Massachusetts Institute of Technology
Lincoln Laboratory

Worcester Polytechnic Institute

Collaborative Robotics Heads-Up Display

Major Qualifying Project

Christopher Bove (WPI, RBE)
Alexander Wald (WPI, CS)

Advised by:
William Michalson (WPI, ECE)
Mark Donahue (MIT LL, Control & Autonomous Systems Engineering)
Jason LaPenta (MIT LL, Control & Autonomous Systems Engineering)

November 14, 2016

This material is based upon work supported by the Department of the Navy under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of the Navy.

Lexington Massachusetts

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Page

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF TABLES

GLOSSARY

LL	Lincoln Laboratory
MIT	Massachusetts Institute of Technology
MQP	Major Qualifying Project
WPI	Worcester Polytechnic Institute
AR	Augmented Reality
ARM	Acorn RISC Machine
CPU	Central Processing Unit
CV	Computer Vision
DOF	Degrees of Freedom
DSO	Direct Sparse Odometry
EKF	Extended Kalman Filter
FOV	Field of View
GPGPU	General Purpose Graphics Processing Unit
GPS	Global Positioning System
Hector SLAM	Lidar-based SLAM algorithm
HUD	Heads-Up Display
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
IR	Infrared electromagnetic radiation
KF	Kalman Filter
LIDAR	Light Detection And Ranging
LLIVE	Lincoln Laboratory Interactive Virtual Environment
LSD SLAM	Large Scale Direct SLAM
MOCAP	Motion Capture
ORB SLAM	Oriented FAST and Rotated BRIEF SLAM
Pose	Position and Orientation
RAM	Random Access Memory
RANSAC	Random Sample Consensus

GLOSSARY
(Continued)

ROS	Robot Operating System
RVIZ	Robot Visualizer
SLAM	Simultaneous Localization And Mapping
SSD	Solid State Drive
TF	ROS Transform Library
UAV	Unmanned Aerial Vehicle
URDF	Universal Robot Description File
VRPN	Virtual Reality Peripheral Network

1. ABSTRACT

Achieving a robust position and orientation estimate is crucial for intuitive interaction with autonomous systems, especially through augmented reality interfaces. However, available passive localization methods in GPS-denied environments do not suffice. This project loosely coupled inertial and visual sensors by modifying the monocular ORB SLAM algorithm. Data collected from LIDAR and motion capture was used to evaluate the realized system. ORB SLAM code was analyzed and performance profiled for real-time implementation. SLAM scale uncertainty was corrected with inertial data, and scale drift correction was attempted by modifying an internally-optimized motion model. A more accurate position estimate was achieved, and additional work can improve precision, robustness, and execution speed.

2. ACKNOWLEDGMENTS

Our work would not have been possible without the support and guidance from our advisors at MIT Lincoln Laboratory and WPI. Mark Donahue of Control and Autonomous Systems Engineering, assisted us greatly in scoping the project, solidifying our approach, and pushing us to see the bigger picture. Jason LaPenta, also with Control and Autonomous Systems Engineering, provided invaluable technical knowledge and insight to help us form a useful, improvable product. Finally, Professor William Michalson from WPI provided an excellent perspective that drove our team to ask difficult questions and strive to produce the best research possible in our limited time frame. Without these three gentlemen, this project would not have produced such a refined and relevant approach.

3. EXECUTIVE SUMMARY

Augmented reality (AR) has grown increasingly popular in recent years, allowing people to interact with digital data in more intuitive ways. Heads-Up Displays (HUD) further this enhanced interface by allowing people to note their real environment while still interacting with AR systems. However, for wearable AR devices, such as HUDs, to properly function, they require the pose (position and orientation) of the wearer relative to the environment. This established pose allows visual elements to be properly overlaid on real-world equivalents, or extra-sensory data, like that supplied by a robot, to be added in a sensible manner to the AR world. Low cost, six or nine axis Inertial Measurement Units (IMUs) are capable of providing accurate orientation, and GPS is normally sufficient where plus-meter accuracy is acceptable. Indoor environments, tunnels, forests, or other areas with an obstructed view of the sky prevent the utilization of GPS as a precise and reliable source of position. Numerous other methods can localize a user, but each has its own setbacks.

Passive and active sensors are available to obtain information from the world. Active sensors, such as laser range finders, are very accurate and directly provide positional data. In mobile systems targeted for the field, passive sensors with lower SWaP (Size Weight and Power) and no energy emission are of great advantage, though positional information is sometimes more difficult to obtain. IMU acceleration can be double integrated to provide position, but this estimate will drift chromatically due to non-zero bias errors in the accelerometer data. The quadratic relationship between position and acceleration implies that small, constant accelerations will result in a large displacement. Visual sensors and algorithms can be used to provide position by determining object or feature movement between frames, which is referred to as visual odometry. This type of odometry, like others, drifts as a function of position. Simultaneous Localization and Mapping (SLAM) algorithms attempt to correct this drift through place recognition and map building, allowing the pose chain loop to close when a familiar location is perceived again.

A single camera cannot determine the scale of its motions without knowing the size of objects within its view, and this represents one of the issues with monocular visual SLAM. Stereo vision with two parallel cameras and a known baseline between them allow scale and distance to be determined, but in systems trying to reduce SWaP, it would be optimal to utilize the fewest components possible. Thus, an ideal solution would be to use a single camera and an IMU to determine the pose of a user on a portable system. At MIT Lincoln Laboratory, a prototype helmet mapping system has been used as a data collection testbed to examine methods of combining algorithms and techniques to achieve this result. Eventually, these developed algorithms can be implemented on a pair of HUD glasses which house a camera and IMU. Past research and experience show that a monocular, feature-based SLAM algorithm called ORB SLAM produces promising results in forested environments, and this project attempts to understand its implementation and ways to improve its behavior.

The larger proposed system design implementation, Figure 1, seeks to loosely couple inertial measurements with visual SLAM to metrically fix scale and mitigate scale drift in ORB SLAM and also combine other estimates to achieve a consistent and robust pose estimate in multiple

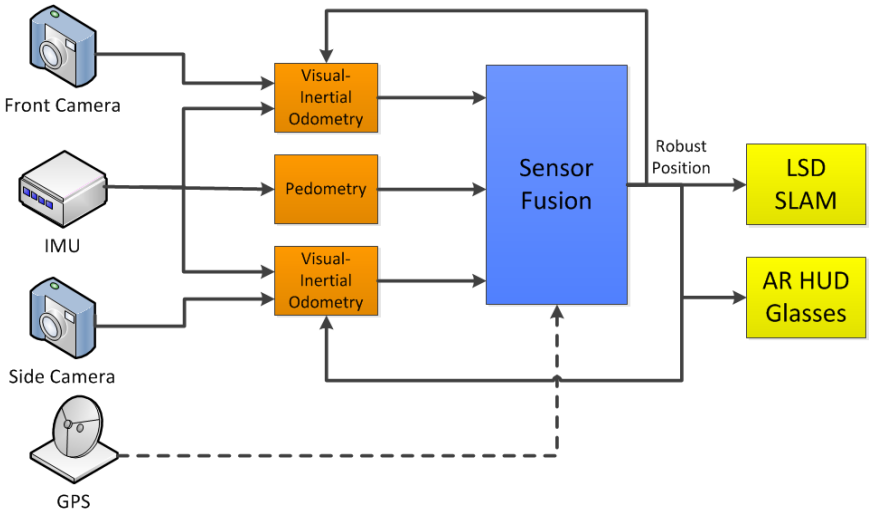

Figure 1. "Big-picture" system design approach. Project focuses on developing visual-inertial module.

failure cases. Within the visual-inertial module, shown in Figure 10, a number of components and modifications had to be created. Several procedures were followed to achieve the desired system.

1) Tests were conducted in a motion capture studio to evaluate the performance of ORB SLAM, accuracy of the proposed solution, and determine the validity of utilizing LIDAR for truth in outdoor tests. **2)** Transformations were developed that allowed sensor readings and positions to be exchanged between visual and inertial systems. **3)** Gravity and DC bias compensation separated acceleration values from biases to enable a double integration position node to determine system position from measured accelerations. **4)** The maximum rotational rate of the gyro was determined to ensure performance could be achieved, and a method to test ORB SLAM's maximum rotation rates was created. **5)** ORB SLAM source code was studied to determine locations where corrections could be applied. **6)** Scale was corrected by injecting IMU or mocap-derived pose estimates into ORB SLAM and scaling the internal positions and map points. **7)** Drift reduction was attempted by injecting absolute orientation and position derived from mocap and the IMU. **8)** Runtime performance of tasks in ORB SLAM thread call hierarchies was studied in order to provide insight and recommendations for eventual real-time implementation on an embedded processor.

The collected data illuminated significant issues and proved to be useful in checking system accuracy in all six DOF, and this was previously unachievable when only LIDAR was used in the outdoor forest environment. Tests indicated that ORB SLAM scale drifted most significantly after rotations occurred, and loop closure greatly improved the results. Transformations between systems were validated by the properly signed movements shown in Figure 2, but ORB SLAM suffered from scale uncertainty. The maximum rotation rate of the gyro was determined to be 8.5 rad/sec, but the EKF in the INS continued to produce correct orientations above this rate. ORB SLAM was less affected from motion blur than it was from reduced frame overlap, but precise measurements were unable to be determined.

