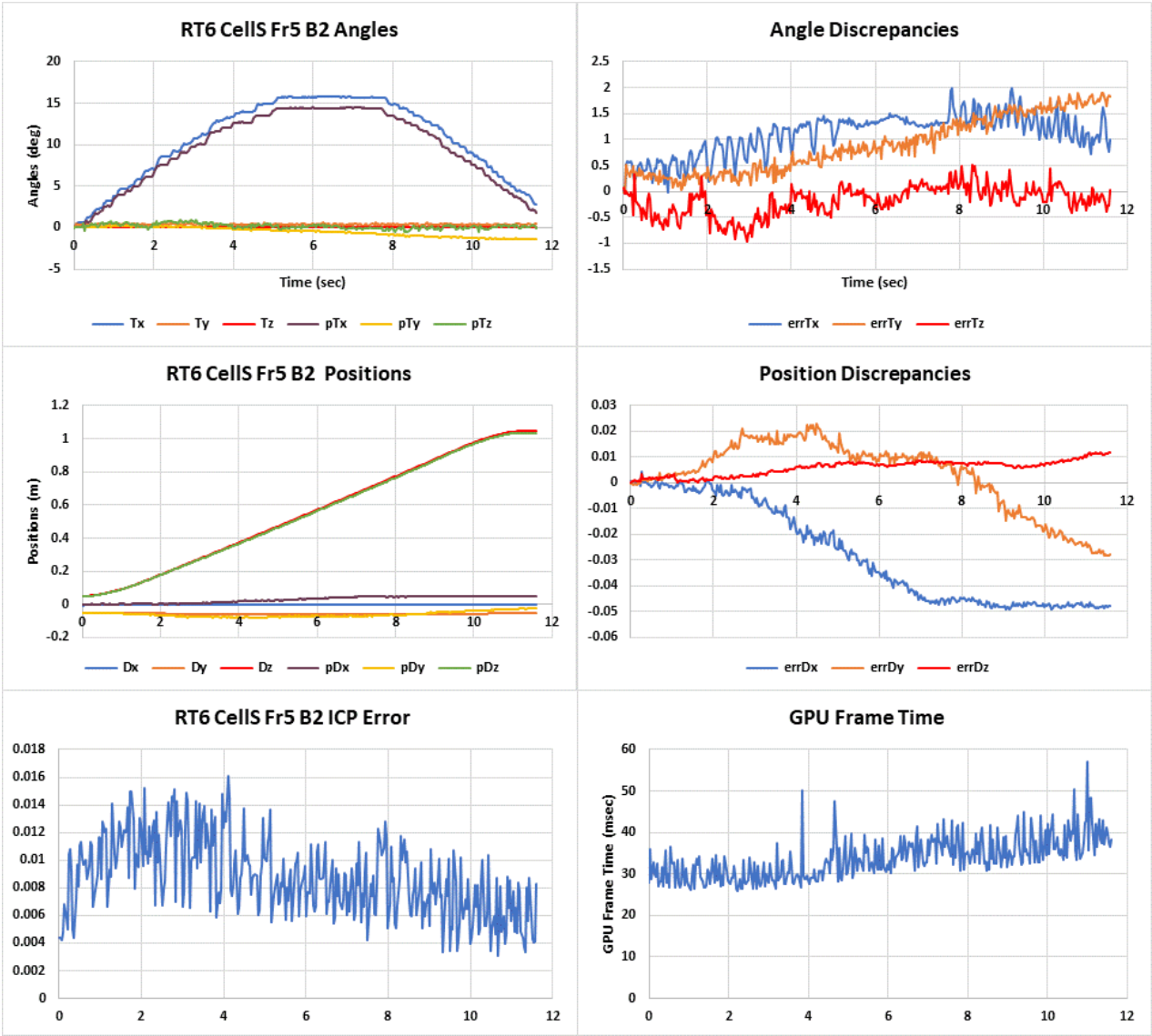


Figure 5.36 Tracking Test RT6 Results for Cell Search with Deferred Update and Bilateral Filtering

As can be seen from figure 5.36 above Pitch tracking (pTx, errTx) is typically 1 to 1.5 degrees in error. Tracking of carriage travel (pDz, errDz) is very good but a 50mm lateral error (pDx, errDx) is far from optimal. However, there are very few means by which further improvement of these open loop tests can be obtained.

Table 5.6 below shows a summary of absolute maximum errors for the Cell Search Tests.

Test	Servo Input	Pitch (Tx)	Yaw (Ty)	Roll (Tz)	Sway (Dx)	Heave (Dy)	Surge (Dz)
RT1	None	1.1274	**1.8015**	0.5557	0.01366	0.00632	0.0083
RT2	10 deg Yaw	1.26207	1.45012	1.05537	0.02415	0.00874	0.02573
RT3	20 deg Pitch	**3.3774**	**2.2364**	**1.98583**	0.01146	**0.04628**	**0.08875**
RT4	None	1.21097	**1.73829**	0.75456	**0.02724**	0.02083	0.01184
RT5	15 deg Yaw	1.2328	**1.72696**	0.87185	0.01849	0.00712	0.01605
RT6	15 deg Pitch	**1.99188**	**1.8981**	0.96857	**0.04926**	0.02828	0.01183

Table 5.6 Cell Search Test Absolute Errors for Real-Time Tests

Before embarking on this phase of testing notional accuracies of less than 1.5-degree error on any angle and less than 25mm on any position were seen as reasonable targets. The bold items in table 5.6 above show that these notional accuracy targets are often exceeded.

It is also notable that computed pitch and yaw responses tend to lesser amplitudes than the sensors indicated and yet the small-amplitude variations in response seen on the same sensors is match very closely by that given on the computed tracking responses. Thus, there seems to be quite good correspondence at small-amplitude but not on the large-amplitude rotations. This curious behaviour is not understood, the following are some speculations as to the possible cause and potential resolution.

1. The ServoTrack sensors are not calibrated correctly. These sensors were independently calibrated prior to testing, the results of this work are given in Appendix B.
2. The camera contains nonlinear lens-based angular distortions. This has been checked and found not to be the case.
3. There is a mistake in the large angle equations. The equations used are standard and as documented in section 2.3.5. Any mistake would likely lead to spurious cross-couplings.
4. The delta-transformation obtained at each Point-to-Plane step is based on the small angle assumptions. This delta-transform is then accumulated into the large angle global transformation. This could be a mechanism for loosing large angle information in the global transform.
5. The tracking algorithm is less accurate at large angle displacements due to occlusion at the leading and trailing edges of registration images. This effect is observed and discussed in ServoTruck Testing in section 5.5.3. Small amplitude deviations do not suffer this effect. This point is much harder to prove or dismiss but is nevertheless suspected to be part of the cause of the observed behaviour.

5.5 ServoTruck Testing

Whereas the ServoTrack test rig enabled verification of positioning algorithm outputs against the test rig sensors, ServoTruck offers no such facilities. There is the possibility of rudimentary calibration of the Radio-control signals for speed and turning angles and with additional vehicle modelling, the possibility of benchmarking one algorithm against another. There are no absolute position or orientation sensors onboard the truck. Consequently, results can only be quantitatively evaluated against the developed 3D model. For as has been found with the ServoTrack rig, there is a good correspondence between model quality and tracking results, especially so when the 3D model results are seen to be accurate.

5.5.1 ServoTruck Horizontal Testing

To give a guide to tracking quality a simple test setup, similar to ServoTrack Real-Time testing, is shown on figure 5.37 below.

Figure 5.37 ServoTruck Chair3 Test Setup

In this test scenario the truck is driven in a straight line towards chair target placed approximately 2.5 meters away. The camera data is recorded by the on-board computer. The laptop communicates with the on-board computer via SSH over WiFi and is used to start the recording program and download the data at the end of the run. Other than that, all control of the truck including steering, speed, starting and stopping recording are controlled via the RC control unit shown to the left of the laptop in figure 5.37 above.

The final 3D model results from this test are shown on the following composite images in figure 5.38.

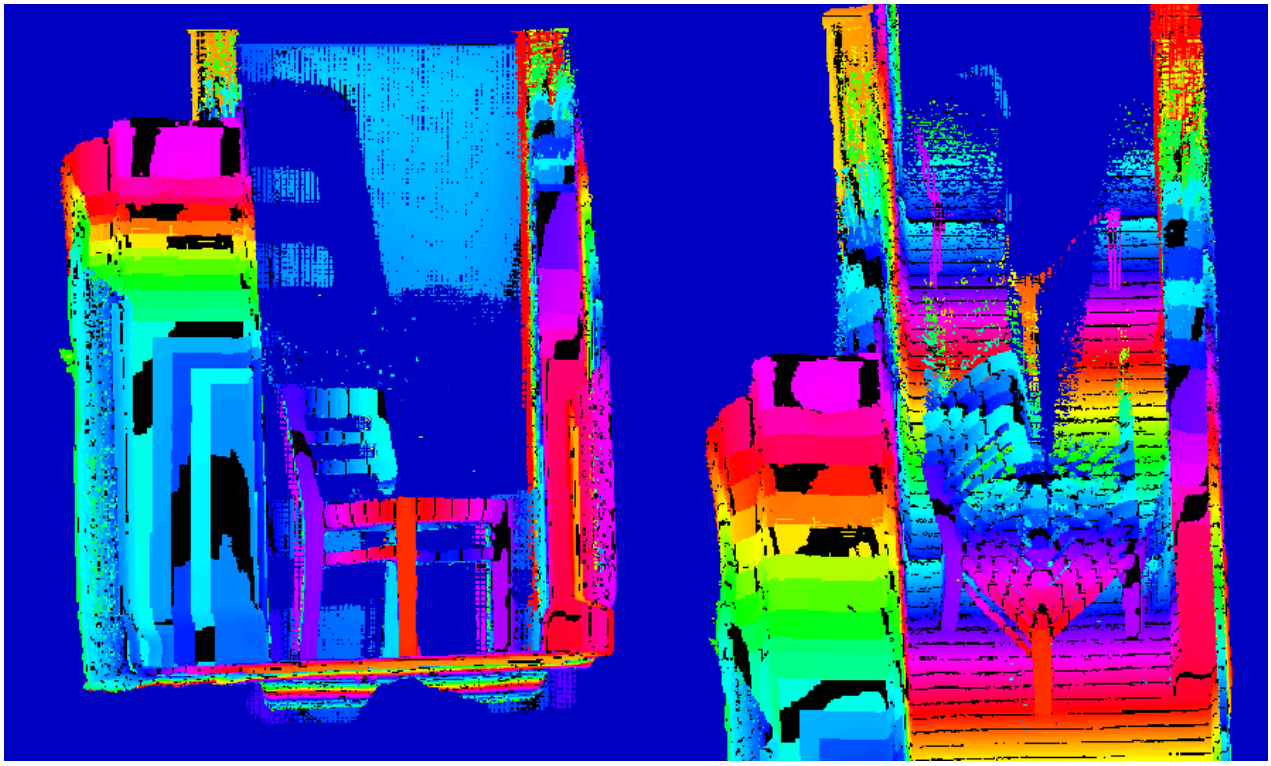


Figure 5.38 ServoTruck Chair3 Test Results Front (Left) and Top-Down (right) View

These 3D model results were obtained using the same settings that were seen to give good results for the ServoTrack rig. The left image is comparable with test result RT1 shown in the top left image of figure 5.34 above.

Clearly, these images show a good deal more distortion of the chair image. The right hand image also shows that both the wall (front left) and the chair are visibly moved along the Z axis indicating very poor position tracking. Tracking results for this test are shown on figure 5.39 below.

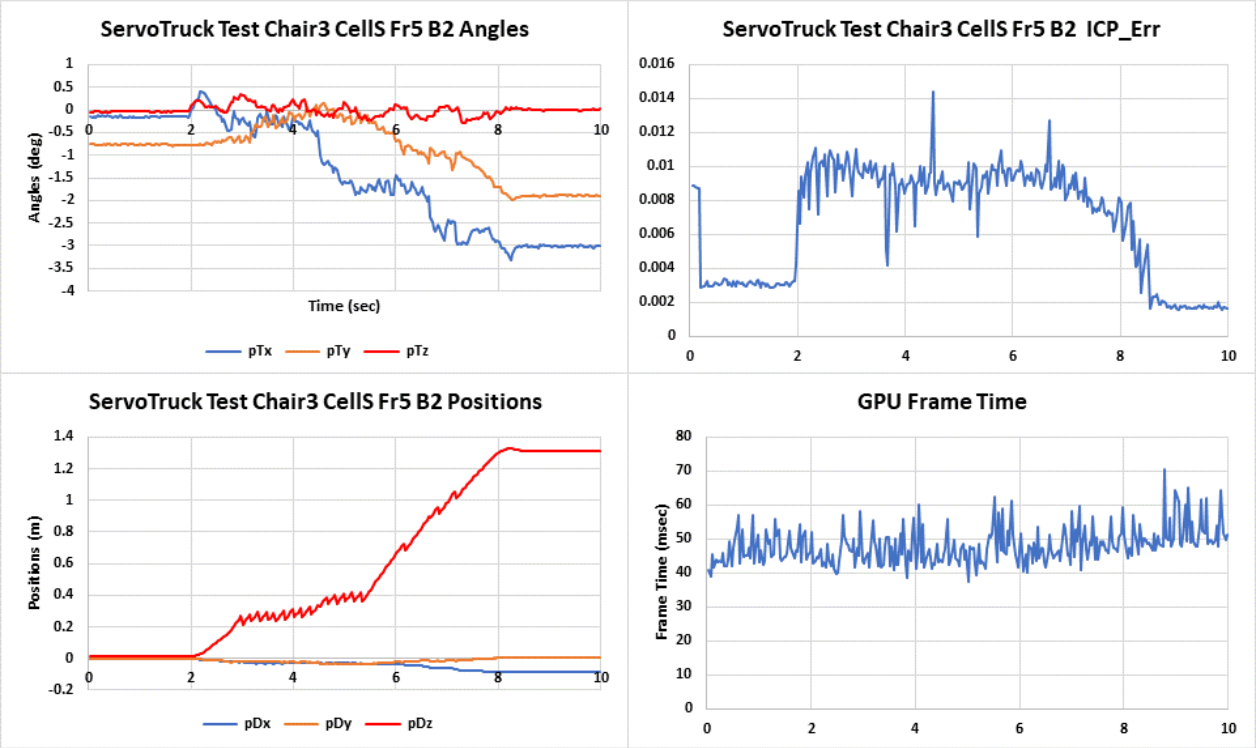


Figure 5.39 Showing Tracking Results for Chair Test3

When these tracking results were first reviewed, they were so poor that it was thought there may be a fault with the test rig. This turned out not to be the case. When parameter settings were changed the following 3D model result, figure 5.40 was obtained.

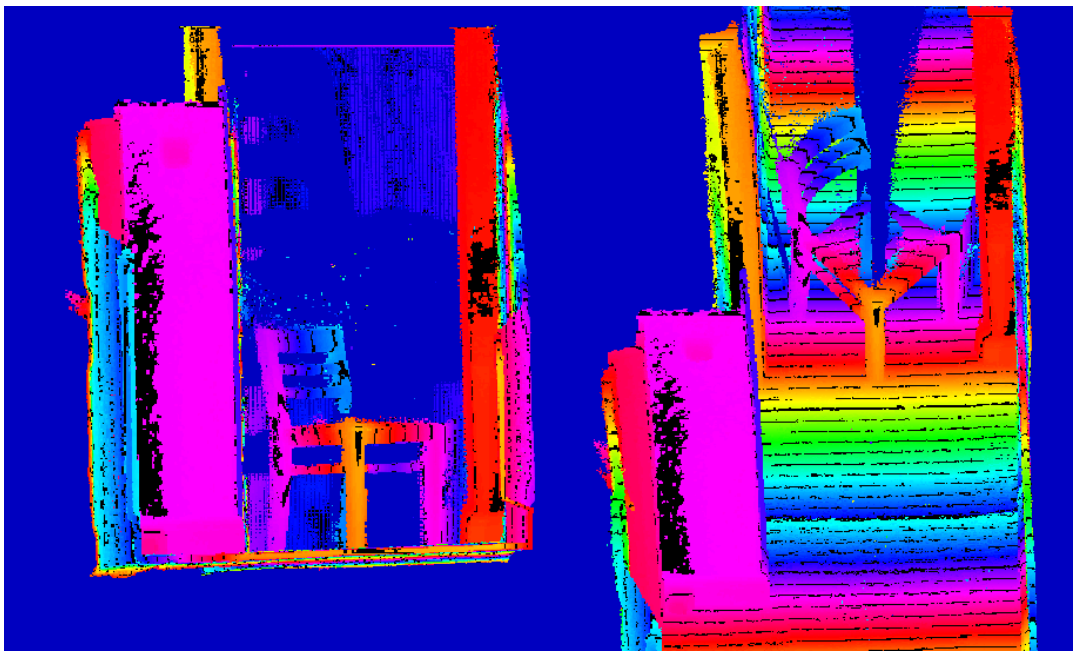


Figure 5.40 Chair 3 Test Results after Parameter Changes

Whilst not perfect, with some evident lateral extrusion of the chair legs, it is still recognisably a chair and it and the walls remain in one place. Tracking results for this test are shown on the following figure 5.41.

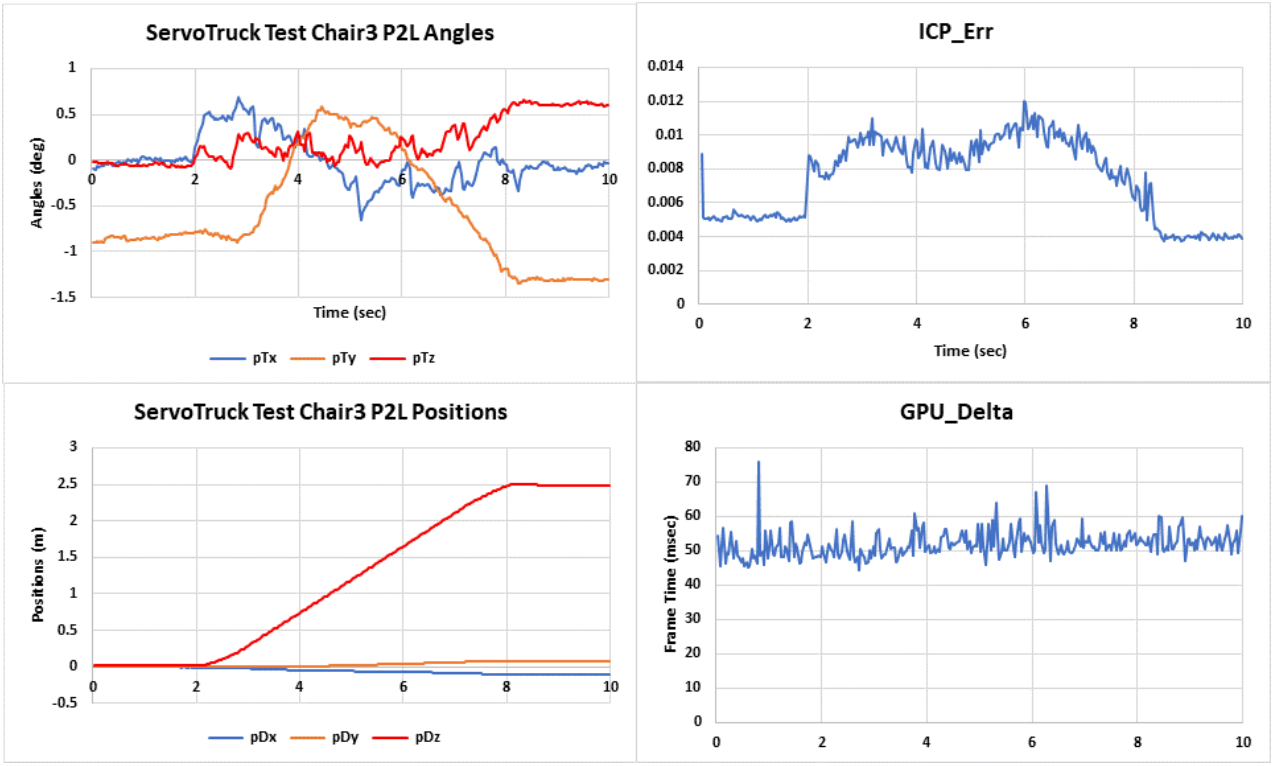


Figure 5.41 Chair 3 Tracking Results after Parameter Changes

The above results show some drift on angular orientations which could well be real given the freedom of the truck. There is a troublesome drift on lateral and vertical responses pDx and pDy and though not large are responsible for the distortions visible on the chair. ICP_Err is reasonable but sadly GPU frame time is well outside (almost double) the 33 milliseconds required for a 30Hz system.

The above results shown on Figures 5.40 and 5.41 were obtained for the standard Point-to-Plane tracking algorithm with:

- CellX=4 Cell Size
- ICPFrames=1 No delay in updating Reference Frame
- PreFilter=0 Bilateral Filter turned off

Thus, all the parameter refinements that were seen to improve tracking and 3D model quality for the ServoTrack real-time tests, were reset to baseline for these ServoTruck tests.

This seems to be a recurrent and rather troublesome theme for depth-image-based tracking. The main factors of difference between these two sets of tests are that ServoTruck is travelling about 3-4 times faster than the ServoTrack carriage. ServoTruck also has more lateral freedom. However, it is well

behaved with very light suspension, soft tyres and no obvious dynamic vibration modes of significance to the tests.

Whilst there are no direct position or orientation sensors onboard the truck, the RC control signals for motor speed and steering are recorded and enable a rudimentary estimation of tracking accuracy. A set of calibration runs at various speeds over a measured 8.5m straight run enabled an approximate calibration curve to be developed for vehicle forward speed (in m/sec) as a function of the non-dimensional RC motor control signal. This calibration curve is shown on figure 5.42 below.

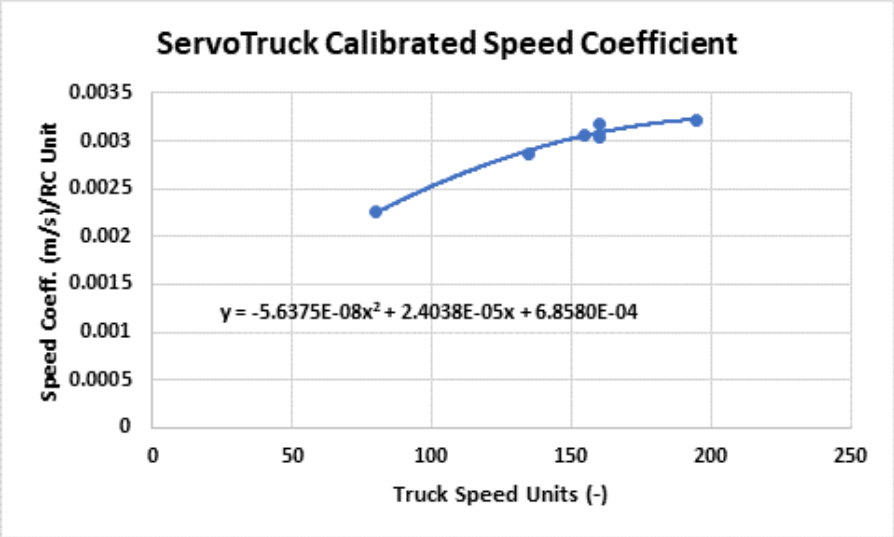


Figure 5.42 ServoTruck Calibrated Speed Coefficient

Using the above curve in figure 5.42 and the time history data for motor speed over the Chair 3 test shown in figure 5.40 and 5.41 above, a basic estimate of horizontal distance versus time was generated using Modified Euler integration. This is then compared with the P2L-computed pDz tracking result for the truck on the following figure 5.43.

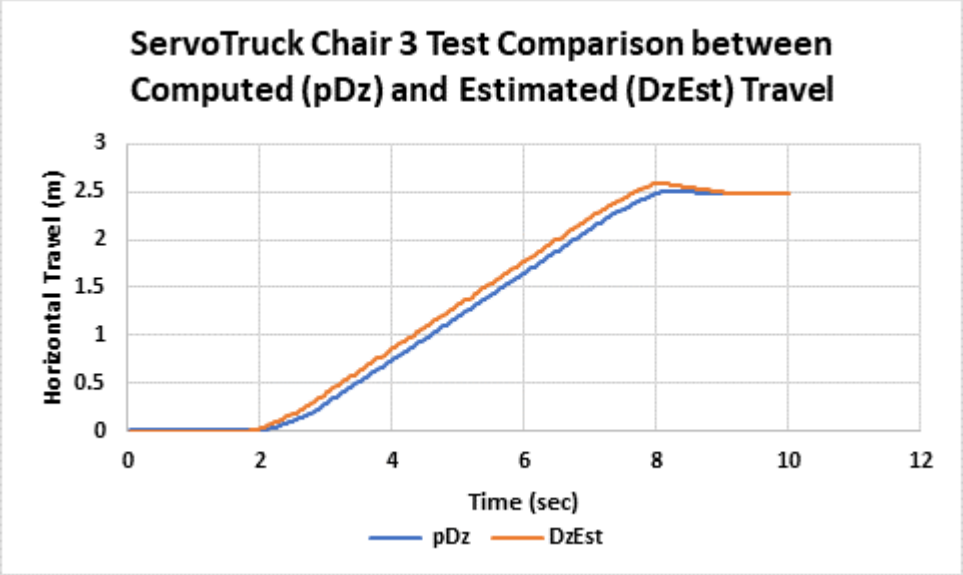


Figure 5.43 ServoTruck Chair 3 Tracking Comparison

This result is not a one-to-one correlation, so that the position estimated from the motor speed is ahead of the computed position is not surprising. What is clear is that both the slope and the final position are closely correlated. This together with the images in Figure 5.40 above, gives some support for the view that the system is now tracking horizontal position quite well.

5.5.2 ServoTruck Steering Test

Figure 5.44 below shows the resulting 3D model after a steering test where ServoTruck is commanded to turn in a tight circle in a typical domestic room environment using the Cell Search algorithm with model parameters found to be optimum for the ServoTrack real-time tests namely:

- CellX=5
- ICPFrame=4
- PreFilter=1

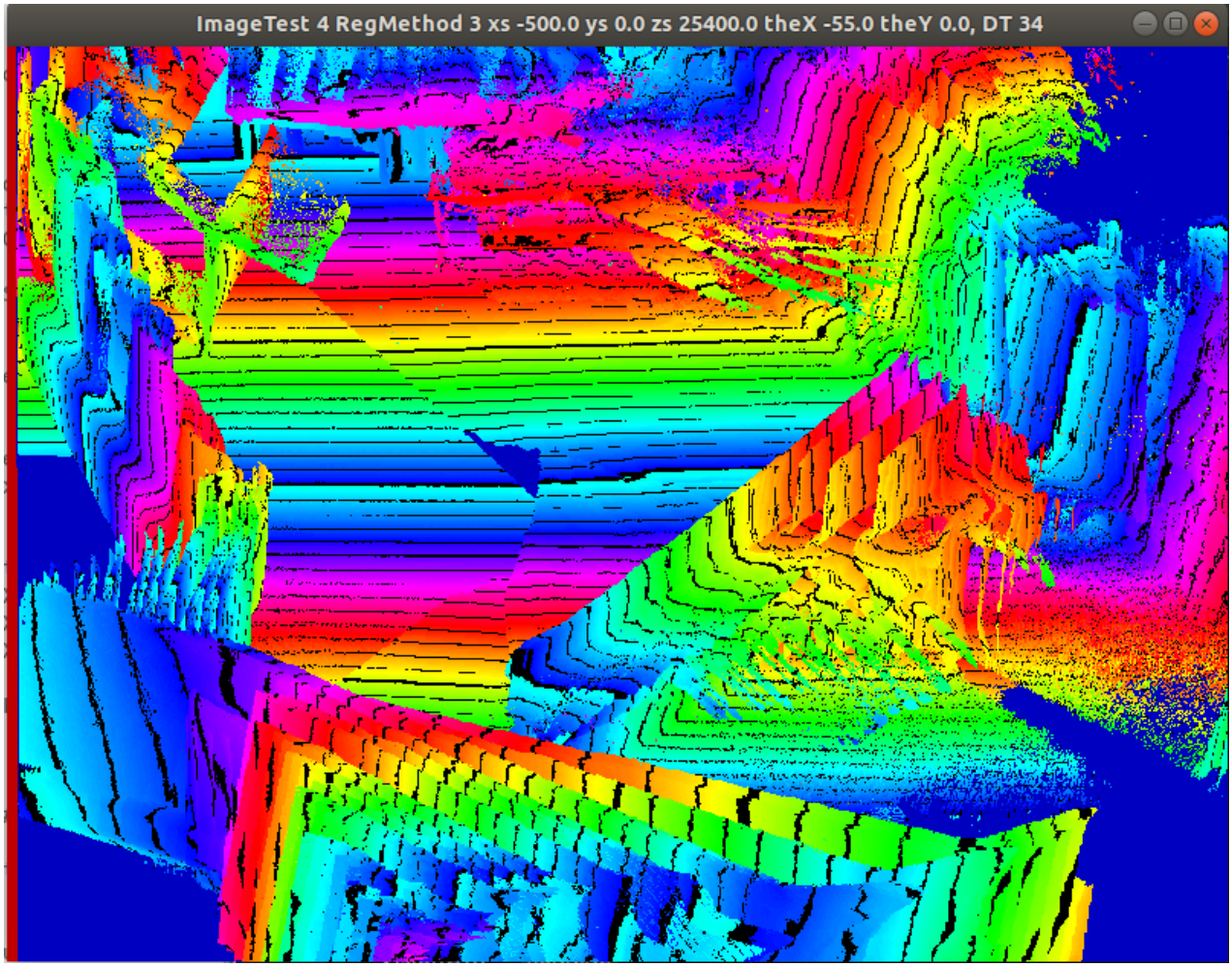


Figure 5.44 ServoTruck 3D Model Result Following Steering Test (Room1) with ServoTrack Parameters

Figure 5.44 is probably one of the authors greatest works of art, though sadly not his greatest technical achievement. The tracking response is not shown as the above image gives all that needs to be said.

ServoTruck would appear to provide another level of challenge in terms of tracking accuracy for steering tests. The above result shows that the tuning parameters that were seen to give the best results for ServoTrack give very poor results for ServoTruck in this test.

Revising the tracking algorithm parameters to those used to obtain figures 5.40 and 5.41 above:

- RegMethod=2 Point to Plane
- CellX=4 Cell Size
- ICPFrames=1 No delay in updating Reference Frame
- PreFilter=0 Bilateral Filter turned off

Re-running the above steering test resulted in the 3D model shown in Figure 5.45 below. Much of the room detail is unfortunately lost due to image scale.

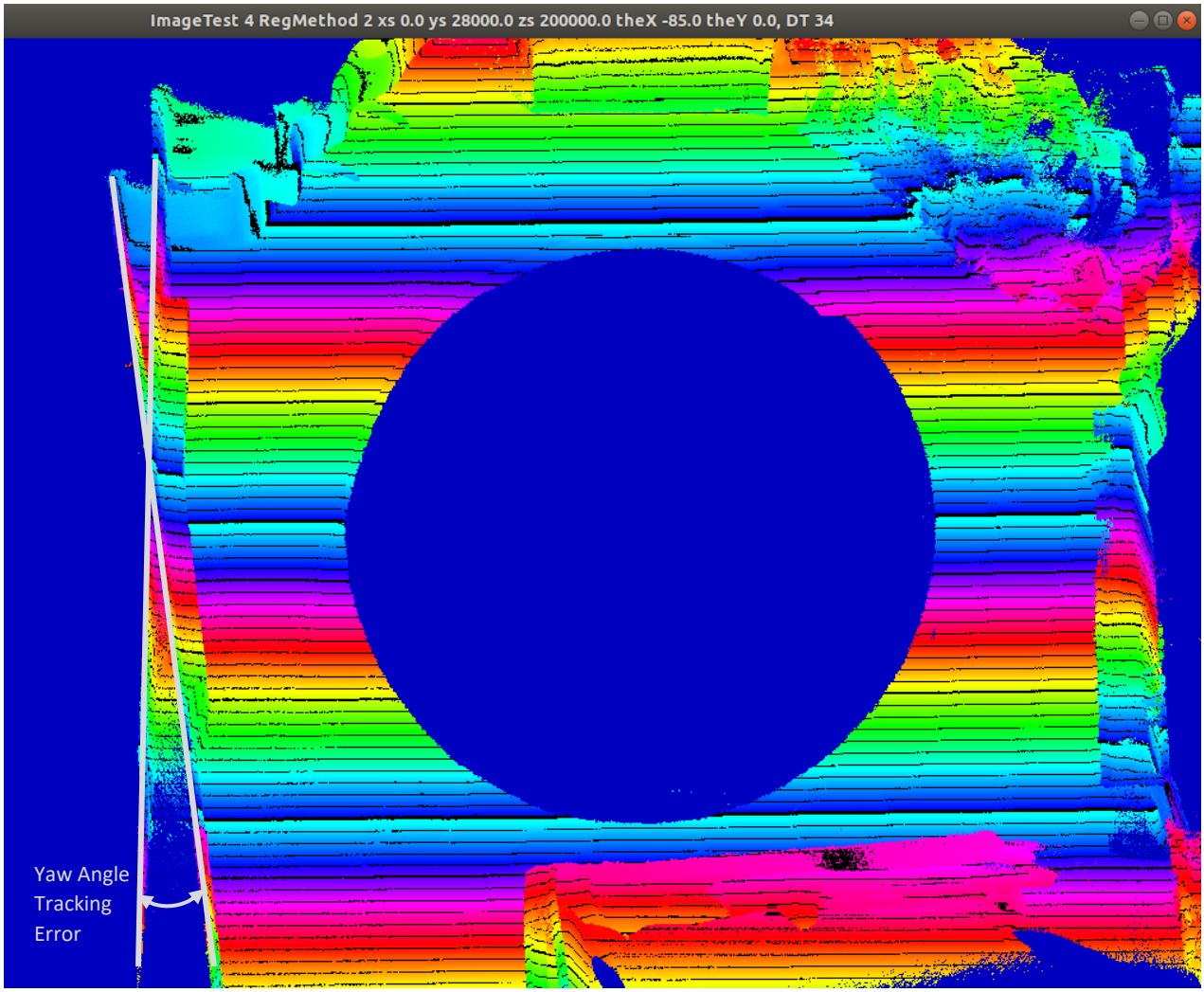


Figure 5.45 ServoTruck Room 3 Steering Test 3D Model

This is a plan view of the resulting 3D model showing the room walls and floor. The circular hole in the centre of the image is the inner part of the floor not visited by the depth camera. If tracking were perfect, the truck should retrace exactly the shape of the room. Unfortunately, as can be seen, the left-hand wall has an angular offset on the second visit relative to the first of approximately 10 degrees. The tracking results for this test are shown on the following figure 5.46 below.