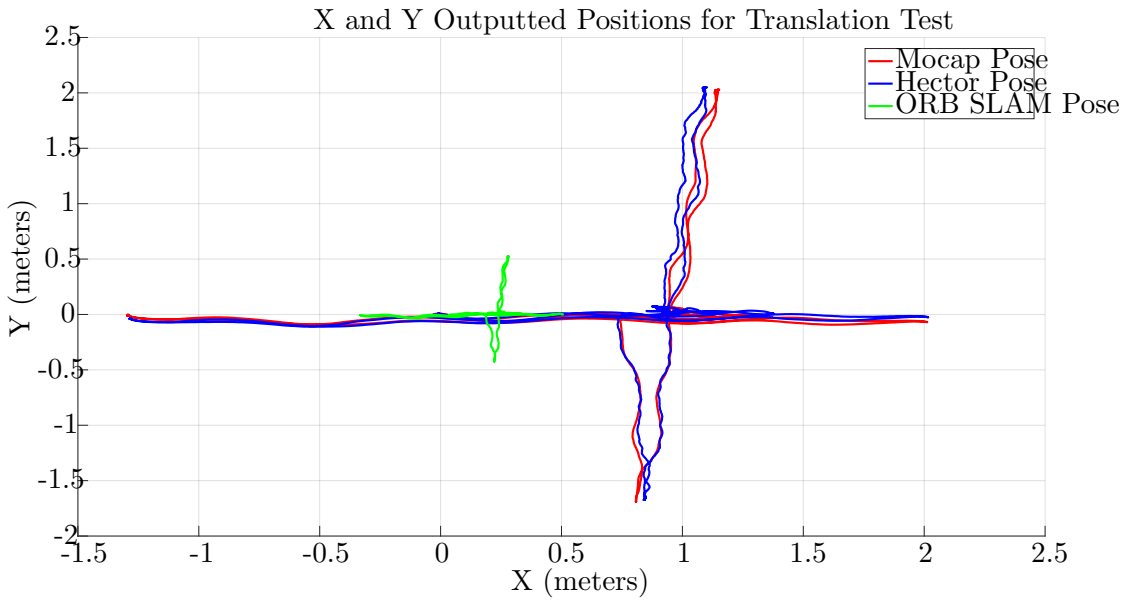

Figure 2. X, Y positions of truth systems and ORB SLAM without scale correction applied.

Scale uncertainty was corrected successfully, as shown in Figure 3, using truth data. IMU-derived scale estimates were also accurate when translational movement began quickly, but if significant integration error accumulated, the scale estimate was poor. In many data collections, the helmet did not move for a long period of time at the start, so more sophisticated corrections would be necessary to reset the inertial dead reckoning estimate immediately before the helmet began movements. This would bypass drift that accumulated before movement was detected. In addition, there was no feedback to incorporate further scale adjustment, thus the scale tended to diverge over large tracks.

Drift correction produced mixed results, as seen in Figure 4. Truth data somewhat improved the estimate, but the drift was not reduced, as indicated by the small spiral towards the circle's center before loop closure snapped the position back to zero. If drift correction was successful, this jump should not have occurred. Large discrepancies existed when the algorithm tracked through previously mapped areas since the injected poses differed from the poses derived from examining the previously-saved map points. This indicated the algorithm was successfully modified, but a different approach was necessary to fully correct drift.

The runtime performance analysis revealed many details concerning the processing hierarchy and bottlenecks in the ORB SLAM system. Figure 5 shows processing time results from the Tracking thread. It was discovered that roughly 85% of the processing time was attributed to ORB feature extraction, and this extraction, utilizing FAST and BRIEF feature recognition, could be replaced with GPU-optimized calls in the NVIDIA VisionWorks API on supported hardware, such as the Jetson TX1 board. [1]

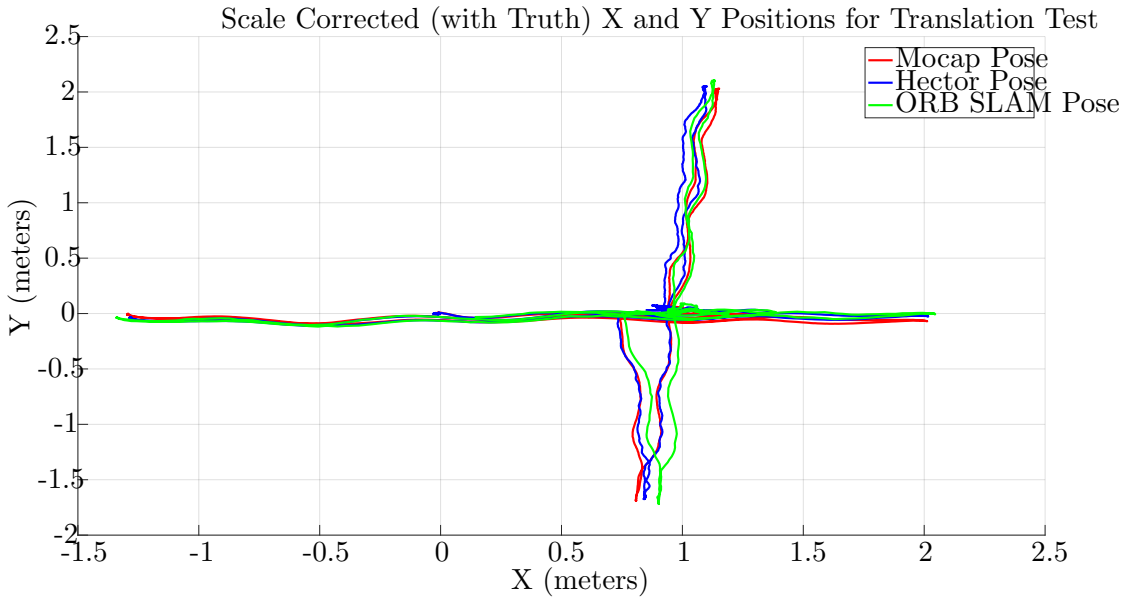

Figure 3. X,Y positions of truth systems and ORB SLAM with scale correction applied.

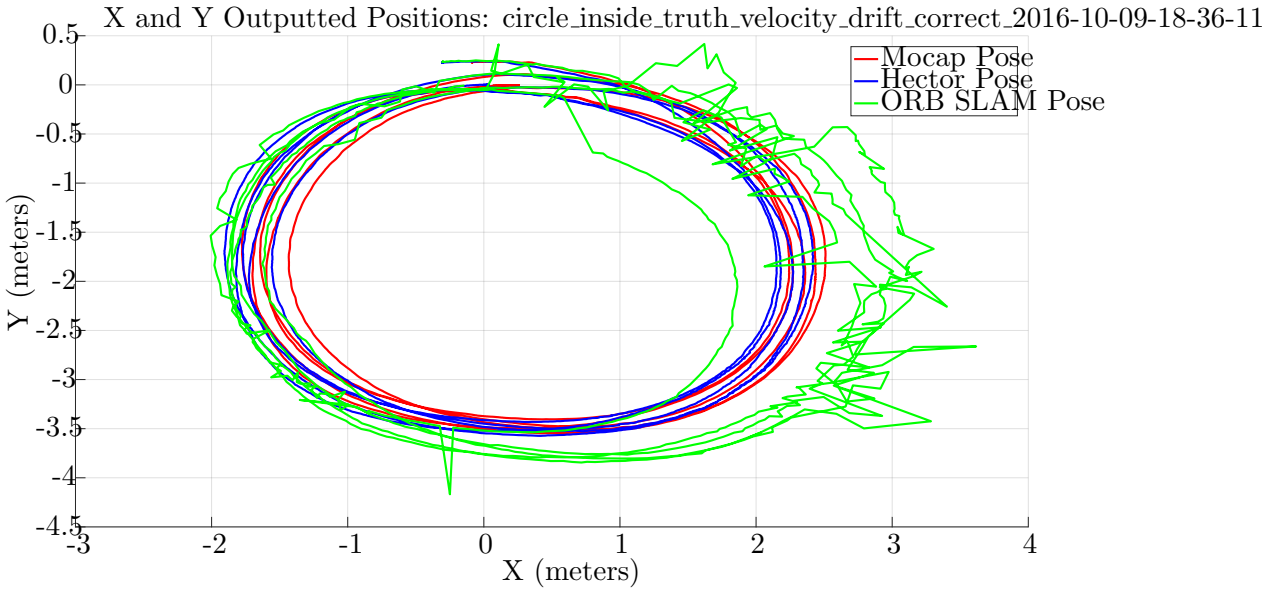

Figure 4. X,Y positions of truth systems and ORB SLAM with scale drift correction applied to circle test.

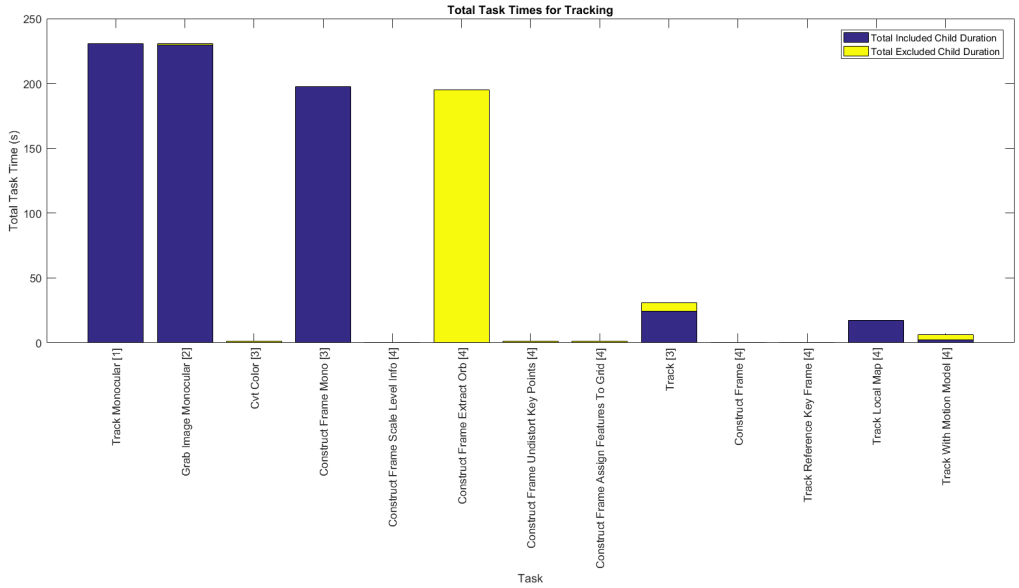

Figure 5. Total subtask times within the analyzed call hierarchy of the ORB SLAM Tracking thread were aggregated from four different trials. The X axis displays subtasks of the Tracking thread while the Y axis displays the total amount of time spent on each task. The blue bar sections indicate run time that was further subdivided as child tasks. The yellow bar sections indicate run time that was not divided further. Therefore, a purely yellow bar represents a leaf in the analysis call hierarchy, and the sum of all yellow sections results in the total time spent on thread cycles. A bar with blue and yellow sections means that only some of its subtasks were analyzed further.