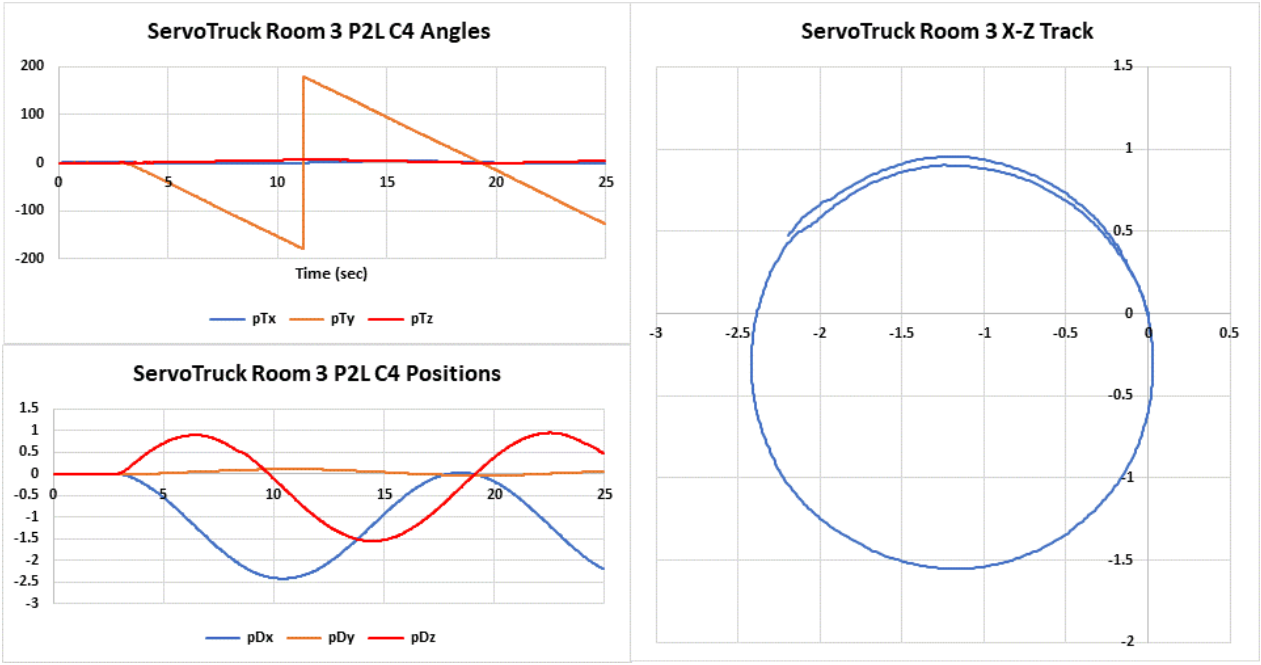


Figure 5.46 ServoTruck Room 3 Steering Test Tracking Results

There is a small drift pDy (vertical motion) and as the room is flat, this is not a real motion. Drift on pitch (pTx) and roll (pTz) are quite small and bounded. Yaw response is as expected but clearly from the 3D image in figure 5.45 above, is not correctly tracking in the same way that large angular rotations were not correctly tracked during ServoTrack testing.

5.5.3 ServoTruck Steering Test with Data Masking

The main confounding factor to good tracking using P2L (or any of the point-based methods) is believed to be none-overlap between registration images. The reason for this being that the least-squares method (on which P2L is based) will always minimise the L2 error for the two images without regard to overlap. Whilst the point-pair matching selection process is designed to exclude points not in the overlap set, this is in practice very difficult to achieve. This sensitivity to data overlap is the fundamental weakness of any of the point-based tracking algorithm.

If some of the information in regions of the two images (that should not be part of the correct overlap sub-set) match closely with some of that data in the overlap errors will be inevitable. Thus, when there is a miss-match between data in the two registration images an incorrect transformation will always result. This is true no matter how well a subset of the two images would match (if they were correctly aligned) the Point-to-Plane algorithm does not seek true alignment, but simply provides the alignment that minimises the L2 norm between the two datasets. Thus, when this data is contaminated by even a smallest amount of none-overlap data, tracking errors will inevitably result.

This, it is hypothesised, is the primary cause of the angular drift seen in the test shown on figure 5.45 above and the cause of errors during large-angle ServoTrack testing. To support this hypothesis in a simple test, the data used in the room 3 test above was deliberately rigged to mask the data in both current and reference frames to reduce, if not completely eliminate, the non-overlap regions. This data masking relies on the fact that the direction of turn of the robot is known to be anti-clockwise and at a more or less constant turn rate and that the room is basically flat. Thus, the leading edge of the current frame and the trailing edge of the reference frame are excluded from the registration process since they will not overlap. This image masking is illustrated on the following sketch.

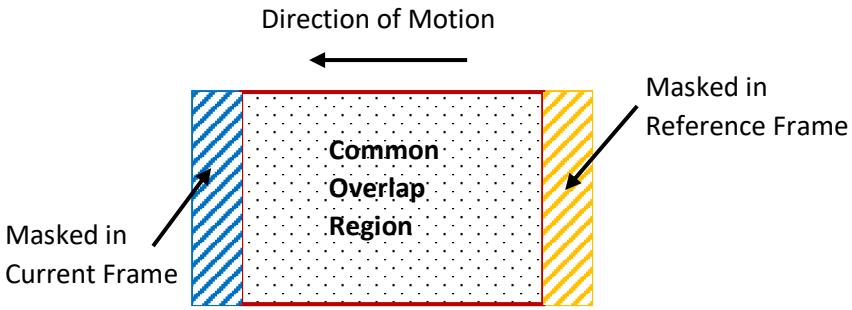


Figure 5.47 Sketch Showing Depth Image Masking Scheme used on Room 3 Retest

With this masking scheme in place, the Room3 test shown in figure 5.45 and 5.46, with the same model parameters, was re-run producing the following 3D model shown on figure 5.48.

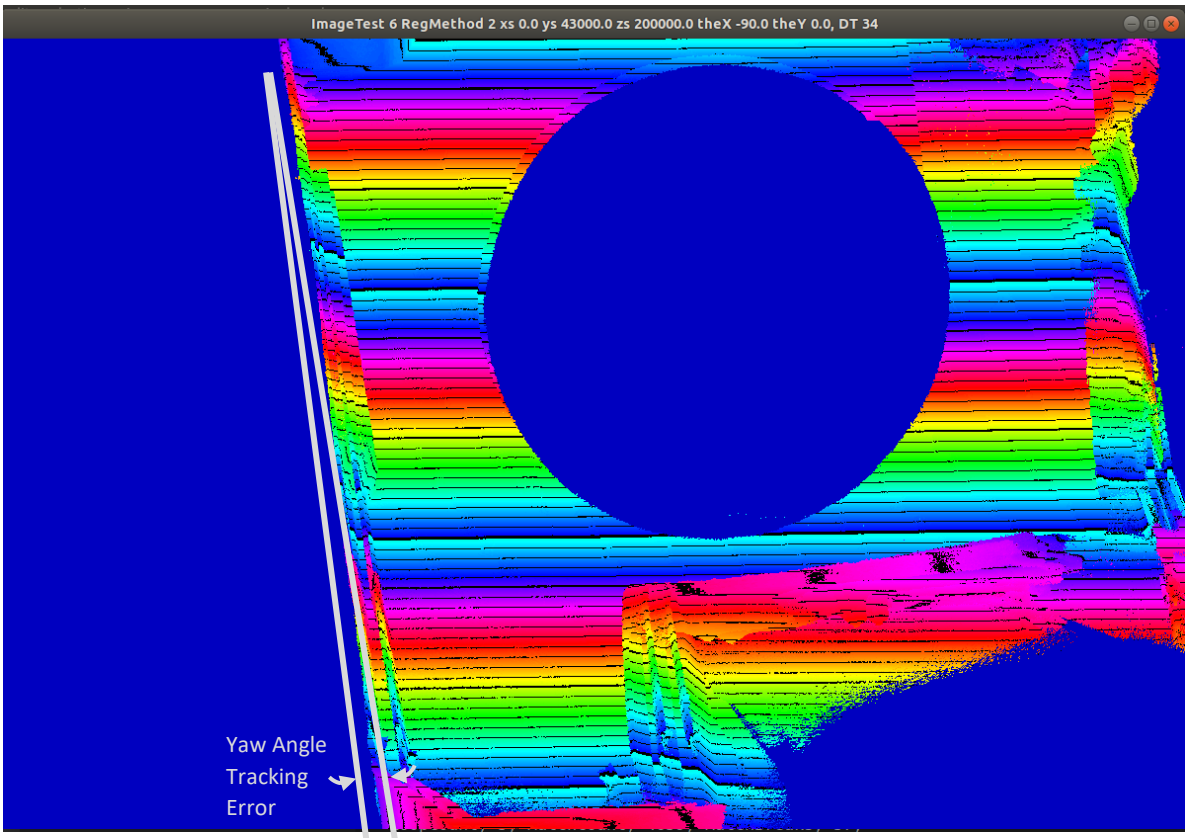


Figure 5.48 ServoTruck Room 3 Test Re-run after Image Masking

Looking at the above 3D image it is apparent that angular alignment mismatch of the left wall is reduced from the 10 degrees seen on figure 5.45, to approximately 3 degrees in figure 5.48. Thus, tracking error has been significantly reduced if not eliminated by this crude masking device. The tracking results are shown on the following figure 5.48 below.

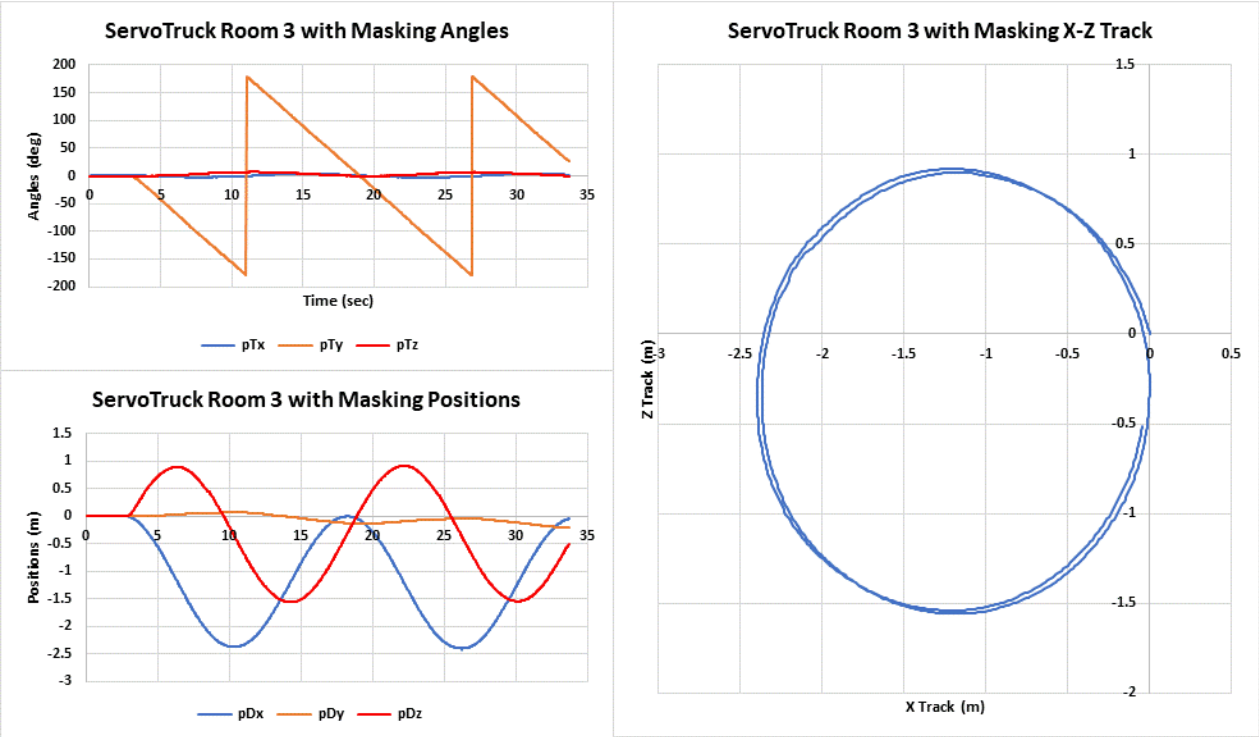


Figure 5.49 ServoTruck Room 3 Test Tracking after Image Masking

The above tracking results shows that the application of masking has improved yaw rotation tracking in the 3D model. Whilst not providing an absolute proof of the above hypothesis, this result clearly illustrates the influence of none-overlap (or image miss-match) on the P2L method. Even if two frames are overlapped spatially (as in the above masked test) there will still be regions of miss-match between the two images due to variation in occlusions due to different views of angular shapes in the scene due to camera motion.

When considering these effects, the use of point-to-plane and other point-based image matching algorithms becomes highly questionable especially in the role of robot tracking. Function minimisation based on a simple single-valued L2 norm of the two data clouds is simply too crude a mechanism to find the best tracking registration and too easily fooled by data mismatch (non-overlap) between frames. Thus, some additional means of eliminating non-overlapping data such as the Forstner-Moonen similarity metric (or alternative method) is a definite requirement for accurate tracking.

5.5.4 ServoTruck Steering Test with Cell Search

The reason for investigating this methodology was the fact that it provides another feature-based selection process on top of the basic point-to-point matching giving the point-to-plane algorithm a stronger association between data sets as they must feature-match and this should in theory remove false pair-matches from the process. The 3D model result on figure 5.50 below is for the same Room 3 test data, but in this case using the CellSearch algorithm in place of the point-to-plane method. The cell size used in this case was found to be optimum at 8x8 pixels.

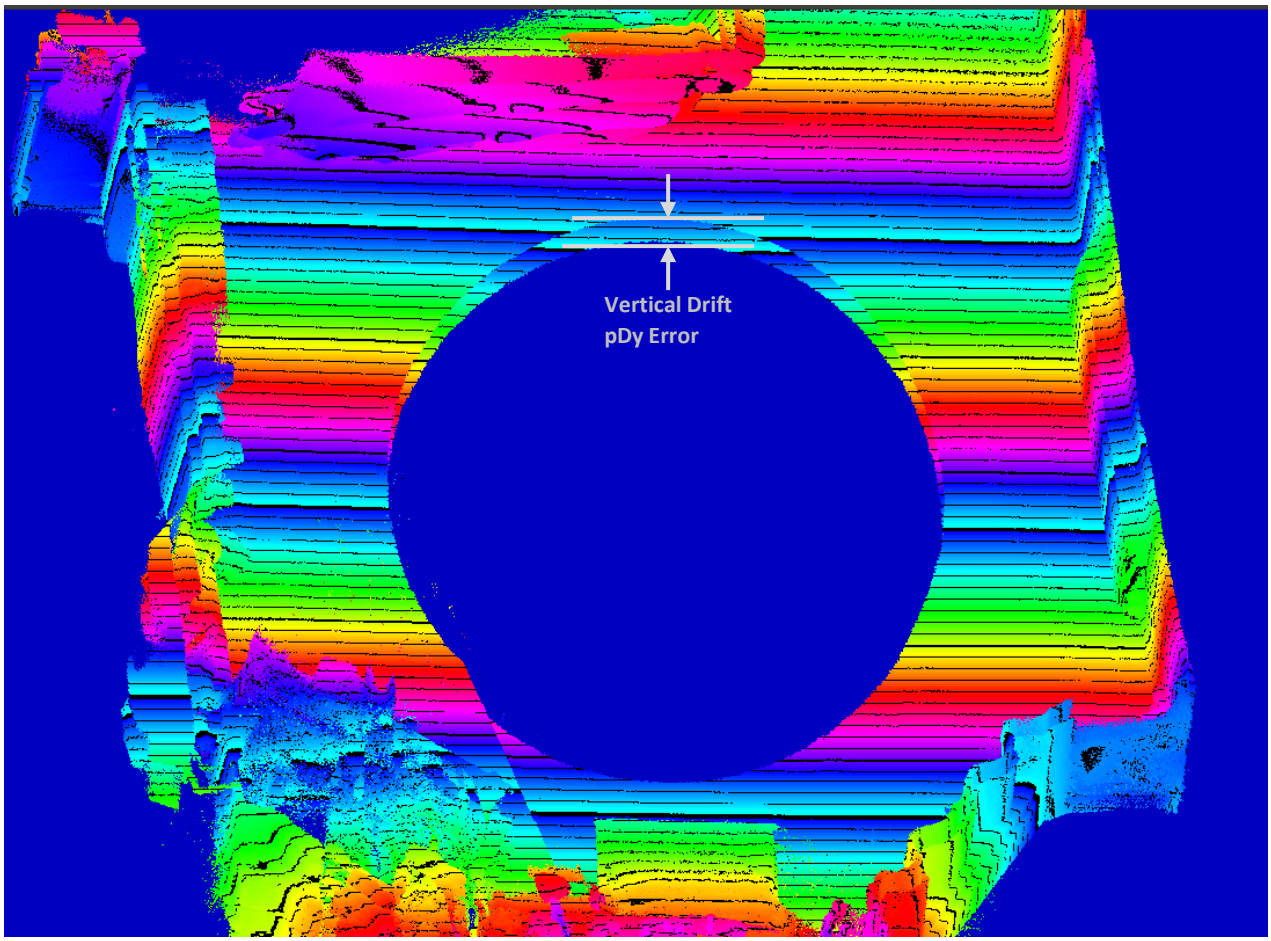


Figure 5.50 ServoTruck Room 3 Test 3D Model using CellSearch

The shape of the room is conserved (showing that yaw tracking is very good) so there is no visible angular difference of the walls after the second circuit. However, there is a visible upward drift (pDy) causing the level of the floor to rise on the second circuit. The tracking results for this test are as shown on figure 5.51 below.