This project demonstrated that it was possible to augment ORB SLAM with external sensor information to correct system shortcomings and achieve a more accurate position estimate without GPS for use in AR systems. The thorough data collection provided a good baseline for determining system characteristics and examining failure cases. Scale correction was adequate and improved the utility of ORB SLAM as a pose estimate for AR systems, but the approach needed further refinement and feedback. Drift correction indicated the algorithm was successfully modified but demonstrated failure to justify differences between computed poses in ORB SLAM based on the internal map and externally-calculated poses. This discovery can guide further research to reduce this drift error by applying more sophisticated data fusion outside of ORB SLAM. Performance evaluation strongly suggests real-time implementation on the Jetson is possible through simple replacement of feature extraction calls with GPU-optimized calls. Notable progress was achieved towards the goal of attaining an accurate pose estimate with passive sensors to enable sophisticated augmented reality applications to interact with autonomous systems.

Future work should focus on improving the drift correction by determining a better method of fusing sensory data on a frame-by-frame basis. Likely, a non-linear Kalman Filter could be applied to the output of ORB SLAM to correct scale constantly from other sources. Better integration of the IMU dead-reckoning estimate is necessary to remove errors caused by positional drift before the ORB SLAM system initializes. This could be attained by reseting the estimate periodically if sufficient velocity is not detected visually. It is also possible that a lighter-weight version of visual odometry should be considered for rapid execution time and tighter coupling with an IMU, similar to the sliding window filter employed by Google Tango algorithms. [2] Further consideration is necessary for properly integrating large octomaps and displaying them on a HUD utilizing the new pose estimate.

4. INTRODUCTION

While robotics platforms have advanced significantly in capability, a gap remains between human understanding and autonomous reasoning. This requires humans to remain in the loop for high level decision making, yet the tools available to perceive robots' environments are typically limited to mouse and keyboard interactions on a traditional computer. To enable rapid deployment, control, and intuitive supervision of autonomous systems in the field, a better collaboration bridge must be developed. However, this bridge requires a mutual understanding of space, objects, and movements of the user and robotic platform. Creating this understanding remains an active area of research.

To better illustrate this outstanding problem, a scenario is considered where a human is sent into a GPS-denied forest to retrieve a dropped package with the assistance of an autonomous UAV. The UAV performs a surveillance fly-through of the forest, providing a map of the forest and location of the package. The person is tasked with navigating to the package as efficiently as possible. While a current solution may be to display the forest map and the location of the package on a smartphone or tablet, interacting with the display removes the user's attention from the environment and reduces tactical awareness. Additionally, it requires the user to localize themselves on the map before beginning their search. A proposed solution is to implement a helmet-mounted system to localize the user without active sensors in the forest, correlate their position with the robot's map, and use his/her localized position to display the object's location on an AR HUD. Visuals of the target package would be superimposed over the corresponding object in the scene, allowing the user to quickly see the package through the woods and navigate towards it. Further aids such as navigation cues and highlighted routes could be added.

In order for this proposed system to correctly display these markers to the user, it must maintain localization of the wearer with respect to a world map. Because GPS is sporadic, imprecise, or potentially unavailable in some environments, a position estimate derived from other means must be maintained throughout the exploration. Sensors such as INS and visual cameras have been utilized extensively for such pose (position and orientation) estimates, and when mounted on the helmet, these sensor systems could potentially glean an acceptable pose estimate without GPS. Unfortunately, neither one of the sensors can create or maintain an accurate position estimate without utilizing other data from the world, which is perhaps unavailable. Sensor fusion techniques combine data from these sensors, complementing the advantages of each system to estimating pose with better accuracy than the individual sensors can provide.

The goal of this project is to achieve robust pose determination using passive sensors in GPS-denied environments to enable localization-driven augmented reality for interacting with autonomous systems. The focus is to obtain a position estimation system that can be implemented on an embedded system connected to the helmet with minimal further effort. More detailed maps of the environment can be created by utilizing the accurate position estimate, and the user's HUD would properly display AR information relative to the user's true pose. Once this system runs in the field, further work can be conducted to relate the robot and user poses to fully bridge the collaboration gap between autonomous systems and humans.

5. BACKGROUND

5.1 SENSORS FOR LOCALIZATION

In the purest sense (no pun intended), sensors can be characterized as either active or passive. Active sensors emit energy into their environment to detect its characteristics while passive sensors absorb ambient energy from the world. In localization, some of the most widely used active sensors are range finding ones. Range is typically found either through ultrasonics, sonar, radar, or light. Typical mapping equipment utilizes LIDAR, a laser range-finding device capable of quickly generating 2D range data in a wide variety of environments. Passive localization sensors generally include gyros, accelerometers, magnetometers, GPS receivers, and cameras. Gyros, accelerometers, and magnetometers are typically packaged into an Inertial Measurement Unit (IMU), and Inertial Navigation Systems (INS) usually include an IMU and GPS to obtain a six DOF pose estimate. Cameras may at first seem an odd sensor platform for position estimation, but optical computer mice prove an exceptional example of utilizing imaging sensors to quickly track visual features and estimate movement. Table 1 shows some of the major differences between LIDAR, an active sensor, and an INS system, a passive sensor suite.

TABLE 1

Comparison of LIDAR and INS Sensor Systems

Active Sensor - LIDAR	**Passive Sensor - INS**
Benefits	**Benefits**
• Directly compute positional data • Operates very quickly • High accuracy and precision	• **No energy emission**[1] • Typically low cost, weight, power • GPS provides high accuracy positioning • IMU provides 3 DOF orientation estimate
Drawbacks	**Drawbacks**
• **Projects IR lasers into environment** • Typically higher cost, weight, power consumption • Reflective materials can cause failures • No position estimate when items are not in range	• GPS can be jammed, interrupted • IMU orientation can be confounded by magnetic interference • Position drifts wildly if integrating IMU acceleration • GPS is only precise to about a meter

[1]Neglecting the emission of heat and small RF noise.

The difference in energy expression between active and passive sensors has two main side effects. **1)** Active sensors typically utilize more energy than their passive counterparts. For instance, a Hokuyo LIDAR consumes nearly seven times more energy than a USB camera.[2] In a power-limited mobile system, this attribute alone can be prohibitive in implementing the sensor, though further accounting must be made for power required to process sensor data. **2)** Active sensors are more easily detectable since they emit energy. This can be of concern in circumstances where stealth is key. LIDAR produces excellent data, but an observer with the correct sensing equipment could detect its IR laser emissions.

For these two reasons, passive sensors are preferred for mobile applications requiring a certain level of discreet operation and run time. However, passive sensors tend to be more susceptible to environment noise than active sensors and provide more indirect information, which necessitates sophisticated algorithms to produce exceptional pose data.

5.2 LOCALIZATION AND MAPPING ALGORITHMS

Recent and intense developments have greatly increased the capabilities of Simultaneous Localization And Mapping (SLAM) algorithms for many different sensors. Methods utilizing range-based sensors are well established, with the LIDAR-based Hector SLAM able to produce accurate position and map data in real time with low computational resources. [5] Algorithms capable of utilizing camera sensors are gaining increasing interest due to the benefits of using cheaper, passive imaging sensors over more expensive active systems such as LIDAR, as discussed in subsection 5.1. However, there are numerous technical challenges that are yet to be fully solved.

5.2.1 Visual SLAM Systems

With a single camera, image processing is completed through a set of intrinsic parameters that describe geometric relationships between the camera's image sensor and the image projection into the world. Knowing a point of interest on the image plane, these parameters allow a ray to be computed that describes the point in the real world, as shown in Figure 6. [6] However, the distance from the camera to the point is unknown.

When two cameras are employed in a stereo configuration, depth information is computed from a single frame pair using the intrinsic camera parameters and knowledge of the camera baseline (distance between the stereo cameras), as shown in Figure 7. The intersection of both image rays allows the point to be distinguished, with some uncertainty that grows as a function of range. [7]

Thus, with only one camera (i.e. monocular configuration), baseline distances must be estimated over several frames produced at different times to compute a 3-D distance. This produces an optimization problem where, without external information to provide either a baseline or distance to the point of interest, the depth of points is obtained from an estimation of the baseline derived from apparent movement of the camera. Inherently, an absolute scale cannot be determined, since a single camera can only provide a ray to relate a particular pixel in the frame to an actual heading in

[2] A Hokuyo UTM-30LX-EW LIDAR typically consumes 8.4W [3] compared to the 1.25W utilized by a PointGrey Flea3 Color Camera [4].

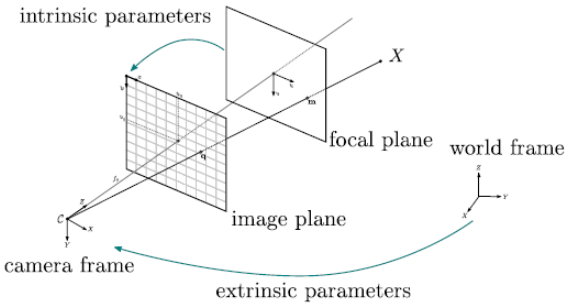

Figure 6. Computing a ray to a point in the world using intrinsic camera parameters. Distance can only be known for an object of known size in the environment through extrinsic parameters.
Source: OpenMVG [6]

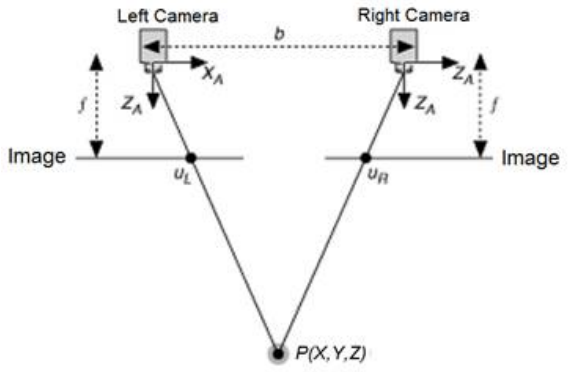

Figure 7. Stereo vision distance computation using intrinsic camera parameters and known baseline.
Source: National Instruments [7]

the world. Given an object of known dimensions, however, its absolute position can be determined with a single camera, but this provides no direct information concerning the absolute positioning of other unknown objects in the frame.

Monocular SLAM systems need to simultaneously estimate camera movement in space and the locations of objects in a map. Two main approaches to this problem are vying for dominance, each with its own drawbacks and benefits. One approach is feature based SLAM, which performs a feature extraction routine on each frame to identify interesting and unique points in the world, stores them in some retrievable format, and recognizes and tracks them in later frames. The second main approach is to skip feature recognition and directly use the pixel intensities for tracking and mapping. Currently, two of the most popular solutions for feature-based and direct SLAM algorithms are Oriented FAST and Rotated Brief (ORB) SLAM [8] and Large Scale Direct (LSD) SLAM [9] respectively, shown running in Figure 8. However, a new Direct Sparse Odometry (DSO) solution is set to be released shortly by the developers of LSD SLAM at the Technical University of Munich (TUM) Computer Vision Group. [10] DSO is likely to replace LSD SLAM as one of the best direct SLAM solutions, though the current implementation of DSO does not correct for full loop closure. A comparison between the SLAM solutions is briefly summarized in Table 2 and in subsubsection 5.2.2.

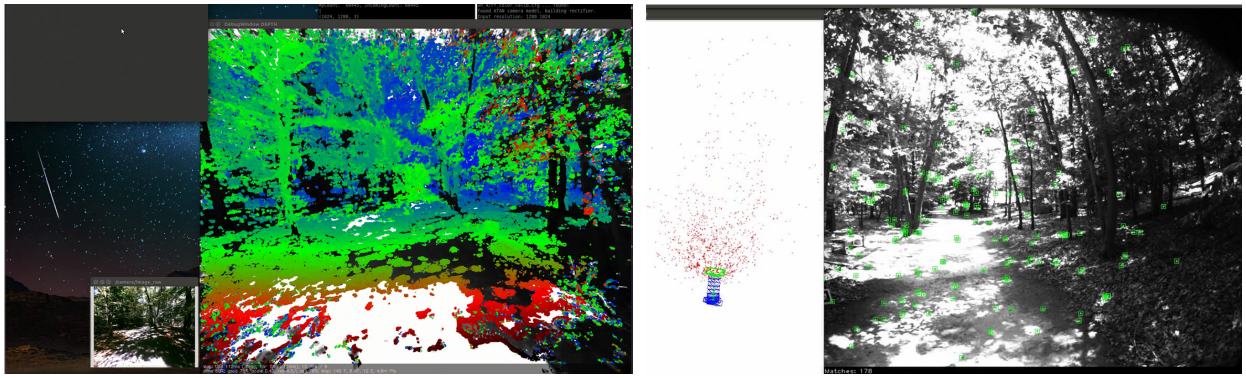

Figure 8. Screenshot of LSD SLAM (left) and ORB SLAM (right) processing data collected in the forest. Colors in LSD SLAM represent depth of points (red is close, purple is far). In ORB SLAM, green marks indicate tracked features, and the map in the middle shows current position in green and currently matched map points in red.

5.2.2 Monocular SLAM Shortcomings

While a given issue with monocular SLAM is that absolute scale of the localization and map are not known, there are other shortcomings that prohibit the system from being a final solution for localization and mapping. First of which is the scale uncertainty previously mentioned. A monocular SLAM system cannot relate movement to real-world units of measurement. In a strictly monocular system, non-camera sensors must be used to calculate a scale. One method is to augment

[3] As internally tested

TABLE 2

Comparison of Monocular SLAM Systems

Property	**ORB [8]**	**LSD [9]**	**DSO [10]**
Tracking	ORB Features	Direct	Direct
Point Cloud Output	Sparse	Semi-Dense	Sparse → Semi-Dense
Global Loop Closure	Included	External Package	External Package
Memory Footprint [3]	1-2GB	4-16GB	Unknown
Main Strengths	Robust, resistant to brightness changes, fast	Semi-dense color point clouds	Very accurate, adjustable pt cloud density, well calibrated
Main Drawbacks	Sparse pt cloud, drifts with no loop closings	large scale drift, slow	Not yet available
Failure Cases	Smooth/featureless environments, false loop closures with indistinct features	Brightness changes, fast movements	Unknown

the visual odometry with information supplied by an IMU, and this process has been performed in various ways. [11] [12]

A second problem is scale drift, which refers to an issue where the estimated scale of the world fluctuates as the camera translates and rotates through the environment. One main contributor to scale drift is image noise, which adds uncertainty that compounds as tracking continues. [13] A way to correct this drift is to detect loop closures in the path, close the loop, then perform scale-drift aware optimizations to regain straight paths. [13] This causes another problem in that the calculated positions can change significantly after a loop closure, which also indicates how far off the system can become. Newer implementations of visual SLAM better calculate and correct scale drift in real time by comparing more frames, expanding models, and optimizing more frequently, as demonstrated in DSO SLAM. [10] However, some amount of drift will be inherent in a visual system until loops can be closed because of uncertainty from camera noise.

Thirdly, disturbances in the image frames themselves can cause visual SLAM algorithms to lose tracking. From tests utilizing datasets in Table 4, there appears to be three main sources of lost tracking events: fast camera movements, sun glare events, and loss of tracking features. When the camera moves quickly, motion blur can occur and reduce the quality of the frames received, negatively impacting the information that can be retrieved from the camera. Additionally, if tracking algorithms cannot execute fast enough to track features between frames (if the features are moving into and out of the frame too quickly), tracking will fail. Periodic bursts of bright sunlight wash out parts of the image and change the intensities in the frames globally, causing issues for direct SLAM approaches.[4] Finally, certain environments devoid of sufficient features or

[4] This is partially compensated for in DSO SLAM with full photometric correction. [10]

intensity gradients fail to provide enough visual elements to sustain visual tracking. In these cases, other methods are necessary for localizing the system during these blackouts.

5.3 SENSOR FUSION

A problem arises when multiple sensor systems are available on one platform and each produces a different estimate of pose: a method must be used to combine these estimates and procure the most statistically viable estimate. Sensor fusion is one such way of deriving information from individual sensor data that exceeds the accuracy of any one sensor's information: "Sensor Fusion is the combining of sensory data or data derived from sensory data such that the resulting information is in some sense better than would be possible when these sources were used individually." [14] This fusion is useful in scenarios where direct sensor information is not sufficient to achieve a certain objective. Sufficiency of individual sensor data is determined from its precision, accuracy, sampling frequency, and measured environmental property.

There are a variety of ways to combine individual sensor values into a robust estimate: "Feature-level fusion involves features that are extracted from different sensor observations or measurements and combined into a concatenated feature vector. Decision-level fusion takes information from each sensor after it has measured or evaluated a target individually." [15] In other words, sensor fusion can occur at a low level by combining raw data of sensors to achieve a better reading before sending that data to further analysis, or a high-level approach could fuse the analyses produced from raw sensor data.

In general, sensor fusion implementations utilize some common techniques to prepare raw sensor data before forming environment parameter estimation. Smoothing is the process of using a series of measurements to estimate the actual state of a variable in the environment, and this smoothing is not necessarily real time. Most filtering, the treating of current measurements according to stored information from previous measurements, can be completed in real time. Prediction is a technique that uses previous measurements and a model of the system being measured to estimate a future state. Kalman Filtering is a technique "developed by Kalman and Bucy in 1960" that uses a discrete-time iterative algorithm to maintain an estimate of environment process variables with minimal noise. [14] By taking into account the dynamics of the observed system and previous sensor data, a Kalman filter can estimate variables in the environment with less statistical error than a solution achieved using the sensor directly.

The fusion of inertial and navigation sensors has been extensively researched due to the promises of increasingly lower cost cameras and IMU's. Studies have shown that visual-inertial navigation systems are quite achievable and effective given proper modeling and characterization of the system. [16] [17] In addition, Google has recently unveiled an Android consumer platform capable of 3D position and orientation estimates in real-time utilizing visual-inertial odometry, now called "Tango." [18] However, the source code is not available for these different implementations.

5.4 VISION PROCESSING ON EMBEDDED SYSTEMS

The mobile phone and computing industries have significantly advanced the capabilities of mobile chipsets in recent years. However, many vision processing tasks remain computationally intensive, and on embedded mobile systems, this can pose an implementation problem. Yet most computer vision tasks are quite adaptable to utilizing General Purpose Graphics Processing Units (GPGPU's), which provide clusters of many small processing cores to enable massively parallelized computational tasks to execute very quickly.