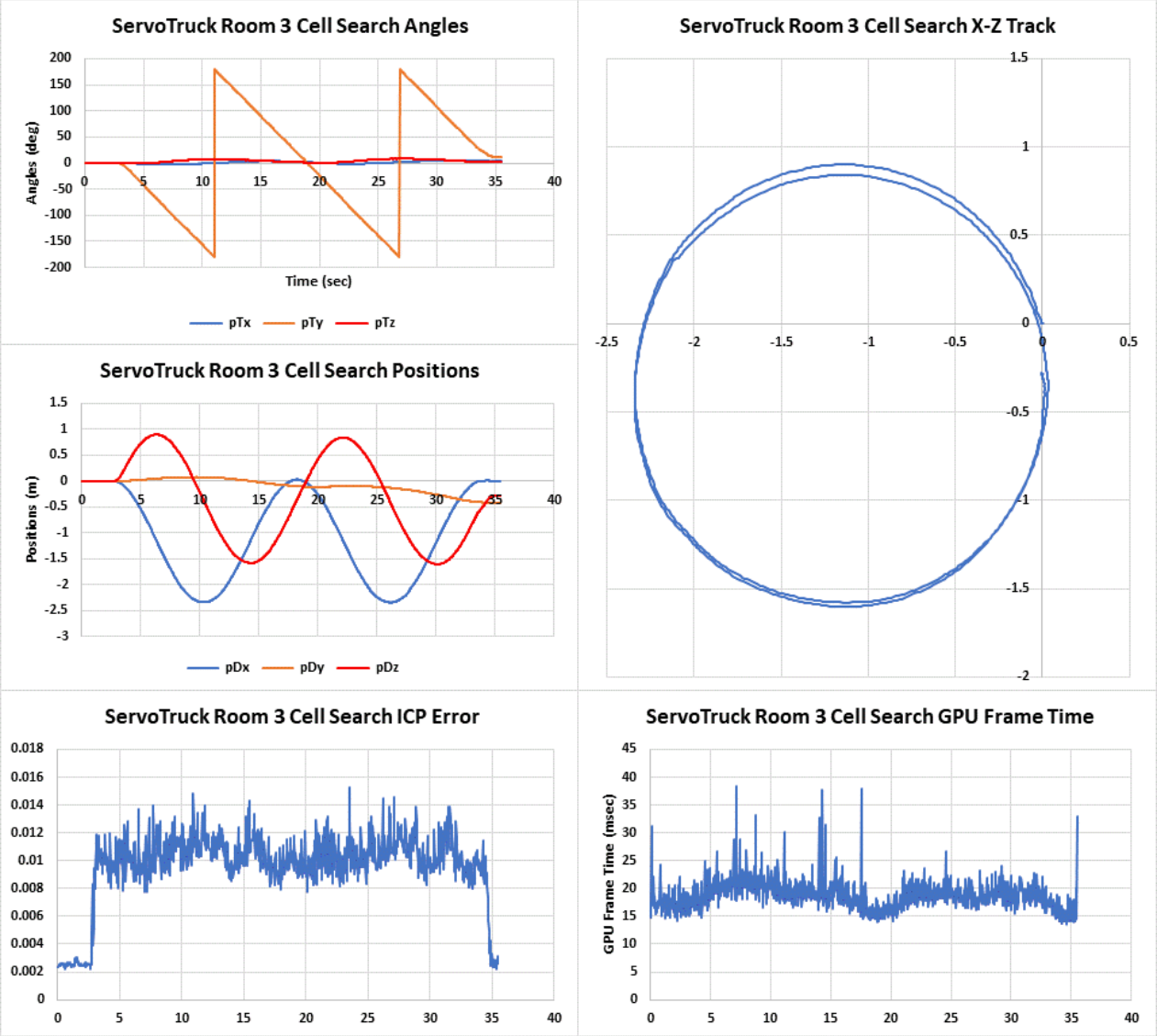


Figure 5.51 ServoTruck Room 3 Test Cell Search Tracking Results

The above results figure 5.51, shows the robot tracking result for two full circuits of the room. The results are as expected with the exception of the upward drift (pDy), the cause for this is unknown. Cell Search helps to reduce drift by excluding points that don't meet the FM similarity metric but the underlying ICP method is point to plane and this will be prone to drift buildup for reasons already discussed above. The cell search method is a helpful improvement in tracking accuracy, but it is rendered less effective if the scene being viewed does not contain strong features. It is also seen to be less effective when cell size is small as multiple (and false) matches become more probable if the features are not particularly distinct.

5.5.5 ServoTruck Hallway Test

Results for this test, based on the same parameters used for the Room 3 test above, are shown on the composite 3D model image and the X-Z ground track on figure 5.52 below.

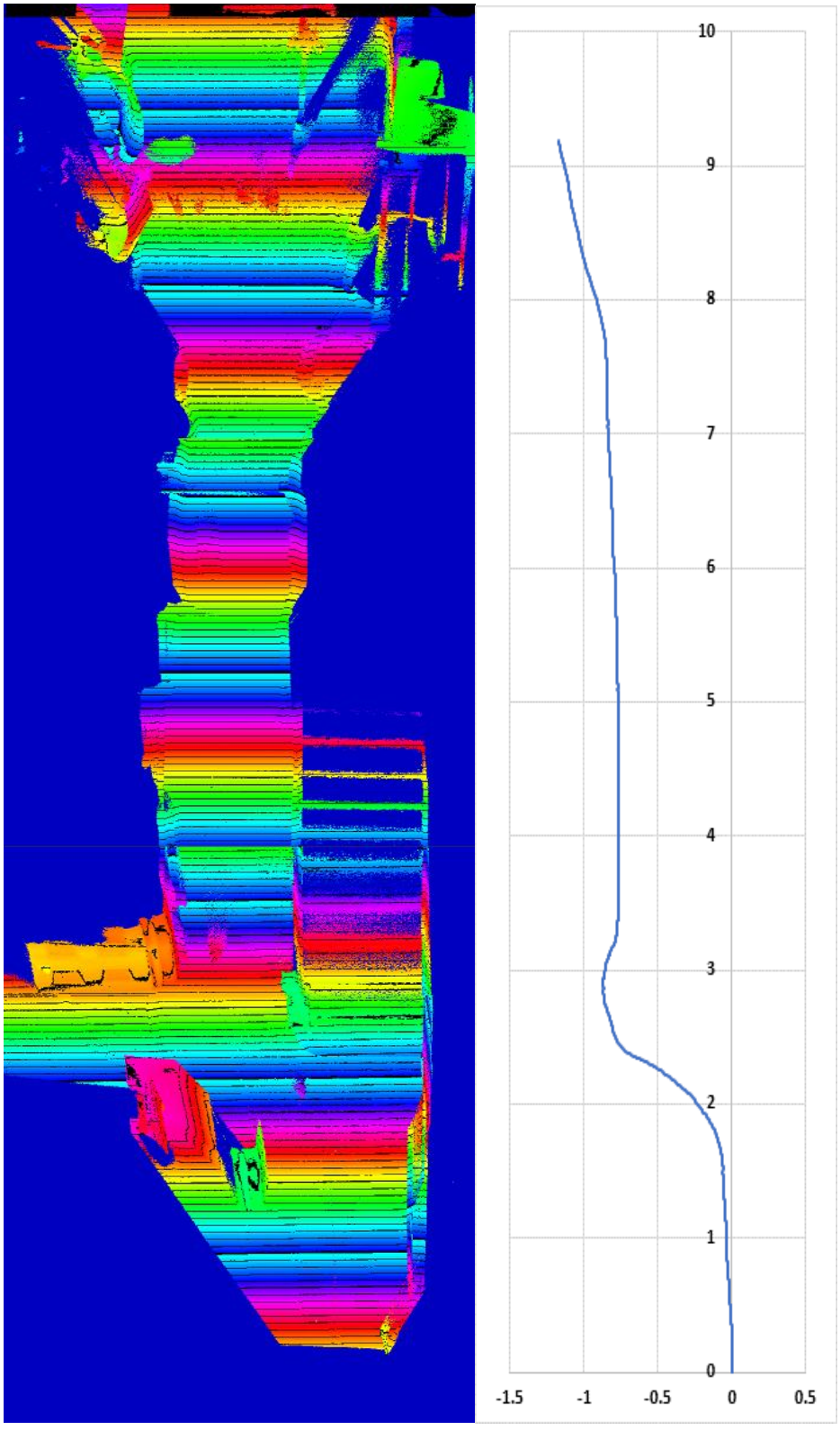


Figure 5.52 ServoTruck 3D Plan and Track for Hall 1 Test using Cell Search Tracking

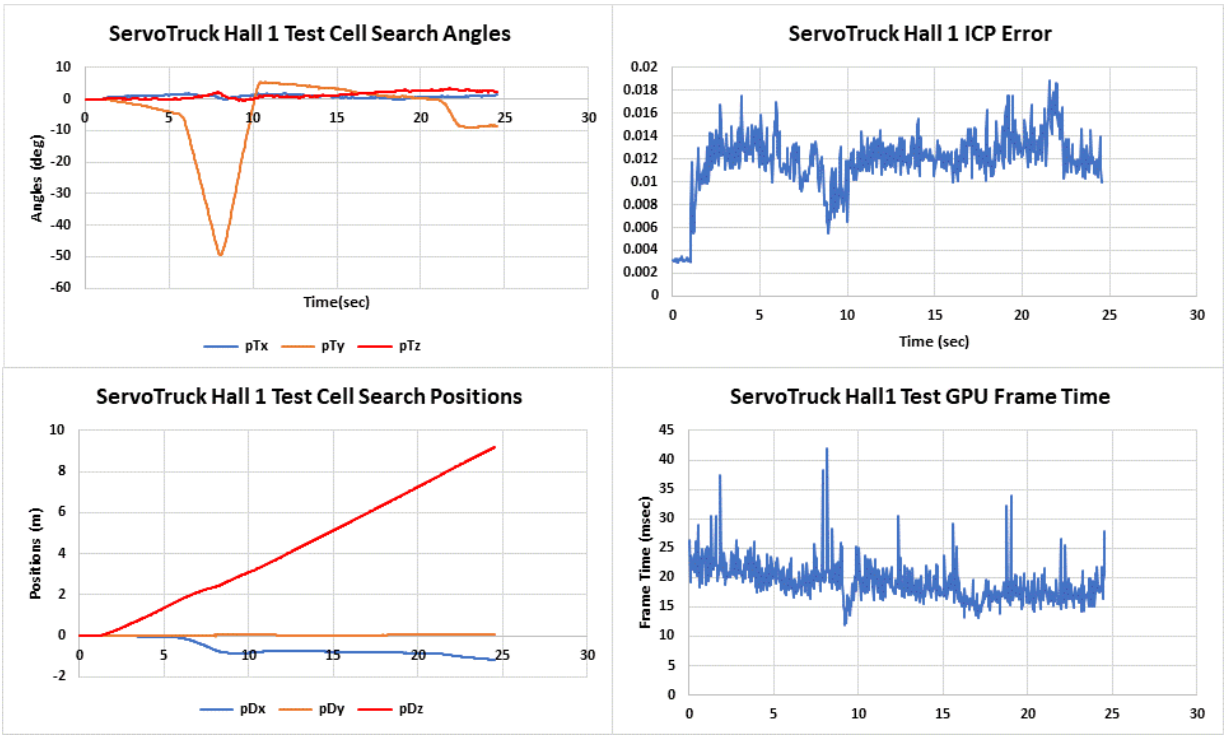


Figure 5.53 ServoTruck Hall 1 Test Tracking Results

The tracking results of figure 5.53 above show that the track followed by the robot is as expected with very little movement on any of the tracking outputs, other than those due to control actions. The quality of the 3D model is also good with little evidence of distortions or extrusions. It also shows that this system is feasible for long range tracking with a GPU frame time well within the target 33 millisecond.

6. Conclusions and Discussion

The overall goal of developing accurate, efficient and robust computer algorithms, based on a low-cost lidar depth camera data and point-based registration methods, for use in autonomous robot tracking and 3D model building at 30Hz has failed.

The aim of achieving 30Hz real-time robot motion tracking using this technology may have been a little ambitious given the now evident limitations of the hardware and the algorithms used. This could not have been deduced aforehand.

The point-to-point registration method proved to be poor in comparison to point-to-plane and for that reason was dropped.

The point-to-plane method, though superior to the point-to-point algorithm also suffers from vulnerability to point-pair miss-matching due to non-overlap in the registration images.

Both the Point-to-Point and Point-to-Plane registration algorithms use a simple L2 norm as a means to register movement between two frames without any regard to occlusions. Thus, without further algorithm modifications designed to eliminate all false pair-matches, minimisation of the L2 metric is too simplistic as a registration mechanism and is therefore not fit for purpose in the context of robot tracking.

Part of the problem identified in the early stages of development was that these depth cameras generate rather a lot of quite noisy data. Thus, a cellular-based Principal Component Analysis was used to reduce noise in the data, reduce quantity of data processed per frame and improve data quality with computed geometric surface normals and other geometric information local to the points analysed.

Further effort was devoted to ensuring accurate point-pair matching augmented with thresholding of geometric distance, the angle between the vectors and minimisation of one-to-many point-pair associations. These measures did not prove sufficient to remove all false pair-matches and hence tracking error buildup.

For this reason, the Förstner-Moonen similarity metric was introduced. This metric definitely helps to remove incorrect pair matches however it also introduces other problems associate with the many scales found in viewing a general scene. This metric works well for larger cell sizes, however use of too large cells for the scene being viewed leads to smaller features being eroded in the registration process. It is also rendered less effective in environments without distinct features such as plain walls and soft furnishings.

Investigations of loop-closure did not provide the expected improvements in tracking accuracy partly because the loop-closure step suffered from the same occlusion-based tracking issues found in open-loop tracking and in fact accelerated rather than reduced drift.

In developing and testing the algorithms on the ServoTrack rig it was found that quite precise tracking with very low drift and good quality 3D models could be produced with a particular set of parameters found during the step-mode testing phase. However, when switching to real-time mode, these original parameter settings were no longer optimal and quite a different set were needed to obtain the best results. This experience repeated again on moving to ServoTruck with quite a different set of tunings needed for optimal results on that rig.

Over all three test regimes it was found that the required parameter settings were mainly driven by motion of the camera on its host platform, rather than the nature of the images being viewed. From this one can conclude that the tracking problem is not robustly solved by these methods, since the camera could in principle be subject to a wide variety of operating scenarios unknown to the system designer.

It could also be considered that for a specific and repeatable operational scenario, a given set of parameters would be the most appropriate and the system usable. For example, a small limited-range robotic manipulator operating in a controlled industrial environment may perform quite reliably with a given set of parameter tunings using this technology.

There is also a question with this type of Lidar technology as to its sensitivity to the lighting conditions under which it is operated. As discussed in Chapter 4 these are quite low power class 1 laser devices and as a consequence are vulnerable to the type and level of ambient lighting. Most of the results of chapter 5 were obtained under low lighting conditions for this reason.

These Lidar cameras also suffer restricted resolution when compared with RGB cameras and produce depth image data that is far noisier than an RGB camera. With development it may be possible for these devices to improve with time but it has certainly proved quite challenging to develop tracking algorithms working at 30Hz using the systems tested.

Industrial Lidar devices capable of operating outdoors, as used in civil engineering surveys, typically use a much larger laser beam diameter 2-3mm and longer wavelength light 1550nm and though still class 1 lasers, have a higher beam power than the devices used here. They are also substantially larger, heavier and slower to gather data, though they also have a much greater precision and range.

A step-by-step systematic developmental approach was used in the project which was started with no precursors or prior knowledge in any of the areas covered. This development process had to be systematic otherwise it would have been impossible to decouple all the many factors without applying

some level of measurement and control whereby performance due to parameter changes could be systematically investigated.

The contributions of this work include:

1. Development of a real-time hardware test environments operating at 30Hz and used for the test and performance evaluation of robot tracking algorithms.
2. Development of a real-time tracking algorithm utilising the Förstner-Moonen similarity metric to improve a point-based tracking.
3. The Development of a GPU algorithm for point-pair association that also inhibits multiple associations to a single point.
4. The development of GPU algorithms for per-thread matrix operations including Eigen-values and vectors by the Jacobi method.