NVIDIA, through their CUDA core architecture and software language, has been one of the industry leaders in providing parallel computing resources. Their VisionWorks library provides a host of vision processing primitives and functions that utilize available CUDA cores and accelerate vision processing pipelines. [1] Their integration of CUDA GPU's into mobile chipsets and development platforms, such as the Jetson TK1 and TX1 boards, have opened up new possibilities in low power computer vision platforms. These developments promise to allow more advanced CV tasks, such as visual SLAM, to be conducted in real-time on low power embedded systems. [19] And indeed, camera manufacturer Point Grey has demonstrated that the Jetson TX1 board is more than capable of fast image capture.[5] [20]

5.5 AUGMENTED REALITY ON HEADS-UP DISPLAYS

Augmented reality is the concept of adding additional information to one's perception of the world. A person using an AR device not only sees the normal environment but also an overlay of additional visual information. A HUD is a user interface that projects extra-sensory information while still allowing the true environment to be seen. The main advantage of a HUD over other interfaces such as a phone or tablet is that a HUD overlays AR components on what the user sees naturally, but other systems require the user to divert their gaze and attention to view the AR content. Using a HUD could increase safety in scenarios requiring a high awareness of surroundings while using AR simultaneously.

5.6 AVAILABLE HARDWARE AND DATA

The Control and Autonomous Systems Laboratory at MIT Lincoln Laboratory has attained several state-of-the-art sensor systems quite suitable for this project and mounted them on a wearable helmet platform, shown in Figure 9. Table 3 shows the equipment utilized on this system. ROS, a software library for interfacing with robotics platforms, was used for handling most sensor drivers and data. The long-range goal of the project at MIT LL is to eventually implement the algorithms being developed in this project onto the ODG HUD glasses themselves, providing localization capability for better AR applications using only the ODG Glasses.

For now, the helmet mapping system represents a research grade equivalent of the ODG Glasses (the glasses contain a 70 FPS camera, IMU, and GPS receiver) that can be used to test

[5] The Jetson TX1 utilizes 30% of its CPU when processing color video streamed at 1920x1200 at 45Hz, totaling 104 MB/s of data. Other configurations have been tested with the Jetson utilizing 27% of its CPU while capturing 356MB/s mono video at 1920x1200 at 163Hz. [20]

Figure 9. Various components of the helmet mapping system. The Jetson board is tethered to the helmet and carried by a shoulder strap.

TABLE 3

Available Hardware Components

Model	Details
NVIDIA Jetson TX1 Development Kit [21]	ARM processor and 256 CUDA cores running Ubuntu 14.04
2 PointGrey Flea3 2.0MP Color Cameras [4]	USB 3.0 interface, 59Hz @ 1600x1200 resolution, 190° fisheye lens
2 PointGrey Flea3 1.3MP Mono Cameras [22]	USB 3.0 interface, 150Hz @ 1280x1024 resolution, 190° fisheye lens
Microstrain 3DM-GX4-45 INS [23]	Integrates GPS, IMU, magnetometer into 6 DOF pose estimate at 53Hz. IMU and Gyro can produce data at 500Hz. The magnetometer produces data at 53Hz and limits ROS driver to this frequency.
Hokuyo UTM-30LX-EW LIDAR [24]	30m range, 40Hz, Used for developing truth data
ODG R-6 AR Glasses [25]	Color 720p HUD display, IMU, 70 FPS camera, running Android 4.4
OptiTrack Motion Capture System	20 IR cameras provide sub-mm position and 0.25 degree angular accuracy

algorithms quickly and compare results to that attained using truth (LIDAR-based Hector SLAM). This also drove some design decisions: while stereo vision solves the unknown baseline problem, the glasses are equipped with only one forward-facing camera, so a solution involving stereo vision was tentatively avoided. Additionally, further study is necessary to determine the effective stereo range of the current camera hardware and show whether that range is useful in typical use cases. Sensor suites, such as the Carnegie Robotics MultiSense SL, are able to achieve a stereo range of 0.4 to 10 meters with a baseline of 7cm, indicating that useful stereo vision may be possible on a helmet-mounted platform. [26]

During the Summer of 2016, as a precedent to this project, the system was developed and used on several data collection runs. Table 4 lists examples of available data sets to test pose estimation algorithms. The data sets include recorded sensor data from the color cameras, INS, and LIDAR systems in synchronized video and ROS bag formats. In performance analysis, two particular outdoor data collections from the forest nature trail, 6_normal and 7_fast, were used extensively. The 6_normal test was a simple walk through the forest at a normal pace, while the fast test was the same path but at a faster pace. The available test data was useful in highlighting the shortcomings of the current system.

TABLE 4

Available Data Sets Prior to MQP

Environment	Weather	Collection Methods
Indoor Lab Space	N/A	Walking and running, rapid head motions, inspection of objects
Wooded Nature Trail	Sunny, no wind	1/2 mile on trail, slow walking, colored objects placed along trail
Wooded Nature Trail	Partly cloudy, windy	1/4 mile varied walking in woods, colored objects with AR tags

5.7 CANUSEA COMPETITION AT MIRROR LAKE

An underwater UUV competition called CANUSEA occurred at Mirror Lake in Devens, MA, where UUV's attempted to locate sunken objects in the lake. The lakebed was scanned as a high-resolution 3D model, and an octomap was used to store the raw sensor information. An octomap is a way of representing point clouds in a tree structure that allows scalable compression to be applied to the collected data. This is necessary when the octomap is to be transfered over networks or rendered quickly on less powerful systems. A research goal for the competition was to load the lakebed map onto the ODG Glasses to allow a user to examine the bottom of the lake, effectively rendering the water see-through. In addition, it was desired that the UUV's intended path, goal, and any identified objects would also be displayed as markers through the HUD glasses. A preexisting Android application provided protocols for receiving these ROS marker arrays and rendering the objects as they would be perceived by the user, whose pose was also passed as a Pose Stamped message to the application. This scenario promised to simply illustrate the practicality of AR HUDs and discover implementation issues with the proposed system.

6. METHODOLOGY

To develop an accurate estimate of position and orientation, a high-level system design was first created. The next step was to evaluate the available sensors and algorithms to determine how well the system could function using either purely inertial dead reckoning or through unmodified ORB SLAM. This part of the process involved recording sensor and truth data in a room equipped with a motion capture system. Sensor data was then compared to truth data to develop an understanding of sensor and software behavior. The following step, the main focus of the project, was to investigate the ORB SLAM code to correct the scale and scale drift by injecting other sensor estimates. Finally, the improved system was reevaluated by comparing the results to the original configuration, and a run time performance evaluation was conducted on ORB SLAM to determine processing bottlenecks for a real-time implementation.

6.1 SYSTEM DESIGN

Figure 10 illustrates the desired design of the system. The available sensors, in gray boxes, are the camera and the INS. The camera provided frames to a Frame Filter node (not implemented in this project), which processed frames to eliminate or fix bad frames that can result from events such as sun glare and cause ORB SLAM to lose tracking. The frames were then passed into the ORB SLAM algorithm, triggering tracking processes. The INS provided the inertial information necessary for a dead-reckoned position estimate. The internal Extended Kalman Filter within the INS also provided an absolute orientation estimate based on magnetometer, accelerometer, and gyro which was utilized by the gravity and DC bias compensation node. The accelerometer provided three-axis raw acceleration values to the compensation node, which removed acceleration due to gravity and internal DC biases. The compensated acceleration was passed to the double integration node.

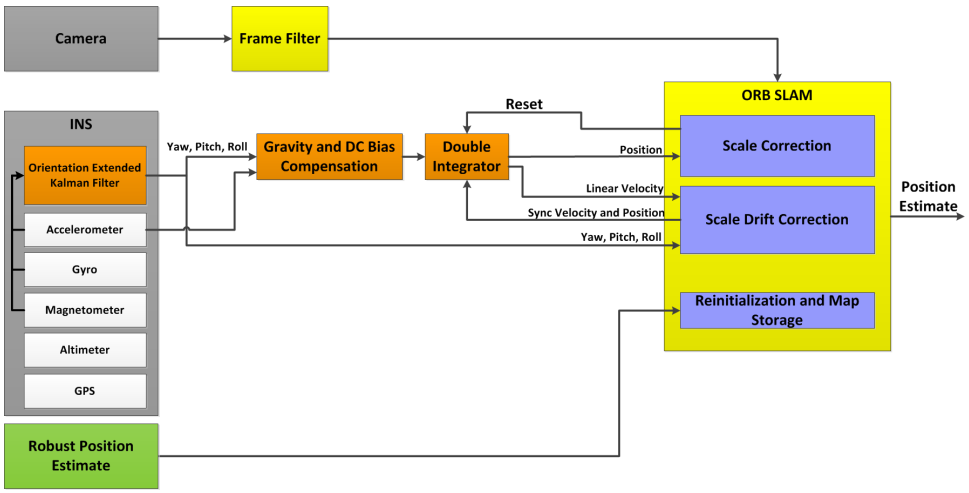

Figure 10. Proposed System Design for Visual-Inertial Pose Estimation

The double integrator integrated acceleration twice to obtain a change in velocity and position due to acceleration. This linear position estimate was used to establish an initial scale in the ORB SLAM code while the linear velocity estimate corrected scale drift. ORB SLAM sent two types of reset signals to the double integrator. The first was sent during scale correction to reset position and velocity to zero, allowing an inertial estimate of position to be established during a particular period of time. This reset occurred on the first key frame to establish scale to the second keyframe. The second reset command was sent by the scale drift correction to reset position to zero and synchronize the velocity in the integrator with the current velocity estimate in ORB SLAM. This provided potential to reduce the effect of integration drift while still providing an inertial estimate.