During the development of the project various avenues of further research were uncovered and should be pursued further. The first of which is the use of PCA to reduce the depth image from a set of points to a parametric CAD surface model. There are several motivations for this:

1. It will significantly reduce the data storage requirement for the developed 3D model enabling a much greater model size and robot range for the same storage capacity.
2. Enable better representation of the modelled environment including the possibility of data export to CAD packages such as AutoCAD®.
3. It could also provide the basis for a surface matching registration process.

Though the full details of item 3 remain to be defined, it is believed that a surface-based representation would remove a key difficulty associated with point-based registration processes. This being point-pair matching and the sensitivity of this process to occlusion.

As an alternative approach, the use of some form of image recognition algorithm to locate the common sub-set between two registration images and therefore providing a sound basis for pair-matching minimising the possibility of false pair matching could prove very fruitful. This avenue has not been followed further due to time constraints. It is also felt that performing this operation successfully would be challenging within a 30 Hz frame update rate.

It was not possible within time and cost constraints to fully instrument ServoTruck and this could be re-engineered to include more sensing capability. It occurred during testing using this rig that additional on-board sensing such as orientation, yaw rate and odometry would very much enhance the capability of this equipment. This would also enable better understanding of the role to play between image-based tracking and tracking based on sensors. Humans do not rely on vision alone to understand position and

have additional gravity and acceleration sensing, which serve to reinforce the visual system. It seems quite logical that an autonomous robot would be made more robust given similar additional sensing.

7. References and Further Reading

7.1 References

1 Wikipedia Entry: https://en.wikipedia.org/wiki/Dartmouth_workshop

2 R. Siegwart et al. Introduction to Autonomous Mobile Robots 2nd Ed, Date: 2011, Pub: MIT Press, ISBN: 978-0-262-01535-6

3 Chen Y., Medioni G., Object Modelling by Registration of Multiple Range Images, Date: 1991, Pub: IEEE Int. Conf. on Robotics and Automation

4 B. Curless, M. Levoy. A Volumetric Method for Building Complex Models from Range Images (SDF), Date: 1996, Pub: ACM Trans on Graphics (SIGGRAPH)

5 P. Besl, N. McKay A Method for Registration of 3D Shapes, Date 1992, Pub: IEEE Transactions on Pattern Analysis and Machine Intelligence

6 K.S. Arun et al., Least Square Fitting of Two 3-D Point Sets, Date: 1987, Pub: IEEE Trans. on Pattern Analysis and Machine Intelligence

7 D.W. Eggert, A. Lorusso, R.B. Fisher, Estimating 3-D rigid body transformations: a comparison of 4 major algorithms, Date: 1997, Pub: Machine Vision and Applications, Springer Verlag.

8 F. Pomerleau, F. Colas, R. Siegwart, A Review of Point Cloud Registration Algorithms for Mobile Robotics, Date: July 2015, Available as: https://hal.archives-ouvertes.fr/hal-01178661

9 V. Garcia and E. Debreuve and F. Nielsen and M. Barlaud. k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching, Date: Sept 2010, Pub: IEEE Int. Conf. on Image Processing (ICIP), Hong Kong, China

10 Pauly M., Gross M., Kobbelt L.P., Efficient Simplification of Point Sampled Surfaces, Date: Oct. 2002, Pub: IEEE Visualization Conf. Boston USA, Ref: 0-7803-7498-3/02.

11 Fransens J., Van Reeth F., Hierarchical PCA Decomposition of Point Clouds, Date: 2006, Pub: Proc. 3rd Intl. Symp. on 3D Data Processing, Visualisation and Transmission

12 Förstner W., Moonen B., A Metric for Covariance Matrices, Date: 1999, Pub: Conf. “Quo Vadis Geodesia” Stuttgart University. (Paper obtained from the lead author via Bonn University)

13 Fernando A. de, et al., Comparing Different Strategies for Frame-to-Frame Rigid Registration of Point Clouds, Date: 2017, Pub: IEEE paper.

14 Cejnog L.W.X. Yamada F.A.A. Vieira M.B., Wide Angle Rigid Body Registration using a Comparative Tensor Shape Factor, Date Dec 2016, Pub: Intl. Jorn. of Image and Graphics

15 Förstner W., Wrobel B.P., Photogrammetric Computer Vision, Date 2016, Pub: Springer, ISBN: 978-3-319-79170-8.

16 M. Magnusson, A. Lilienthal, T. Duckett, Scan Registration for Autonomous Mining Vehicles using 3D-NDT, Date: 2007, Pub: Journal of Field Robotics.

17 M. Magnusson, et al., Evaluation of 3D Registration Reliability and Speed – A Comparison of ICP and NDT, Date: May 2009, Pub: IEEE Intl. Conf. on Robotics and Automation, Kobe, Japan

18 Magnusson M. et al., Beyond Points: Evaluating Recent 3D Scan-Matching Algorithms, Date: 2015, Pub: IEEE ICRA, Seattle

19 Thrun S., Burgard W., Fox D., Probabilistic Robotics, Date: 2000, Unpublished draft downloaded from: https://docs.ufpr.br/~danielsantos/ProbabalisticRobotics.pdf

20 Montemerlo M., Thrun S., et al., FastSLAM: An Efficient Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association, 2003, Available as: http://robots.stanford.edu/papers/Thrun03g.pdf

21 Jie Han, et al., An Improved RANSAC Registration Algorithm Based on Region Covariance Descriptor, Date: 2015, Pub: IEEE Chinese Automation Conference. Wuhan China

22 Truong G. et al., Fast Point Cloud Registration using Semantic Segmentation, Date: 2019, Pub: IEEE

23 Newcombe R.A., et al., KinectFusion: Real-Time Dense Surface Mapping and Tracking, Date: 2011, Pub: IEEE 06162880, Basil, Switzerland

24 Sanders J., Kanderot E., CUDA by Example, Date: 2011, Pub: Addison Wesley, ISBN: 978-0-13-138768-3, 2011.

25 Storti D., Yurtoglu M., CUDA for Engineers, Date: 2016, Pub: Addison Wesley, ISBN: 978-0-13-417741-0

26 Wilt, N., The CUDA Handbook, Date: 2013, Pub: Addison Wesley, ISBN: 978-0-321-80946-9

27 GitHub Code reference link To Here

28 Love R., Linux Systems Programming, Second Edition, Date: 2013, Pub: O'Reilly, ISBN 978-1-440-33953-1

29 Seiwert S., Pratt J., Real-Time Embedded Components and Systems with Linux and RTOS, Date: 2016, Pub: Mercury Learning and Information, ISBN 978-1-942270-04-1

30 Abbot D., Linux for Embedded and Real-Time Applications 4th Ed, Date: 2018, Pub: Newnes, ISBN 978-0-12-81127

31 Ibrahim D., ARM-Based Microcontroller Multitasking Projects – Using the FreeRTOS Multitasking Kernel, Date: 2020, Pub: Newnes, ISBN 978-0-12-821227-1

32 Gentle J.E., Matrix Algebra – Theory, Computations and Application in Statistics. 2nd Ed, Date: 2017, Pub: Springer, ISBN: 978-3-319-64867-5

33 Press W.H., et al., Numerical Recipes – Third Edition, Pub: Cambridge University Press, Date: 2017, ISBN 978-0-521-88068-8

34 Das H., Slotine J-J. E., Sheridan T.B., Inverse Kinematic Algorithms for Redundant Systems, Date: 1988, Pub: IEEE CH2555-1

35 EEMBS CoreMark® Web site: https://www.eembc.org/coremark/

36 Sanders P., et al., Sequential and Parallel Algorithms and Data Structures, Date: 2019, Pub: Springer

37 Anon, Intel® RealSense™ Product Family D400 Series Datasheet, Date: Aug 2021, Pub: Intel

38 Anon, Intel® RealSense™ Lidar Camera L515 Datasheet, Date: June 2020, Pub: Intel

39 Intel SDK Web site: https://www.intelrealsense.com/get-started

7.2 Further Reading

FR1 Mitchell T.M, Machine Learning, pub: McGraw Hill Intl, Date: 1997, ISBN-0-07-115467-1.

FR2 Goodfellow I.J., Bengio Y., Courville A., Deep Learning, Pub: MIT Press, Date: 2016, ISBN 9780262035613.

FR3 Bishop C.M., Pattern Recognition and Machine Learning, Pub: Springer, Date: 2006, ISBN: 10-0-387-31073-8

FR4 Watt j., et. al. Machine Learning Refined, Pub: Cambridge University Press, Date: 2016, ISBN: 978-1-107-12352-6

FR5 Russell S., Norvig P., Artificial Intelligence – A modern Approach 3rd Ed., Pub: Pearson, Date: 2016, ISBN 13-978-1-2921-5396-4

FR6 Wilamowski B.M., Irwin D.J., Intelligent Systems 2nd Ed., Pub: CRC Press, Date: 2011, ISBN 978-1-4398-0283-0

FR7 Github article CS231n: "Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/neural-networks-1/

FR8 Athans M., et. al., Systems Networks and Computation: Multivariable Methods, Date: 1974, Pub: McGraw Hill, ISBN: ISBN: 0-07-002430-8

FR9 Panin G., Model-Based Visual Tracking, Date: 2011, Pub: Wiley, ISBN: 9780470943922

FR10 Brogan W.L., Modern Control Theory, 3rd Ed, Date: 1991, Pub: Prentice Hall. ISBN 0-13-590415-3

FR11 Verhulst F., Nonlinear Differential Equations and Dynamical Systems 2nd Ed, Date: 2000, Pub: Springer, ISBN 13-978-3-540-60934-6

FR12 Stewart J.M., Python for Scientists and Engineers, Pub: CuP., Date: 2014

FR13 Mendenhall W., Sincich T., Statistics for Engineering and the Sciences 5th Ed., Pub: Pearson, Date: 2007, ISBN 0-13-187706-2

FR14 Chen G., et. al. Fundamentals of Complex Networks: Models, Structures and Dynamics, Date: 2015, Pub: Wiley, ISBN: 978-1-118-71811-7

FR15 R. Cant, C. Langensiepen, L. Foster, A Composite Convex Hull Collision Detector for the Open Dynamics Engine, Pub: IEEE ISBN: 0-7695-3114-8/08, Date: 2008.

FR16 David V. Gealy, et al., Quasi-Direct Drive for Low-Cost Compliant Robotic Manipulation, Presented at ICRA 2019 in Montreal, Canada.

FR17 Shannon C.E., A Mathematical Theory of Communication, Reprinted for the Bell Systems Technical Journal. Date: 1948

FR18 Kolmogorov A.N. Three Approaches to the Quantitative Definition of Information. Int. Jour. Of Comp. Math. 1968, Available as: https://doi.org/10.1080/00207166808803030

FR19 Hilbert M., Formal Definitions of Information and Knowledge… Available as: http://doi.org/10.1016/j.strueco.2016.03.004 2016

FR20 Sutton R.S., Barto A.G., Reinforcement Learning: An Introduction, Second Edition, MIT Press, Date: 2018.

FR21 Brooks R.A., A Robust Layered Control System for a Mobile Robot, M.I.T. A.I Memo 864, Date: September 1985

FR22 Brooks R.A., Intelligence Without Reason., M.I.T. A.I. Memo 1293, Date: April 1991

FR23 Arkin R.C., Balch T., AuRA: Principles and Practice in Review., Georgia Inst. Of Tech. Date: 1997, Available as: https://www.cc.gatech.edu/ai/robot-lab/online-publications/jetai-final.pdf

FR24 Raol J.R., Gopal A.K., Mobile Intelligent Autonomous Systems, Pub: CRC Press ISBN 978-1-138-07245-9, Date: 2013.

Appendix A Embedded Computing

A1 Introduction

Anyone entering the world of Embedded computing is faced with a bewildering array of different hardware and software options. There are many vendors each with their own off-the-shelf and bespoke product ranges but very little in the open literature to provide useful comparisons or guidance in selecting such systems.

For example, during the development of this projects ServoTrack test hardware, no less than 5 different computers were tried before a satisfactory solution was obtained. It was not thought that the task being set was particularly difficult or challenging, but nor was it appreciated just how optimised Embedded computing options can be. The following table shows a selection of some of the options investigated.

Hardware Version Format	Lenovo 330S Laptop	Latte PandaV1 Card	Latte Panda α Card	Raspberry PI 4B Card	ONLOGIC EMP160 Heatsink	Intel I7 NUC Box
Linux Distribution	Ubuntu 18.04LTS	Ubuntu 20.04LTS	Ubuntu 20.04LTS	Raspbian raspi3	Ubuntu 18.04LTS	Ubuntu 20.04LTS
Kernel Version	4.15.0	5.4.0-104	5.13.0-28	5.4.83	5.4.0-74	5.4.0-109
Processor Type	X86_64	X86_64	X86_64	Arm64	X86_64	X86_64
Processor cores	8	4	4	4	2	4
Processor Model	Intel I5 8250	Intel x5 z8350	Intel m3 8100y	ARM CortexA72	Intel N3350	Intel i7 1165G7
Proc Speed (GHz)	1.1	1.4-1.92	1.1-3.4	1.5-2.145	1.1-2.4	2.8-4.9
CoreMark Score	109424	17027	53619	33150	22831	154738
CoreMark/MHz	68	12	48	22	20	55
RAM (Gb)	8	4	8	8	1	8
SSD Storage (Gb)	256	64	64	0	128	128
SD card (Gb)	slot	slot	slot	64	No	slot
WiFi	Yes	Yes	Yes	Yes	Yes	Yes
Ethernet	No	Yes	Yes	Yes	Yes	Yes
HDMI or Equivalent	Built in screen	Yes	Yes	Yes	Yes (via Adaptor)	Yes
USB Ports u2=USB2 u3=USB3	2xu2 1xu3	2xu2 1xu3	3xu3	2xu2 2xu3	2xu3 1xu2	4xu3 2xu2
Supply Volts	19-21	5	12	5	12	19-21
CoreMark Power W	N/A	1.8	15.1	3.9	8.4	39
Quiescent Power W	N/A	0.6	2.76	1.5	4.8	1.9
Approx. Cost (UK£)	270	129	338	25	290	420

Table A1 Showing a Selection of Tested Linux Embedded Systems

The laptop is too large to be considered as an Embedded computer, but it is used as the primary control interface on the ServoTrack project and provides a normative comparison with the selected Embedded systems.

With increasing computing power and reducing cost there is an expectation (reasonable or not) that the above systems will be capable of in-situ code development, debug and testing. For example, all of the systems investigated, offered a General-Purpose Operating System (Linux). Whether they were capable of running it efficiently and effectively or not was in some cases questionable.

Items indicated in red in table A1 above were found to be a problem adversely affecting performance and/or functionality in some if not all tasks carried out.

A useful metric for assessing relative performance of Embedded computers is the CoreMark scoring system. The systems in Table A1 were tested using the standard and unmodified comparison software available from: https://www.eembc.org/coremark/. The value obtained for the Raspberry Pi 4B was validated against the CoreMark performance database.

The CoreMark test only ranks processor capability so another two tests; a compile test and a random number generation test were also carried out. Results for the computers in Table A1 are shown on the bar-chart figure A1 below.

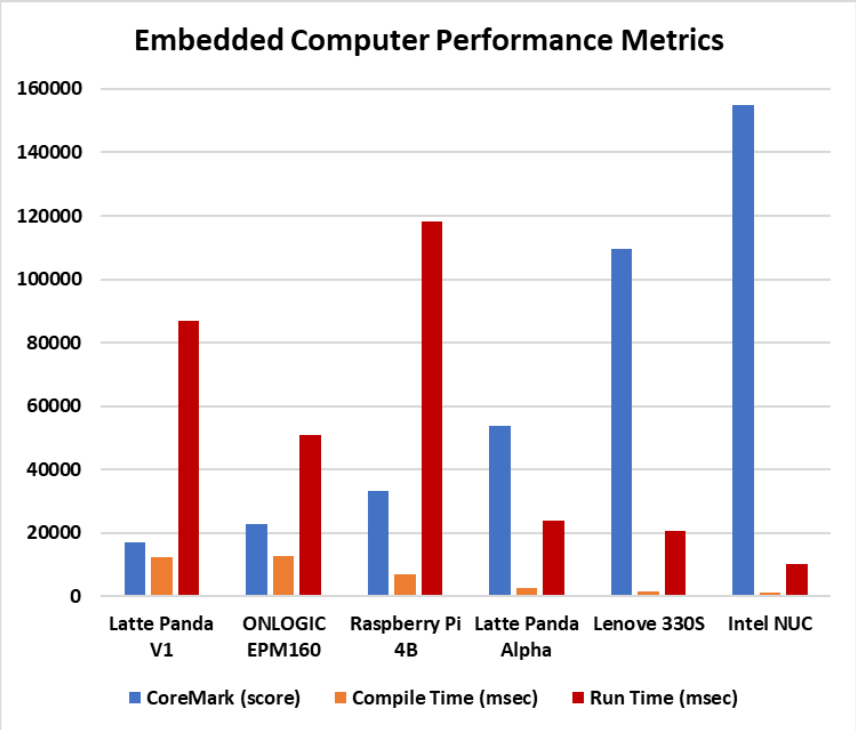


Figure A1 Embedded Computer Performance Metrics for Table A1 Systems.

These computers are ranked in order of the unnormalized CoreMark score. Note that although the Raspberry PI 4B ranked 3rd by this metric, it performed least well in the Run Time test. It is not clear as to why this is the case, though it was also found to be a poor performer when working with the device. The LattePanda V1 and the ONLOGIC EMP160 were both also found to be poor performers in practice so are ranked correctly by CoreMark metric. The LattePanda Alpha was the first found to perform adequately and thus, a CoreMark above 50,000 would seem to be required for a real-time application using Linux. Whist the above table focuses on speed, power consumption is also a concern especially for embedded devices with resource constraints such as battery life. The following plot figure A2, shows power consumption for the devices in table A1 above, the laptop is not included.

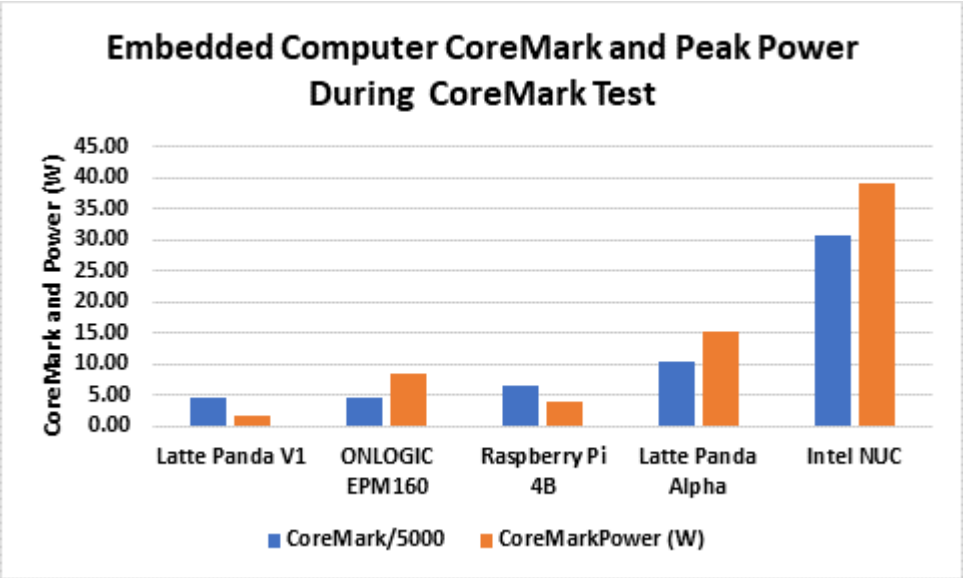


Figure A2 Power Consumption for the Table A1 Devices

The best ratio of CoreMark to power is the LattePanda V1 followed closely by the Raspberry PI 4B all of the rest require much more power with the ONLOGIC EMP160 being notable the least efficient by this measure. The Intel NUC scores heavily on both metrics and is probably not an optimum choice for a constrained power environment, however it does clearly perform well and such performance does not come without cost.

A1.1 Smaller Embedded Systems

Smaller Embedded systems such as are typically used in automation and domestic appliance applications are not in general capable of running a general-purpose OS such as Linux[2]. The operating paradigm in all cases shown being that code development is carried out via an Integrated Development Environment (IDE) running on a desk-top computer, with the cross-compiled code downloaded to the target device.

[2] Some vendors for example NXP offer a Yocto-based and highly optimised Linux on their larger embedded solutions such as the i.MX8. These system combinations were not investigated due to project time and cost constraints.

Download and debugging is typically carried out over a serial link such as JTAG or USB. The CoreMark/MHz scores in this case were not tested by the author but taken from [25] and the Coremark web-site [30].

Hardware Version Format	Arduino Uno	Arduino Mega 2560	Arduino Due	STM Nucleo32 F411RE	NXP Xpresso 1549	Atmel SAM V71
Operating System	N/A	N/A	FreeRTOS?	Not known	FreeRTOS	FreeRTOS
Integrated Development Environment	Arduino	Arduino	Arduino	IAR or Keil	Eclipse-based IDE	Atmel Studio
Processor Core IP/Type	Atmel ATMega 328P	Atmel ATMega 2560	Atmel SAM3 ARM Cortex M3	ARM Cortex M4	ARM Cortex M3	Arm Cortex M7
Proc Speed (MHz)	16	16	84	32	72	300
CoreMark/MHz	low	0.53	2.8	3.4	3.2	5.01
RAM	2Kb	8Kb	100Kb	192Kb?	36Kb	128Kb
Flash	32Kb	256Kb	512Kb	512Kb	256Kb	128Kb
SSD Storage (Mb)	No	No	No	No	No	No
SD card	No	No	No	No	No	Yes
WiFi	No	No	No	No	No	No
Ethernet	No	No	No	No	No	Yes
Arduino Shield	Yes	Yes	Yes	Yes	Yes	Yes
USB2 Ports	1	1	2	1	2	2
JTAG/Debug	No	No	Yes	via usb	No	Yes
GPIO Pins	14	54	54	30+	No	30+
I2C	Yes	Yes	Yes	?	Yes	Yes
SPI	Yes	Yes	Yes	?	Yes	Yes
Supply Volts	12/5	12/5	12/5	5 (usb)	5 (usb)	5-14
Serial/UART	1	4	4	Yes	Yes	Yes
Size LxWxD (cm)	8x5.5x1.5	11x5.5x2	10.5x5.5x1.5	7x8.5x2	8.5x5.5x1	14x10x2
Cost (UK£)	18	32	32	30	25	203

Table A2 Showing a Typical Range of Smaller Embedded System

It is interesting to compare the above Embedded boards in Table A2 with those of Table A1. The difference in processor speed between the two sets of Embedded systems is clearly very significant. The best Embedded system in Table A1 giving a CoreMark/MHz of 55 compared to the best of Table A2 at a CoreMark/MHz of 5 giving a ratio of 11:1 and yet all of the boards in Table A2 are hard-real-time capable.

For example, the Arduino Mega 2560, with one of the slowest processor speeds at 16MHz, is still capable of hard-real-time performance, processing all analogue and digital IO and receiving/sending messages over Serial USB at the required frequency of 30Hz with high time precision. Two of these devices are used

in the ServoTrack project and work very well and with only a fraction of the processing power of the Linux boxes to which they are connected.

Arduino have also clearly influenced the embedded computer world with all of the above boards offering Arduino shield compatibility. However, on the grounds of speed and simplicity the Arduino operating system and IDE also wins out over many of the other embedded offerings which, have the bad habit of presenting the programmer with far too much fine-grained and unnecessary detail. For example, large header files containing nothing other than reams of enumerated IO and DMA vectors with hexadecimal addresses, rather than simple integer IO pin number addressing.

The Embedded systems in Table A2 are also clearly dominated by ARM Cortex M series processors. These are noted for low power and high efficiency and are especially useful in constrained power devices such as smartphones, where they are quite dominant.

A1.2 What is Required of an Embedded System

The outline specification requirement for this project was as follows:

1. HDMI Screen
2. Keyboard and Mouse
3. At least 1 USB-2 port; for Arduino Serial Comms
4. At least 1 USB-3 port; for Intel Depth Camera
5. WIFI for headless operation and communications with laptop
6. Sufficient RAM and SSD storage to operate as intended[3]
7. Small footprint; less than 15x15x5 centimetres
8. Low power; ideally less than 10 watts[4]
9. Low voltage; ideally 12VDC or less

A fundamental dilemma for anyone selecting an appropriate Embedded computing platform for a new application is that it is difficult if not impossible a-priori to specify the required computing capability. This proved to be the greater part of the learning process on this project. By a process of trial and error (found to be expensive in both development time and cost) a fit-for-purpose system was finally developed. Since the system had not yet been built and no prior systems existed, the demands upon the computing hardware were not at all understood and thus a specification of required computing performance simply not possible.

[3] The Raspberry Pi has no SSD and relies on a micro-SD card for storage.
[4] This requirement and the low voltage were both exceeded by the Intel NUC finally selected.

A2 Why Linux for a Real-Time Embedded System

Along with the bewildering array of Embedded hardware on offer is also a range of software platform solutions on which to build an Embedded system.

A2.1 Bare Metal Programming

The starkest of choices is known as “bare-metal programming”. In this model the programmer works directly with the computer hardware and a cross-compiler typically supplied by the hardware vendor and develops the system from scratch. This involves working directly with hardware interrupts, via Interrupt Service Routines (ISR’s) and Direct Memory Access (DMA) typically with a simple cyclic real-time loop structure. This can work well for specific applications such as (for example control of a Washing Machine or Microwave Oven) with very limited human interaction and no unknown or unexpected inputs. Quite often embedded system vendors (NXP, STM, Atmel and others) will provide an IDE and cross-compilation tools to develop the software with other support tools to run and debug the code on their target hardware.

A2.2 Real Time Operating System

At the next level in the compute hierarchy are real-time operating systems (RTOS) and there are a number of commercial and at least one free operating system (FreeRTOS). For a hard real-time system, where failure to meet a deadline could result in loss of life or expensive equipment, this is the best and perhaps only choice.

Proprietary RTOS systems are available and come with the advantage of support, but also lock the user into their processes, limited hardware selections and potentially additional costs.

Perhaps for this reason FreeRTOS is becoming a de facto standard within the Embedded community with many copies downloaded and active support from many hardware vendors. The fact that it has now been taken over and is supported by Amazon Web Services is sure to secure its future as the dominant free offering.

It also supports Asymmetric and Symmetric Multiprocessing for multi-core systems. It still does not provide much support for X86 systems tending to focus mainly on the smaller ARM-based Embedded system shown on table A2.

Choosing this path comes with limitations in terms of hardware platform compatibility, the potential need to write hardware drivers for specific devices and a very limited range of applications with which to develop and debug such systems and this can translate into high workload. For a real-time hardware project where unit sales can be used to write off development costs this can be the optimum route. For a

one-off time-constrained project such as this it is not so inviting. However, for future projects this path appears to offer a simpler and more efficient means of achieving hard-real-time performance without all the encumbrances of a general-purpose OS.

A2.3 A General-Purpose Operating System - Linux

Linux is one of the most popular operating systems for general computing and is widely used in many application areas including Embedded computing applications. It is freely available, supports many different platforms hardware devices and software packages.

This concept of “free” should be qualified as there are also downsides in terms of the bewildering array of different distributions available and the overall lack of any coherent support. The saying: “That which is free comes with no guarantee” is particularly relevant here. Whilst the most popular free Linux distributions (including Ubuntu used here) will provide some level of support, this is usually confined to posting questions on user-group web-pages with indeterminate levels of response in terms of time and quality. Most distributions also offer a commercial support package for a price.

There are some notably useful websites such as: stackoverflow.com and stackexchange.com where it is often the case that the question has already been asked and answered. However, some caution is needed here with regard to the timeline of the questions and answers. Updates to the software in question can and often do, result in earlier issues becoming stale so the suggested “solution” may be long outdated and no longer relevant.

One of the most popular Linux distributions is Ubuntu, this also offers Long Term Support (LTS) versions every couple of years, which are supported for at least 5 years. This choice was also heavily influenced by use of the Intel RealSense™ Depth Cameras and NVIDIA GPUs both of which supported Ubuntu LTS versions. The only other option being Microsoft™ Windows®, but this would have added significantly to project cost with a need for several licences.

A2.4 Linux Issues and Real-Time Performance

Linux, can never be considered a hard-real-time (HRT) operating system, it is far too complex and as a result the execution path can never be considered predictable or deterministic. However, for many systems which are not safety critical soft-real-time (SRT) can be acceptable. Measures of quality of soft real-time performance are latency (lags in the system) and jitter (variability in response).

There are several aspects of Linux that are not attuned to real-time operation. The kernel itself must complete certain actions in entirety leading to variable levels of delay in providing interrupt response. The unmodified kernel will not be particularly responsive to real-time interrupts whilst it is carrying out such tasks (which may well be as a consequence of a requests from the “real-time” program).

Other non-real-time aspects are the Completely Fair Scheduling (CFS) system, the Symmetric Multi-Processing (SMP) system and virtual memory paging system (swapping live but unused memory pages to a swap file on a permanent storage device). There is also no user-programming interface or access to hardware interrupts which are all handled within kernel code Interrupt Service Routines.

A2.4.1 CFS

The aim of CFS is to share the available processing capability between all of the running tasks. This includes operating system tasks (though not kernel code) as well as all user programs if no special action is taken. Every non-kernel task is given equal priority (zero in the CFS case), and all running tasks get an equal portion of the available compute capability. This ensures a reasonable process throughput but does not fit well with providing real-time responses to hardware interrupts. The fraction of time the CFS system gives each process can be biased using a “nice” value. If this is used the process with the lowest nice value will be given the most processing time. However, the CFS can be disabled by selecting one of the advanced real-time schedule options these include:

SCHED_OTHER	This is the same priority (zero) and behaviour as the CFS.
SCHED_FIFO	This option will allow tasks to run until blocked or interrupted by a higher priority task.
SCHED_RR	This option will allow a set of tasks to run concurrently in round-robin fashion each getting a predefined time-slice.
SCHED_DEADLINE	This more complex and new option will tell the OS that you have a given deadline to meet allowing it to schedule other tasks in and around it.

A2.4.2 SMP

The aim of SMP is to balance workload across all the available processors. Since most processors are multi-core these days, this should improve overall throughput; however, an undesirable side effect of this system is that a process can be migrated from one processor core to another and this introduces random delays which are incompatible with real-time programming constraints. This can be circumvented for the given program by setting processor affinity. This will not prevent the OS from running other tasks on the same core however if process priority is also set to a high real-time value the OS will not be able to run anything else on the selected core.

A2.4.3 PREEMPT_RT and Low Latency Kernels

The responsiveness of the kernel to real-time interrupts can be improved. For many years now a kernel patch known as PREEMPT_RT has been available. This does require a complete recompilation of a source kernel and patch on the target platform. This is typically a 3-4hour task on any decent system with all the

build tools needed. One problem found with this approach was that the WIFI system disappeared. Whether this was a deliberate part of the strategy or simply an issue to be corrected was not established.

An alternative solution for Ubuntu LTS versions, known as the low-latency kernel is also available. This kernel comes pre-compiled for X86-64bit hardware and was a matter of minutes to install. It does not provide the complete pre-emption capability of the PREEMPT_RT patch but works with all other application used on the system including WIFI and is supported by Ubuntu.

Figure A3 below shows a simple statistical comparison (based on code supplied in [23]) of real-time performance carried out on the Lenovo Laptop. This timing test is based on waiting for a fixed 10 millisecond timer to complete and measuring the over-run time on return. The test shows the relative effectiveness of the given configurations to meet real-time interrupts. For this test process priority was set to SCHED_FIFO with a priority of 99 and process affinity was not set.

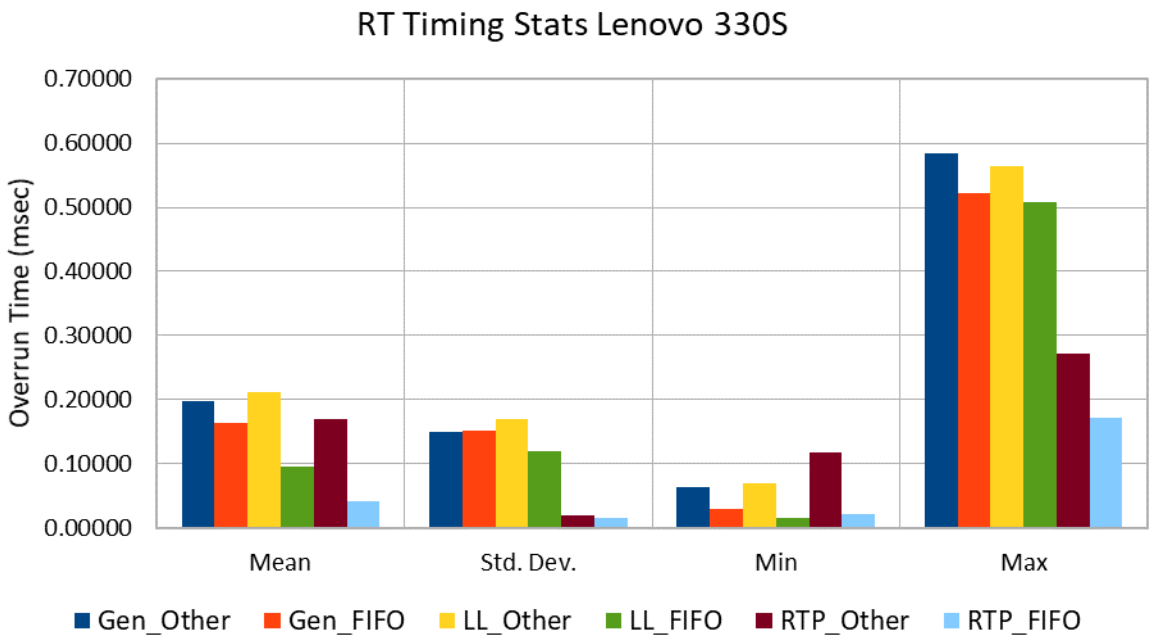


Figure A3 Showing Real-Time Performance Statistics for Various Linux Configurations

The configurations denoted in figure A3 plot labels above are:

Gen_Other	Ubuntu 18.04 LTS generic build with SCHED_OTHER (priority=0)
Gen_FIFO	Same config. as Gen_Other but with SCHED_FIFO (priority=99)
LL_Other	Ubuntu 18.04 LTS + Low Latency Kernel (priority=0)
LL_FIFO	Same config. as LL_Other with SCHED_FIFO (priority=99)
RTP_Other	Ubuntu18.04 + PREEMPT_RT patch (priority=0)
RTP_FIFO	As above with SCHED_FIFO (Priority=99)

All OS options were installed on the same laptop computer using Grub to allow menu selection of the OS configuration at boot time. Several conclusions can be drawn from the above results.