6.2 DATA COLLECTION AND SYSTEM SETUP

To evaluate the system performance, extensive test cases were developed and captured using the helmet mapping system. These tests required several components working together to provide sufficient data collection to determine system performance and failure cases.

6.2.1 Development of Test Cases

There were four different rounds of tests conducted. The first involved generalized motions around the motion capture environment while the helmet was on a cart and a person. A secondary round of testing was implemented to provide simplified movement for establishing correct coordinate frame transformations. Thirdly, several tests were conducted to more closely examine system performance at varying rotational rates. Finally, some additional outdoor testing was conducted to examine system behavior in a novel environment. Each of these test rounds are discussed below.

Several tests were conducted in order to fully characterize each sensor and algorithm in a number of different situations. Tests were conducted with the helmet mounted on a dummy, which was firmly attached to a rolling cart, and again with the helmet worn by a person. The tests on the cart provided position changes in x and y and rotation about z (yaw). These planar movements simplified analysis and allowed a direct comparison to be made between the LIDAR-based hector SLAM solution and the "truth" provided by the motion capture system. This direct comparison was important as it would provide the proof that LIDAR could be considered sufficient truth in outdoor environments. The motion capture system also recorded truth data in all six DOF, which was not available in outdoor environments. Tests were conducted at a subjectively defined "slow" pace and again at a "fast" pace. Since truth data was available, it was more time effective to characterize the data produced rather than conduct controlled motion tests of the system. However, such approaches may have yielded better test cases and results.

The following motion patterns were tested on the rolling cart and person, at slow and fast speeds, as shown in Table 5. Looping patterns were repeated clockwise and counter clockwise, as cameras could pass very close to the walls in some cases. Additionally, since monocular SLAM needed translation for initialization, rotationally-intensive tests were preceded with brief linear motion. A description of each test case is shown in the column to the right.

TABLE 5

Motion Capture Tests and Descriptions. Repeated at two speeds, both on a cart and a human.

Motion Pattern	**Description**
Static	10 minutes of the system remaining motionless.
Linear	5 translations forward and backward of ∼12 feet.
Rotation	After a brief translation to initialize, system was rotated in place for 5 revolutions.
CW Square	A square of about 14ft was traced 5 times, with a Clockwise motion about the room and the camera facing inwards.
CCW Square	Same as above, but counter-clockwise and the camera facing outwards.
CW Circle	A circle of a 14ft diameter was traced 5 times, with the camera facing inwards. Walking tests were CW while cart tests were actually CCW.
CCW Circle	Same as above, but counter-clockwise and the camera facing outwards.
Erratic	Fast, rapid motions and changes in direction in random movements about the room. When system was worn, also involved large orientation changes.
Head Rotations	Only when worn, helmet was rotated as fast as a person could move their neck.
Jumping	Only when worn, subject crouched and jumped vertically several times.

Further tests were conducted to better understand system coordinate frames and achieve more systematic tests for rotational tracking failure. The coordinate system tests were divided in three parts, meant to isolate issues and create unit test cases. **1)** Translations in positive and negative x, y, z directions. **2)** $\pm 90^\circ$ rotations about x, y, z, axis. **3)** $\pm 360^\circ$ rotations about x, y, z, axis.

Tests for obtaining rotation rates in the lab required a bit more setup to produce acceptable data. The dummy was placed in a rolling chair and tied down with the helmet mapping system secured to the dummy's head. Approximately ten separate tests were conducted where the rolling chair was spun, after an initialization movement, at incrementally faster rates. Thus the first test had the longest time for a full rotation and the last test had the shortest duration.

Brief outdoor tests were conducted at Mirror Lake, which provided new outdoor data where the cameras viewed mostly sparse vegetation and a large moving body of water. The Sun was also low on the horizon and caused some interesting contrast issues. Only LIDAR and GPS data were available for truth determination, and GPS was considered reliable as the tests occurred with an unobstructed view of the sky.

6.2.2 Motion Capture System and LLIVE Preparation

LLIVE is a room with projectors and motion tracking equipment that can display a virtual environment around a user while their position can be tracked and utilized. Within LLIVE, a sophisticated motion capture system employing passive retro reflective marker spheres was utilized to gather indoor truth data. The manufacturer, OptiTrack, supplied the software, Motive, for Windows, which enabled calibration of the 20 IR cameras and tracking of markers placed in the room to mm accuracy. Calibration for the system was conducted according to manufacturer directions. Nine spherical markers were placed on the helmet over a scattered array of locations, and the helmet was placed in the room. Through the Motive software, markers corresponding to the helmet were highlighted and defined as a rigid body, with its frame at the centroid of the detected markers. Position and orientation of the centroid was then tracked by Motive and published through a VRPN onto a lab network. To correlate the centroid rotations to the actual helmet, one marker was placed directly on the IMU center of rotation, which allowed the centroid position to be defined in reference to the helmet sensor suite.

The LLIVE room had to be modified in order to provide textured elements similar to those found in the forest. As previously mentioned, feature-based visual SLAM solutions perform poorly in untextured environments, and the smooth walls in LLIVE were known to be hazardous in previous tests. A program was utilized to display static images of generic forest scenes on the walls using the projectors, which provided unique textures. The lights in LLIVE were left on, because while the lights washed out the projector images, it was the only way to provide enough light in the corners of the room, walls, and ceiling to allow tracking on most of the environment. When lights were off, the projector view was visible, but the room itself was too dark. Conversely, if the fixed ceiling spotlights were used to illuminate the middle of the room, the floor was washed out and the shadows only became worse. Thus having the main lights on provided the best balance. It was ensured that the washed out projector images were sufficient to allow ORB SLAM and LSD SLAM to track adequately in the room.

6.2.3 ROS Preparation

Once the Motive system was correctly publishing data on its socket, a ROS VRPN Client node was run on a Linux machine to capture the data and republish the marker positions in the ROS framework as a TF frame and PoseStamped message. The motion capture system utilized a rotated frame of reference (with the Y axis vertical), so a simple TF was used to correct the rotation of reported mocap position to the reference frame utilized by ROS (from a platform perspective: x axis forward, y to the left, z upright). However, as will be described in the following section, the TF data was not recorded on the Jetson, so the positional data had to be transformed once again at the time of data playback.

Next, the Jetson and secondary Linux computer were connected to the same network, and ROS was configured to allow operation of both machines with the Jetson as ROS master. Since both machines were on an isolated WiFi network, and the Jetson board did not possess a backup clock battery, clock synchronization was necessary. Chrony was used to implement NTP date/time corrections to rectify time difference between the two computers. This was necessary to ensure local machine times were as close as possible, though ROS messages would still be time stamped by the master's time. The time delay from mocap to the VRPN client was not compensated for.

6.2.4 Data Collection

The Jetson data collection program, which was developed over the summer preceding MQP, was employed for capturing the sensor information available on the Jetson. It was modified to record the position of the helmet as reported by mocap through the ROS VRPN client. The TF tree was not recorded by the Jetson, as TF information would be provided during data playback and working with multiple sources of TF trees can become complicated.

With the systems in place, data collection was started through an Android data capture UI client, and the Jetson board began capturing video, IMU data, INS heading, motion capture position, and LIDAR readings. The helmet remained stationary for a few seconds before motions began. After motions were conducted, the helmet was brought to rest for a few seconds before terminating the data collection run. All motion patterns were conducted and saved individually, and test files were then copied off of the internal SSD onto main computers.

To replay the data, bash scripts were developed to parse through the folder structures and select the appropriate bag and video data. User specified arguments were interpreted by the bash script, and the ROS launch file properly started the video playback, transform, robot state publisher, Hector SLAM, ORB SLAM, RVIZ, and rosbag nodes. Based on the selected camera, the proper ORB SLAM calibration file was loaded along with a faster-loading binary version of the vocabulary. An additional script was created to run tests on all data collections present within a particular directory so that a characterization/analysis node could record observations for each of the data collections and record the results in an organized text file.

6.3 TRANSFORMING COORDINATE SYSTEMS

To fully utilize the data captured with the Jetson system, a complete model of the helmet mapping system had to be developed in order to properly transform the data from each sensor into the common helmet frame. This was necessary for two main reasons: **1)** Position estimates from multiple systems needed to align to properly compare errors and **2)** coupling the IMU with the vision algorithms would only be possible with the correct transformation of sensor measurements into the coordinate frame of the camera. A URDF was necessary to properly define the system. A SolidWorks assembly of the helmet mapper system was built to include the most up-to date models of the hardware and sensors used on the platform. Coordinate frames for each sensor were properly added to the parts, and an STL for each part containing the corrected coordinate frame was exported, ensuring SolidWorks did not move the part into positive coordinate space. The translations and rotations between the model's frames were then determined through measuring part frame distances in the model assembly. These measurements were then integrated into a URDF file and the STL files were the reference meshes for the links of the robot. Several iterations were created and examined in RVIZ until the proper configuration was attained, as shown in Figure 11.

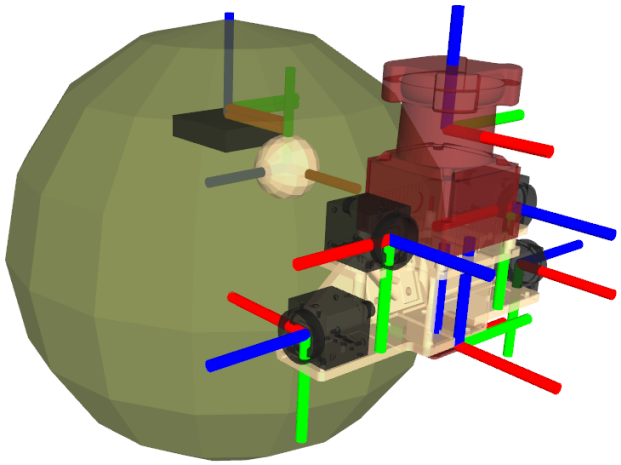

Figure 11. 3D Rendering of the helmet mapper URDF showing frames of various sensors. Red is x, green is y, blue is z. The black rectangle in the center is the GPS receiver, and the tan sphere is the mocap centroid.

With the URDF loaded and a Robot State Publisher creating the proper TF tree from the URDF, defined transformations could easily be used to transform sensor readings, such as acceleration and rotational velocity data, into different frames. This was necessary in order to obtain visual and inertial measurements about the same reference point.

While inter-helmet frame transformations were trivial, significant time was spent achieving accurate transformations into and out of the ORB SLAM system. Initially, it was believed ORB SLAM used a left-handed coordinate system as indicated by a Github issue for the ORB SLAM

code [27] and a pull request that flipped ORB SLAM positions and published the pose as a TF [28]. However, experimentation revealed this to be false. Positions of a given camera frame were reported as the origin's position with respect to the frame, and an OpenCV coordinate frame was utilized, with origin at the top left of the image and z pointing into the image, x to the right, and y pointing down.

To convert from ORB SLAM coordinates into standard ROS coordinates which the helmet base used, the inverse of the ORB position was taken to obtain a position with respect to the origin, then the transform between the physical camera and the helmet base was used to transform the position to accurately represent the helmet base position according to ORB SLAM.

```
(tfCamtoHelmetBase*orbPoseTF.inverse()) * tfCamtoHelmetBase.inverse()
```

Transforming helmet frames into ORB SLAM was slightly more complicated since verification needed to happen inside ORB SLAM without ROS visualization tools. This is where the coordinate system tests were very valuable for verifying transformations. However, it proved to be a near inverse of the previous transformation. An example of the transformation from the mocap reported pose to ORB SLAM coordinates is shown below, and Figure 12 shows the TF tree used to make the transformation.

```
(mocapPoseTF*tfMocapToCameraRF).inverse()*tfMocapToCameraRF)
```

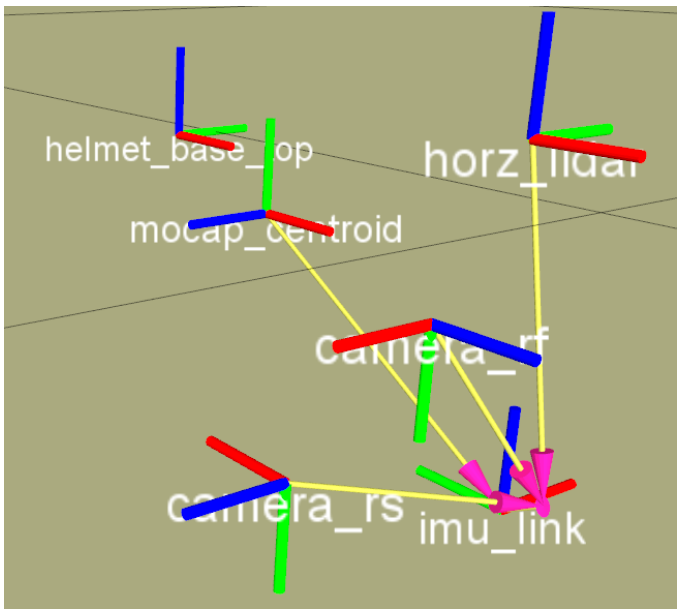

Figure 12. 3D Rendering of the main sensor frames on the helmet. Red is x, green is y, blue is z.

6.4 DEAD RECKONING WITH THE IMU

Achieving an inertial position estimate required integrating the IMU acceleration after gravity and DC biases were removed. Two ROS nodes were developed to process the data directly from the IMU and convert it to an inertial position estimate. The first node removed acceleration due to gravity and DC biases inherent to the accelerometer, and the second node double integrated linear acceleration to obtain a position estimate.

The gravity compensation node performed initial processing on the IMU data. Its function was to remove any undesired biases that were included in the raw accelerometer output. There were two types of biases that the node removed. The first bias was that due to gravity. Because gravity remained fixed relative to the world coordinate frame and rotated relative to the IMU coordinate frame, the compensation node utilized the current orientation of the IMU to compute a unit vector in the direction of gravity relative to the IMU. This node used a saved gravity magnitude and multiplied this with the unit vector to obtain the gravity vector. The gravity vector was then subtracted from all raw acceleration measurements.

The second DC bias was removed next. DC bias included all biases that remained fixed relative to the IMU. When the compensation node started, it performed a calibration phase for the first 100 IMU readings. Gravity was removed from the first 100 readings and the results were averaged for each axis to obtain an estimate for the DC bias. This method for estimating DC bias relied on the IMU remaining relatively still during this calibration period. During normal operation, the compensation node recalculated the estimated gravity bias vector whenever a new reading was received. The node then subtracted both the gravity bias vector and the DC bias vector from the raw values and published the compensated acceleration value.

The position integration node determined position using these compensated acceleration values. Acceleration was integrated once to calculate a change in velocity, and then velocity was integrated to calculate a change in position. The node heavily relied on effective bias compensation because a constant bias resulted in position drift that accumulated quadratically due to the double integration. In the completed system, the velocity integration was provided to the scale drift correction modification in ORB SLAM, and the position integration was provided to the scale correction modification.

6.5 DETERMINING MAXIMUM ROTATION RATES

Characterizing the system tolerance for rotations was important for understanding what limitations the hardware may impose or what cases the system would be unable to recover from. The IMU and ORB SLAM systems were analyzed individually.

6.5.1 Maximum Rotation Rate of IMU

According to the manufacturer, the gyro in the IMU could accurately record rotational speed up to 180° per second. This was verified by examining the angular velocity reported by the IMU during fast head rotation tests and comparing the value to that reported by mocap. A node first had to be written that calculated the angular velocity of the system from the mocap-recorded poses.

Simple filtering was necessary to smooth the velocity as the mocap data suffered from jitter. A ROS plotting program was then used to monitor the angular velocity of both systems over time.

6.5.2 Maximum Rotation Rate of ORB SLAM

There were two variables at play in the tracking ability of ORB SLAM: motion blur and frame skipping. Since ORB SLAM skipped frames when a tracking cycle took longer than the frame capture period, there was potentially lost information during the rotation. The motion blur had to be ruled out as a factor. The data recorded from the rapid head rotation was played back at a rate below the tracking frequency of ORB SLAM so that it had time to process each frame fully. The tracking status was examined as the data was played back. This would demonstrate if tracking failed solely due to camera motion blur, as every frame was processed and tracking would only fail if very blurred frames prohibited tracking.

The other case for ORB SLAM failure was that it did not have enough information about rotations due to dropped frames. In other words, there could be a lack of sufficient frame overlap between two sequentially processed frames. Unfortunately, a lack of time prohibited an implementation of this test. A plan was created to properly isolate tracking loss cases by chopping out some frames during playback of a slow rotation, which would have simulated having no motion blur but a fast rotation with reduced frame overlap. The frame overlap percent would have been determined by feeding frames into ORB SLAM at particular angle checkpoints, as recorded by mocap and through utilizing the camera field of view. In essence, this would simulate static pictures taken at set intervals around the rotation. This would require code modifications to allow all frames to enter ORB SLAM for initialization until the rotation sequence of the test started. ORB SLAM's tracking quality as a function of sequential frame overlap percent could have been determined by applying the test at varying frame release angles to achieve a wide enough spread to see relationships between angular rate and number of tracked features.

6.6 INVESTIGATING THE ORB SLAM SOURCE CODE

To determine which parts of the ORB SLAM algorithm to modify, the overall design of the algorithm was considered. Below is a list of relevant C++ classes in the ORB SLAM algorithm implementation. The understanding of each file's function was developed as part of the project since there was no documentation for each file as a whole.

- Frame - Stored a single frame, which represented a two dimensional camera image. Each time a new image was received from the camera, a Frame was created. The Frame contained estimates for the translation and rotation of the frame on the map and a KeyFrame that the Frame was referenced to.
- KeyFrame - A new KeyFrame object was created each time the current Frame's set of features differed enough from the previous KeyFrame to be useful for feature tracking and efficient bundle adjustment and loop closure. A KeyFrame object stored a reference to its corresponding Frame, a bag of words description for loop recognition, any child KeyFrames, and information

to represent a KeyFrame graph such as edge weights. The graph enabled the algorithm to reoptimize the pose of each KeyFrame on loop closure and local bundle adjustment.

- Tracking - The Tracking class maintained a thread that performed processing on each frame as it was received from the camera. It positioned Frames and constructed KeyFrames according to feature tracking and a constant velocity motion model. [8] Tracking was able to initialize the algorithm using monocular, stereo, or RGB-Depth sensors such as the Microsoft Kinect.

The above understanding of these essential components of the algorithm was achieved in part through examining the Frame.cc, KeyFrame.cc, and Tracking.cc files of the source code, available on GitHub. [29] [8] In the Tracking.cc file, a function was used for the monocular implementation that initialized the map (CreateInitialMapMonocular), which stored feature map points and KeyFrames. One of the relevant steps of the code (Tracking.cc, lines 688-702) established an initial baseline distance between the first two KeyFrames. The algorithm computed a median depth of feature points collected by these two KeyFrames to produce a crude scale estimate (Tracking.cc, line 689). The algorithm computes this scale estimate and scales the position of the current (second) KeyFrame and the associated map points. Camera poses were represented as the transformation of the current KeyFrame to the initial KeyFrame. The first KeyFrame pair was then stored in a LocalMapping object (Tracking.cc, lines 715-716). Further new frames were positioned with reference to the last KeyFrame, and that KeyFrame was listed as a reference frame. An estimated position of the camera was updated from the transformation of the current frame to the last KeyFrame (Tracking.cc, line 732).