- Selecting SCHED_FIFO did not cause any measurable improvement on the standard generic system; this conclusion was also born out on more complex tests and other platforms.
- Using the low-latency kernel does not have any particular benefit if SCHED_OTHER is selected, however there is significant benefit when using this kernel in combination with SCHED_FIFO (and priority=99).
- The PREEMPT_RT patch provides the best overall response (and the most consistent – with lowest Standard Deviation) especially with SCHED_FIFO. WIFI would not work (was disabled) with this option, it is not known as to the cause but an experience shared with others online.

Other aspects that are not real-time compatible such as virtual memory paging can also be overridden using Linux memory locking API to lock program memory into RAM provided there is sufficient RAM within the system to allow this.

A3 Some Actual Linux Real-Time Issues

In spite of implementation of all the above options, with such a complex operating system it is still possible to encounter problems. There are a number of issues that can affect real-time performance that are not easily fixed by setting a high priority on a given code. This is because although the code itself will run at high priority, it will generally call lower-level Linux services such as file and IO buffers that are not running at that higher priority.

A quick check of any resource monitor on Linux will show that most if not all tasks and daemons are running at priority zero. This is the CFS system in action and it is fundamental to the overall behaviour of the Linux operating system.

This has several implications beyond user control. For instance, a high priority application can end before its file buffers have been purged back to disk, thus care is needed to ensure the file IO buffer has been flushed before program closure. This is not an issue for a normal priority task which would end with all files purged to storage and closed in an orderly manner.

On a computer with only 2 cores, it was also possible to completely lock the system up with nothing other than power-off cycling as a means of escape. This was due to the fact that with only two processor cores the use of higher priority on two tasks left Linux with nowhere to run the CFS tasks that had been requested by the higher priority processes hence total lock-up.

From the experiences gained from this exercise one feature comes out above all others. For good real-time performance, first and foremost Linux requires a high-performance processor. Three of the systems tested and presented in Table A1 failed in various ways and for various reasons, but the main issue was lack of computing performance. Linux can do soft-real-time but the processing requirement to achieve this is significant. It is almost certainly the case that a simpler real-time operating system would perform all of the tasks required more efficiently and with a much lesser performing computer. The only problem being interfacing with proprietary hardware such as the Intel Depth Camera, leaving the programmer to write the interfacing code. A task that would have added significantly to the project time.

A3.1 Latency and Jitter Due to Propagation Delays

As discussed in section 4 there are various data transmission media used in the ServoTrack test rig these are all bandwidth limited to some degree and are thus important sources of latency and jitter which are additional to the Latency within the Linux OS.

In order to assess propagation delay across the various data transmission media a messaging test was set up. At the sending end the time is taken using the system clock. This time is then sent in a data packet to one of the connected devices. On receipt the device then immediately returns the same data packet back to the source. The time different between send and receipt is then a measure of the two-way round-trip time for that data packet on that channel. The timer packet is quite small (total length 19 bytes).

Some tests were carried out for both the Generic (Gen) and Low-Latency (LL) Ubuntu/Linux Kernels.

Both TrackHost (Linux) to TrackDRV (Arduino) and CarHost to CarDRV communicate via Serial USB-2 port. However, the data transmission rate is fixed at the lowest common denominator which is 115200 baud or approximately 12Kbps (USB-2 is actually capable 0f 24Mbps). Results for the TrackHost to TrackDRV tests are shown on figure A4 below.

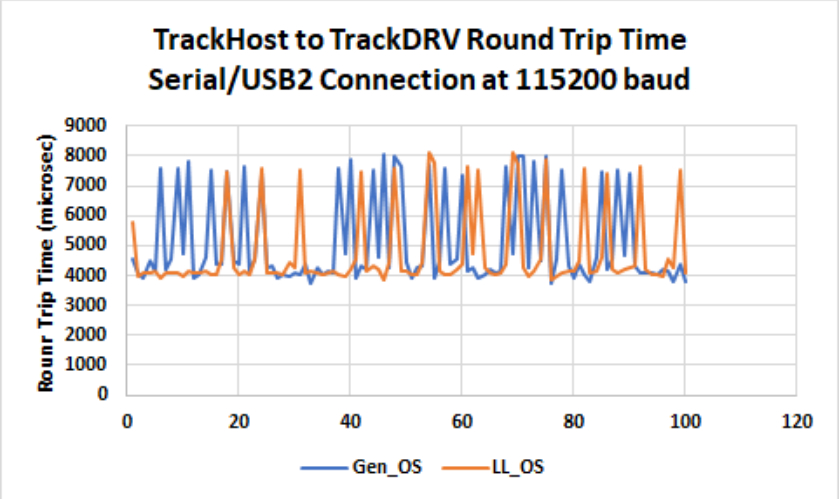


Figure A4 Linux to Arduino Serial Propagation Delay

If a symmetric round-trip time is assumed, then the one-way propagation delay is between 2 and 4 milliseconds. Based on the data-packet size and baud rate a one-way propagation time of 1.7 milliseconds (1.7 msec) would be expected. The above results show a base 4msec response however they also show occasional sporadic peaks up to 8 msec. The reason for these peaks is not fully understood, both the Linux and Arduino loops will respond as soon as the data arrives. Results for the Ubuntu Low Latency Kernel are very similar to the Generic Kernel in this test. The underlying interrupt service handling USB serial communication is not affected by the change of kernel, this does not mean that the low latency kernel is not useful, just that it has no effect on this issue.

The WIFI system proved to be very much more problematic. The main issue being the very large variation in latency. Figure A5 below shows a typical result between TrackHost and CarHost.

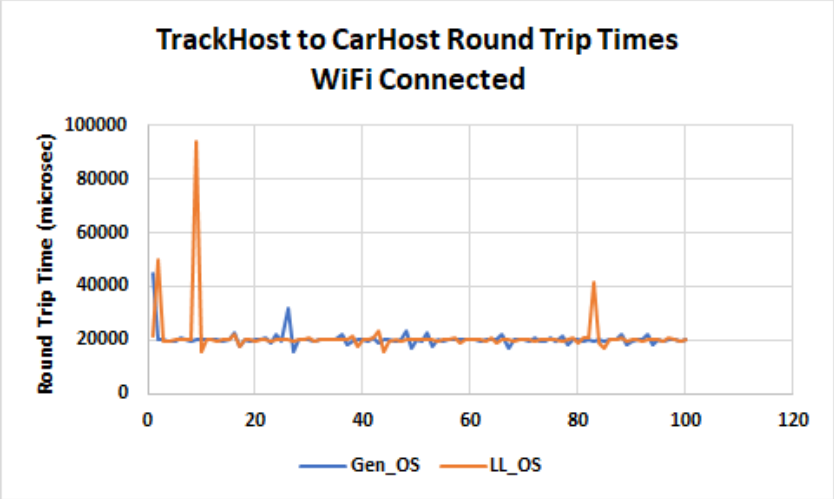


Figure A5 Linux to Linux WIFI Propagation Delay

The general round-trip time of 20msec implying a 10msec propagation delay is marginally acceptable. However, the plot also shows one spike at around 90 msec. This is unacceptable and unfortunately not untypical; the cause of the large variation is believed to be due to the WIFI router. Although 35Mbps can be achieved when transferring large files between two computers using WIFI, the system is very variable and sporadic for small data packets such as used in this test. These results are also typical of the variation observed in ping test times between the two computers.

The disappointing performance of the WIFI system prompted investigation of other wireless delivery mechanisms. Firstly, to establish that there was nothing wrong with the TCP/IP messaging code between Linux systems, a timing test using a wired Ethernet connection was carried out. Results for this test are shown on figure A6 below.

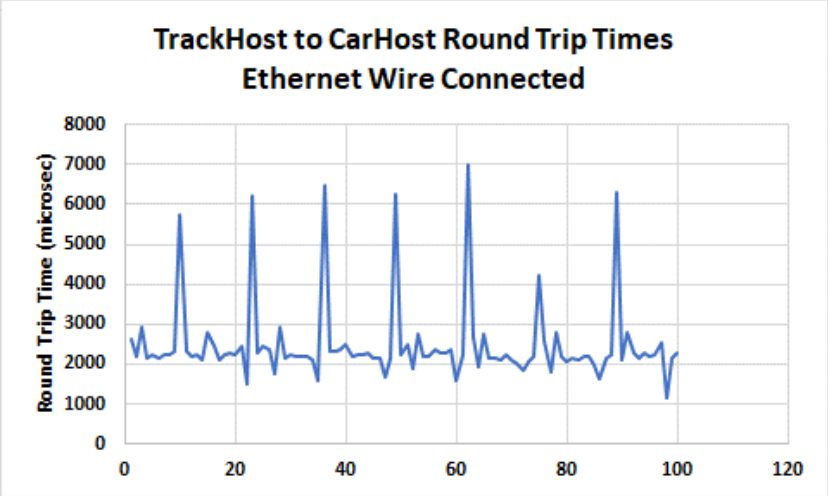


Figure A6 Linux to Linux Wired Propagation Delay

Surprisingly the above results for Ethernet connection are better than the Serial port with a one-way trip time of between 1 and 4 msec. This proved that there was nothing wrong with the code, but a wireless solution was the preferred mechanism between the moving carriage (CarHost) and the stationary Trackside (TrackHost) laptop.

Hence other wireless alternatives were investigated and a pair of Digi XBee S2D Serial ZigBee devices were tested. These work at the same frequency as the WIFI (2.4GHz) and at a published data rate of up to 25Kbps, they also provide a serial peer-to-peer interface via USB connectors rather than sending signals through a WIFI router. They are programmed using Hayes AT commands using a specific Digi application (XCTU). A picture of one of the devices is shown on figure A7 below.

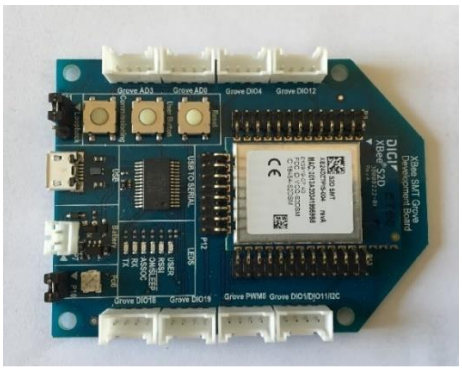


Figure A7 Digi XBee S2D USB Serial Interface Card

Results for typical round-trip times are shown on figure A8 below.

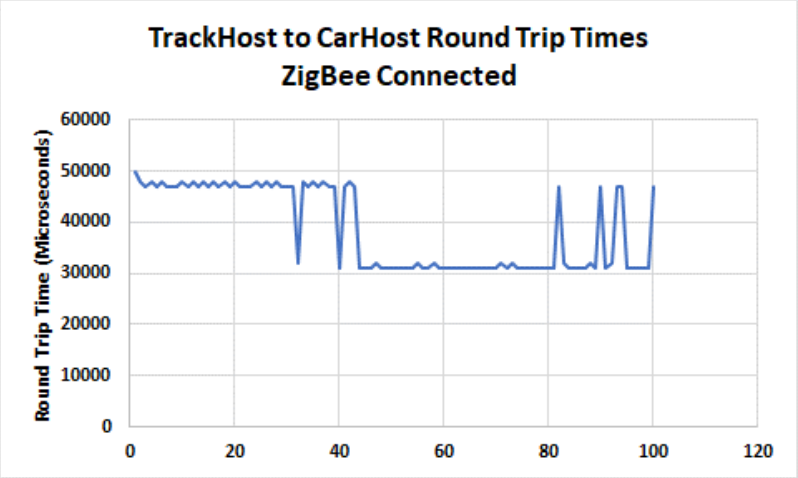


Figure A8 ZigBee Round Trip Times TrackHost to CarHost

As shown, the round-trip times varies between 30 and 47 msec and is worse than the nominal WIFI round trip times of 20 msec. However, both devices are connected to Linux via USB-2 serial interfaces so there are also two serial delays out and back (4x3=12msec) plus the wireless propagation delay between end points. However, this does not account for the rather poor data rate (400-600 bps) observed on test.

The ZigBee system does not appear to suffer from the very large variations in latency seen on the WIFI tests. If the above results are typical of actual behaviour on full system test, then they are marginally acceptable with a propagation delay between 15 and 25 msec for a system operating at 33.33msec (30Hz).

There is one more connection mechanism not covered so far this being the Linux FIFO pipe between CarHost and the Depth Camera program CarCam. This is shown on figure A9 below.

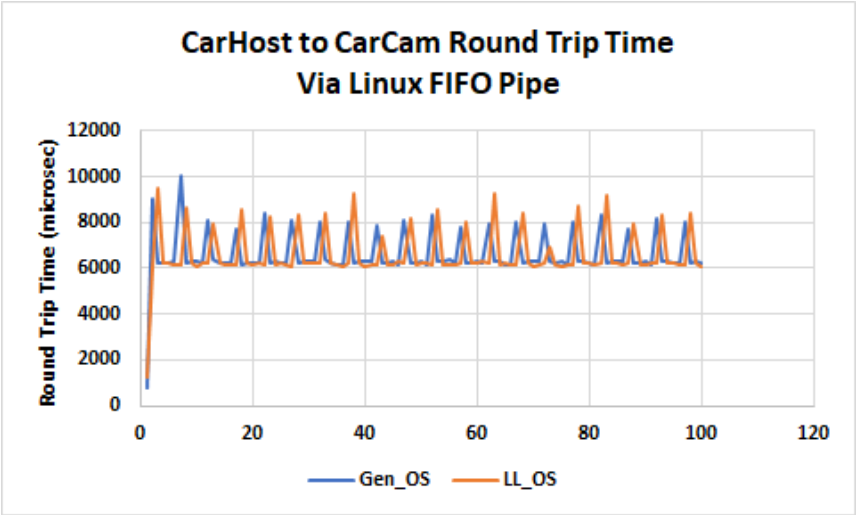


Figure A9 CarHost to CarCam Propagation Delay

As can be seen the above plot implies a one-way propagation delay of 3 to 4 msec. Though surprisingly slower than the wired Ethernet connection it is perfectly acceptable for this system.

A4 Real-Time Embedded Computing Lessons

As someone with a lifetime background in general purpose computing, it came as something of a surprise (and not a particularly pleasant one) to find how optimised Embedded computing hardware platforms were in comparison to a typical and not particularly performant general-purpose laptop.

Typical Embedded processors are in general much smaller (less RAM and storage) and much slower CPU speeds measured in MHz rather than the GHz of typical laptops and desktop machines.

Embedded systems are generally optimised for application to specific hardware control tasks and are typically better furnished with Digital and Analogue I/O interfaces than are general purpose systems.

It is highly questionable to utilise a general-purpose operating system such as Linux in a real-time embedded application. Its use will invariably require performance optimisations designed to disable its default behaviour and result in sub-optimal use of the computing hardware and in some instances the operating system will simply get in the way with its default CFS scheduling system.

The key problem is one of computer performance. To make Linux work in a real-time application at 30Hz requires a processor far more capable than would be needed from a dedicated real-time OS given an equivalent set of tasks.

The choice clincher for this project was that the Intel RealSense Depth Cameras used, the NVIDIA GPUs and the Arduino and C++ code IDEs were all only supported on popular computing platforms such as Microsoft™ Windows® and Linux. Linux was therefore chosen on the grounds of support within the embedded community, however to achieve an acceptable level of real-time performance these systems demanded far higher processor resources and CPU speeds than were typically found available in the Embedded hardware market place.

Hence quite a lot of effort was expended finding an optimum configuration for a Linux based real-time computer system, all be it operating in soft-real-time mode. It was never anticipated that this would be quite such a challenge.

Appendix B ServoTrack Sensor Calibration

B1 Calibration of the ServoTrack Carriage Table

For table pitch and yaw calibration the Pololu control program “Maestro control centre” obtained from the Pololu web-site is used. The laptop is then directly connected to the Pololu servo via USB. Note that the digital servos used operate at 300Hz not the standard 50Hz and the serial-over-USB link is set to 115200 baud not 9600. The simple calibration test setup is shown on the following figure B1.

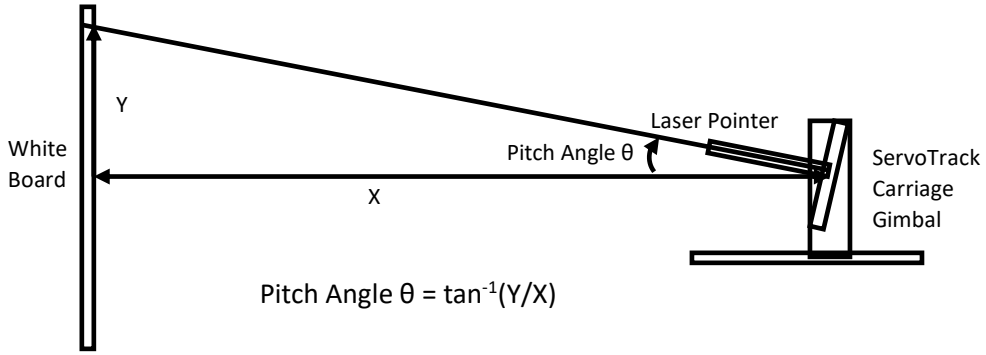


Figure B1 ServoTrack Calibration Test Setup

A class 2 laser pointer pen is attached to the camera mounting and pointed at a white board 2.2m (X) from the table gimble rotation axis and orthogonal to it. The table is then commanded in small increments in pitch using the Pololu control program (the increments used are PWM microseconds). After each small increment the distance Y from the reference datum (established using a sprit level) to the new position pointed to by the laser was measured and recorded along with the voltage from the Pitch sensor. The procedure for the Yaw axis was similar. The data from these two tests was post-processed in Excel and the resulting figures B2 and B3 below are produced for the pitch and yaw axes calibrations respectively.

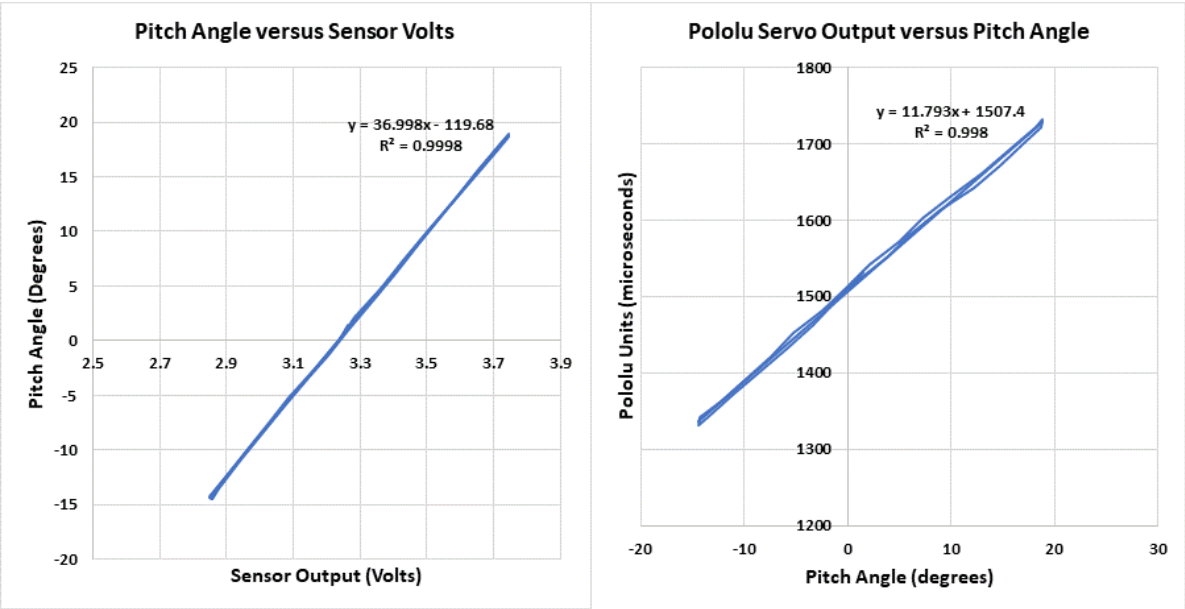


Figure B2 Pitch Axis Calibration Curves for Pitch Sensor and Servo

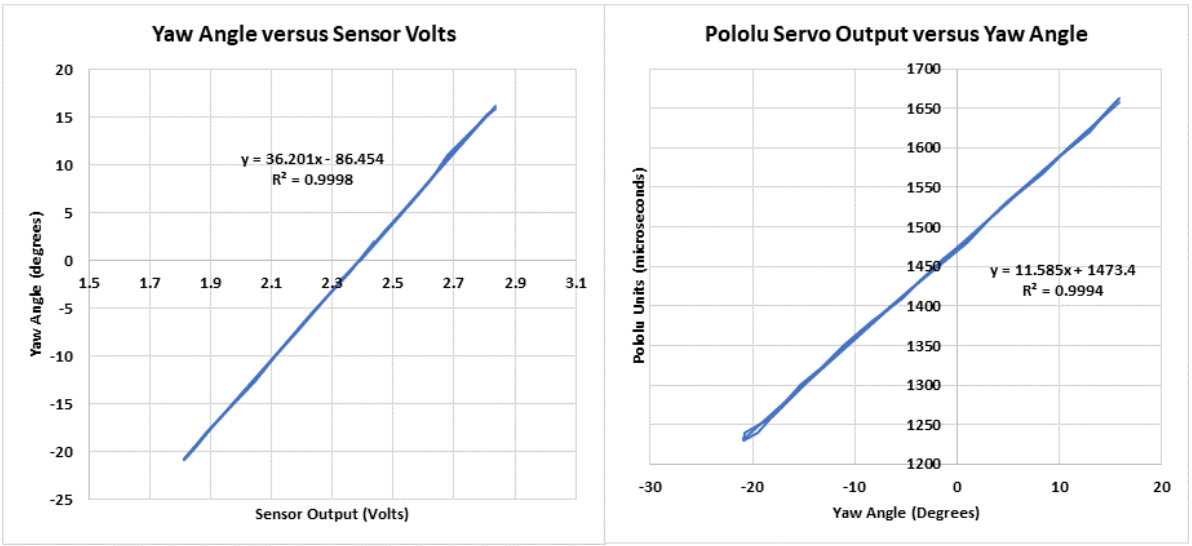


Figure B3 Yaw Axis Calibration Curves for Sensor and Servo

The calibration data shown on the above plots were then programmed into the Arduino (CarDRV) so that a given demand angle in degrees is translated into the correct servo input to achieve the required position in degrees. There is no feedback of sensor position to the servo to correct for friction. Similarly, voltage output from the sensor is converted into position actually achieved. Note that these RC-servos contain their own internal servo-loop, the sensors on the test-rig are to provide accurate camera position data not servo-control. The following figures B4 and B5 shows output from the pitch and Yaw axis to a slow 20-degree sine wave input at 0.5 rads^{-1} respectively. There is clear evidence of nonlinear (friction dominated) behaviour from both servos especially at the turning points. However, the pitch and yaw sensors provide accurate tracking of actual position and this is sufficient for the project requirements.

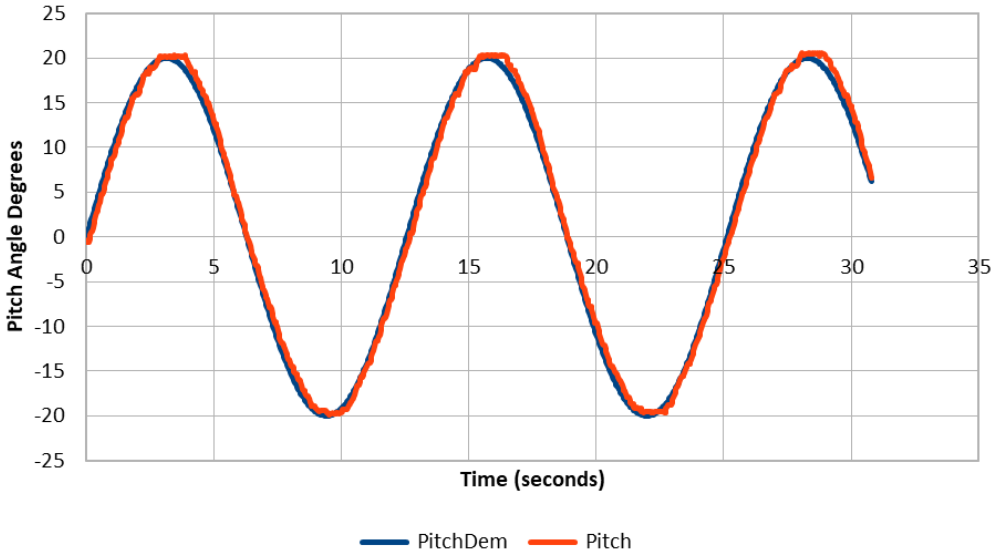


Figure B4 ServoTrack Pitch Axis Sinusoidal Response

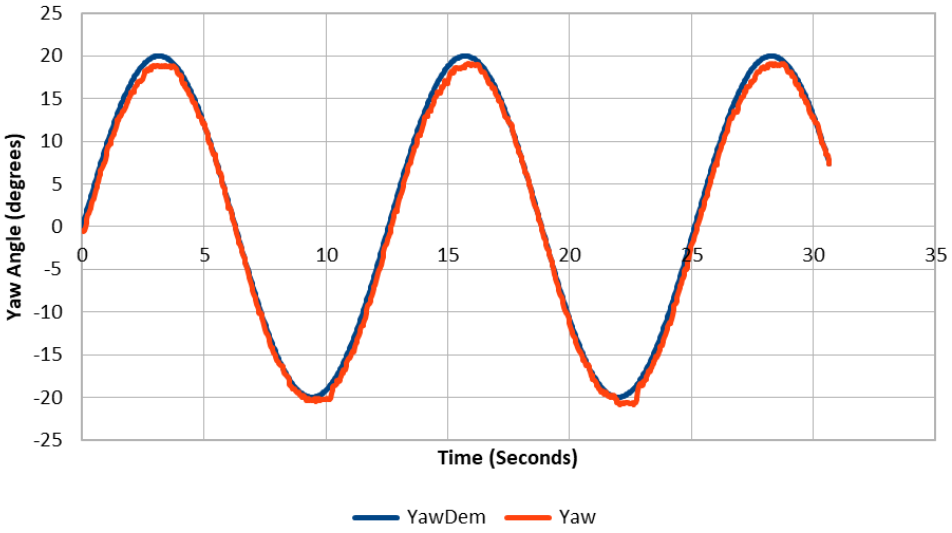


Figure B5 ServoTrack Yaw Axis Sinusoidal Response

In addition to the above tests, a separate software verification test of the implemented calibration data was performed using the laser pen calibration setup shown above. In this test system output for a given demand was compared with physical measurements for a series of step demands on each axis. These results are summarised on the following tables B1 and B2.

Step	Sensor Output	Delta output	Measured Output	Error
deg	deg	deg	deg	deg
0	-0.637	0.0	0	0.0
5.0	4.765	5.4	5.4	0.0
10.0	9.679	10.32	10.36	-0.04
15.0	15.298	15.94	15.78	-0.16
20.0	20.682	21.32	21.0	-0.32

Table B1 Pitch Axis Verification Test

Step	Sensor Output	Delta output	Measured Output	Error
deg	deg	deg	deg	deg
0	-0.49	0.0	0	0.0
5.0	-6.252	-5.76	-5.86	-0.1
10.0	-11.36	-10.87	-10.98	-0.11
15.0	-16.03	-15.54	-15.89	-0.35
20.0	-20.85	-20.36	21.0	-0.64

Table B2 Yaw Axis Verification Test

In both the above tests the initial offset was removed from the program output (to give a delta output) as this was the zero-datum starting position for each test. As can be seen both outputs remain within an error of less than 1 degree over the range tested when compared with the measured angles. This level of error is considered acceptable given the limitations of test measurement.

B2 Track Position Calibration

Calibration of the track linear motion was straightforward owing to the simplicity of stepper motor operation. These are micro-stepping devices and thus send a large number of pulses for any give distance required. The stepper motor pulley drive wheel can be changed for a number of different sizes but by repeated measurement for a large number of pulses it was found that, for the 20-tooth pulley actually used, a distance of 0.0186mm per pulse was obtained. The following table gives measurements for the various other drive pulleys available. These numbers have been verified by test.

Teeth on Drive Pulley	Distance per step (mm)	Steps/mm	Distance per Rev of Drive Pulley (mm)
20	0.0186	53.7	60.0
30	0.0297	33.67	90.0
36	0.03348	29.87	108.0
48	0.04464	22.4	144.0

Table B3 Stepper Motor Drive Pulley Dimensions Available

To verify the that the TrackDRV software was working correctly a number of tests were carried out using a Bosch GLM40 industrial distance measurement device. This class-2 laser measures distance with a stated accuracy of +/-1.5mm. Performing repeated positioning tests over the full range of the track the accuracy of positioning along the track was seen to be very repeatable within +/-1mm.

B3 Axis Cross-Coupling and Alignment

There is a further issue of cross-coupling between motion of the carriage along the track and pitch and yaw motions resulting from small variations in position of the carriage along the track. Although the track is smooth and linear, the wheels/axles of the carriage allow for some free-play both laterally and in yaw. There is also a small amount of pitch motion evident. To assess this the laser pointer was attached to the camera mounting point and the carriage commanded up and down the full length of the track.

The laser beam was seen to move both vertically and horizontally as the carriage moved along the track and a rough bounding box was drawn as illustrated on figure B6 below.

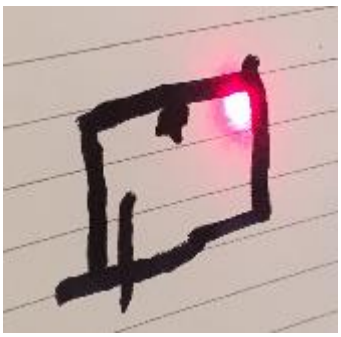


Figure B6 Bounding Box on Pitch and Yaw Motion due to Carriage Linear Motion

Although the carriage can be displaced manually by a couple of degrees in yaw, the drive mechanism effectively centres the carriage very well and the cross-coupling uncertainty is of the order of +/-0.375 degrees in yaw and +/-0.25 degrees in pitch. These are both within acceptable limits.

Finally, being a hand-built system, the build quality of the servo-table and carriage assembly is not perfect. Any misalignment of the gimbal axis bearings will lead to rotation of the gimbal axes system

relative to true earth axes. To check this possibility a simple test using the laser pen mounted at the camera attachment point was carried out. In this test the ServoTable was exercised in pitch and yaw and the track traced by the laser pen was marked on paper taped to an adjacent wall. A spirit level was then used to mark the true vertical and horizontal lines. This is illustrated in the following two images in figure B7.

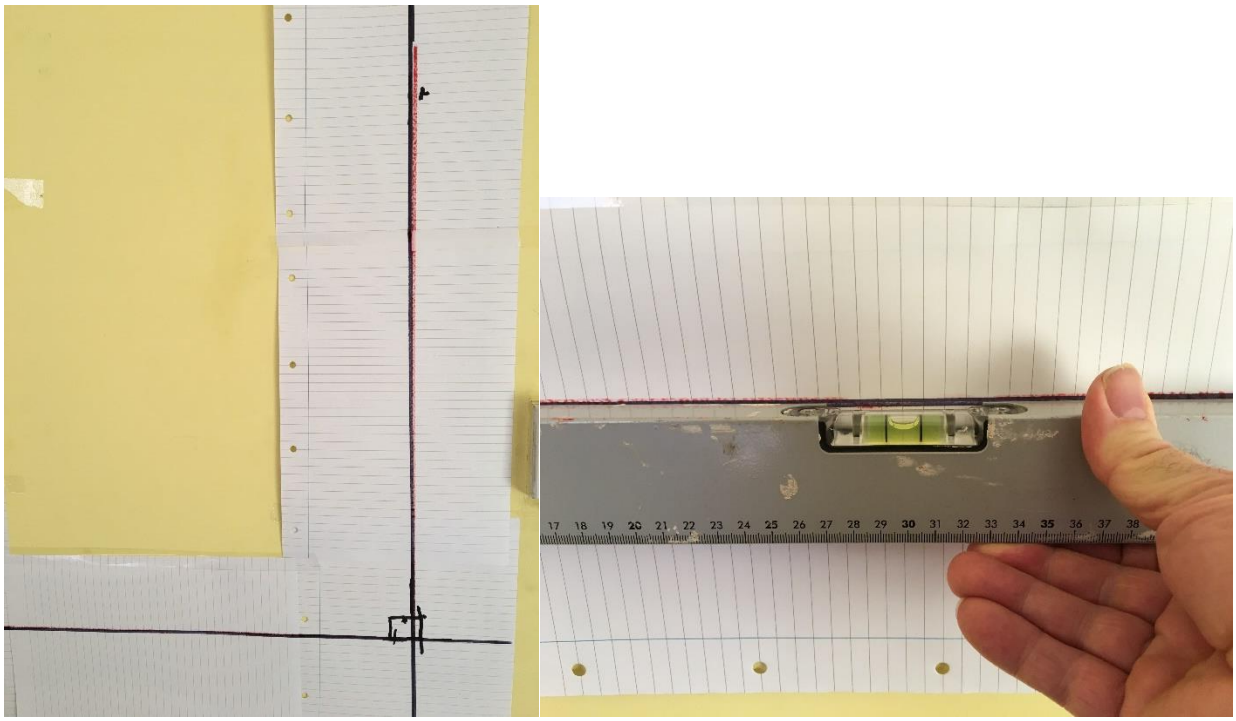


Figure B7 Testing for ServoTable Gimbal Alignment with Earth Axes

The black lines are the track of the Laser pen and the red lines are the vertical and horizontal lines as indicated by the spirit level shown in the right-hand image. From measurements taken in this test, a small 0.43-degree roll discrepancy between the two axes systems was found, this is evident on the left image. This is considered within acceptable limits for the purpose of this project. There is little point in trying to correct such small discrepancies, as there will always be minor alignment errors of this nature with such a movable test rig. The important point being that the test equipment is experimentally verified such that errors are understood and within acceptable limits.