The next step of code examination was to determine how the current Frame's position was updated in the code since the scale drift correction objective could be achieved by repeatedly correcting the position of the current Frame according to external estimates. [29] [8] The function that updated the position of the current Frame was called TrackWithMotionModel in Tracking.cc, which began by updating the last (previous) Frame (Tracking.cc, line 873) through the UpdateLastFrame function. After retrieving the position of the last Frame relative to the most recent KeyFrame in the form of a four by four translation/rotation matrix, the function multiplied the position by the KeyFrame position matrix to obtain the absolute pose of the last Frame (Tracking.cc, lines 803-807). Next, back in TrackWithMotionModel, the position of the current Frame was set equal to an estimate of the current velocity (a four by four linear and angular velocity matrix) multiplied by the translation/rotation matrix of the last Frame relative to the most recently added KeyFrame (Tracking.cc, line 875). This matrix transformation tended to carry the velocity between frames, similar to an object having momentum. The author's paper confirmed that ORB SLAM utilized a constant velocity model. [8] After positioning the current Frame, the SearchByProjection function was called to find common features between the current and previous Frame (Tracking.cc, lines 885 and 891).

With the velocity estimate understood, the next question to answer was how the velocity estimate was being set. The answer provided a means to merge the inertial data into the velocity estimate. The TrackWithMotionModel function was called within a larger tracking function called Track. One of the relevant operations in the Track function was updating the velocity estimate in the motion model (Tracking.cc, lines 423-432). If tracking was being maintained, on each cycle of Track

the estimated velocity was updated with the product of the current frame's translation/rotation matrix and another matrix consisting of the inverse rotation and camera center position of the last frame (Tracking.cc, line 429). This operation computed a difference between two relative poses to obtain velocity. Connecting this with the preceding discovery, this velocity acted as an estimate for the velocity between the next pair of frames. At this point in the code investigation, when the velocity estimate update was discovered, the next step was to determine a means to deliver the inertial data to both this point in the algorithm and the point where the baseline between the first two KeyFrames was established.

6.7 SLAM SCALE CORRECTION

To correct the ORB SLAM scale, a two step process was utilized. First, truth data from mocap was used to seed the initial baseline estimate to obtain a metrically accurate SLAM scale, and second, modifications were made to replace the mocap position estimates with inertial ones.

6.7.1 Utilizing Truth Data for Scale

Understanding that the map scale was set during monocular initialization in the tracking class of the SLAM algorithm, effort was focused on injecting the scale estimate at this location. Injecting the mocap position required that the externally-estimated position was available on any frame to allow the baseline distance to be computed at initialization. Thus, the grabFrame function was modified to also save a passed position into each frame, and the ROS interface was modified to record mocap positions, transform them, and pass them into the grabFrame function. This approach assumed the positional data lacked any drift or error in it; additional work was needed to adjust the system to use inertial position estimates since they drifted after several seconds.

Frame and KeyFrame classes and constructors were modified to save and contain a four by four OpenCV matrix representing the rotation and origin of a transformation. The ROS interface gained a subscriber to record the incoming position data from the motion capture system. Initially, only the position was passed through the system, as the orientation did not change scale estimates. The position data was passed into grabFrame, and the tracking algorithm then saved a new frame. Once the algorithm selected the first keyframe pair, the position data in the frames were retained to compute the initial baseline.

The original developers of ORB SLAM utilized an inverse median depth value to scale the initial baseline estimate. This provided some change in scale based on the perception of depth in the world. In other words, without any external information available, the best pure vision system can simply calculate how far stereo-matched points are from the camera, and assume that if points are very far from the camera, detecting such points required a larger baseline change than if points were very close to the camera and only required a small baseline change. In this way, the scene did have an impact on the initial scale estimate. The position of the second keyframe was multiplied by this inverse median depth, and the three dimensional stereo-matched map points were also scaled by this amount.

Since the positional estimate in the new system was a much better metric for the actual distance the camera translated, the inverse median depth value was no longer utilized for the baseline estimate unless no external pose estimate was available. A straight-line distance was calculated between the two key frame positions supplied by mocap. The original translation of the second camera had to be normalized and then multiplied by the new baseline. Matched map points also needed to be scaled by this baseline difference, and the scaling factor between the ORB baseline and the truth-baseline supplied this scaling factor. As the map continued to be built, the original scale estimate was utilized in adding new keyframes and computing the distances in map points.

6.7.2 Utilizing Inertial Data for Scale

Integrating the inertial data for a scale estimate required additional development since the double integration from the IMU drifted over time. The approach was to reinitialize the integration when an initial keyframe was selected, producing an IMU-derived translation from the first keyframe to the second keyframe. However, this assumed the estimate would not drift significantly in the time it took to attain the second keyframe. If the helmet did not move for an extended period of time, SLAM initialization would not complete quickly, IMU-derived position drift error would accumulate, and the scaling would not be accurate.

To solve this problem, a flag was set in the tracking class when the first keyframe was selected, indicating that the inertial position integration should be reset. Once this reset occurred, the next keyframe's position was checked and the baseline was computed as it had been for the truth-provided scale correction.

Further changes were necessary but not fully implemented to reduce error associated with the dead-reckoning drift. To ignore time delay problems between a reset request and the position being reported as zero, the initial baseline should have been assumed zero when the second position estimate was checked. To solve the problem where drift would accumulate during slow initializations, the velocity of the camera should have been used to reset the integration node when the motion was near zero velocity. In addition, if the camera and IMU estimate were both translating before the camera stopped, the dead-reckoned position could be stored and added upon once translation continued. Alternatively, initialization could be withheld until translational motion had been achieved, yet this would delay useful results from ORB SLAM.

6.8 SLAM SCALE DRIFT CORRECTION

The goal of scale drift correction was to ensure that SLAM maintained an accurate scale estimate throughout its operation. The technique that was used to correct scale drift was to utilize inertial data from the IMU. Using an inertial estimate for current rotational and linear velocity, the current scale estimate could be corrected. The first step in correcting scale drift was to determine the specific modifications to the source code of the SLAM algorithm that were necessary to accept and make use of the additional information that the IMU provided. This was discussed in subsection 6.6. The next step was to apply the modifications to algorithm and deliver the truth and then inertial data.

6.8.1 Scale Drift Correction Using Truth Data

In order to determine if the ORB SLAM code was successfully modified to incorporate a functional scale drift correction, truth data generated from the mocap system was utilized. There were two main ways that scale drift correction could fail. One was that the inertial data provided may not resemble truth well enough to produce good results. The other way was that the approach for modifying the ORB SLAM code was inadequate. The ORB SLAM algorithm contained many intermediate steps in its estimation of velocity in the motion model. Selecting the wrong intermediate step to intercept with the correction could result in inconsistencies among various propagations of the estimate. Truth data isolated the possibility of issues resulting from the chosen ORB SLAM modification from the quality of the correction data. After being transformed into the right coordinate frame, the truth data was sent to the scale drift correction modification in ORB SLAM.

6.8.2 Scale Drift Correction Using Inertial Data

As shown in Figure 10, the system implemented scale drift correction using a two way correspondence between part of the ORB SLAM implementation and the integration node. The double integrator node provided inertially estimated X,Y,Z velocity to the scale drift correction in SLAM. The inertial velocity estimate was used to periodically make a correction to the velocity that was normally estimated in the motion model of the SLAM algorithm. After the correction to velocity was applied, the scale drift correction sent a message containing the velocity estimate to the double integrator to synchronize the integrator's own velocity with the corrected velocity estimate in ORB SLAM. This isolated integration drift that arose directly from within the integration node to one correction cycle and prevented previous drift from interrupting future correction cycles. However, this technique did not entirely isolate integration drift between cycles because part of the drift was inevitably stored in ORB SLAM and carried back to the integrator as a result of the correction itself. The extent to which integration drift was carried between cycles depended on the nature of the correction.

The type of correction applied was a simple replacement of the original motion model velocity with the externally established velocity. This type of correction was simple but maximized the integration drift carried from one correction cycle to the next. The velocity sync message simply sent the correction back to the integrator and had no effect in this case. However, it would be advantageous in conjunction with correction methods that find a compromise between the inertial information and the original SLAM estimate. A compromise between the two estimates could be able to counteract drift that accumulated in the integrator before each velocity sync message was sent to remove some of the drift.

6.9 ORB SLAM PERFORMANCE EVALUATION

To prepare for implementing the position estimation system to run in real time (supporting a frame rate fast enough to avoid loss of tracking during quick movements) on the Jetson TX1 board, the runtime performance of the ORB SLAM algorithm's implementation was analyzed. The implementation of ORB SLAM consisted of four threads:

- Tracking - Each time an image was received, this thread processed the image, created new Frames and KeyFrames, adjusted the position of the current frame, and compared features between consecutive KeyFrames.
- Local Mapping - Running in parallel with Tracking, this thread, for each KeyFrame produced in Tracking, performed bag of words classification on the frame, used triangulation to position new map feature points, found feature point correspondence with neighboring KeyFrames, and executed local bundle adjustment on recently placed KeyFrames. Local bundle adjustment utilized a graph containing constraints between KeyFrames to locally optimize the placement of the KeyFrames.
- Loop Closing - This thread ran in the background and detected loops in the path traveled. Using bag of words comparison, the thread checked for KeyFrames that resembled previously added KeyFrames. When a KeyFrame similar to a previous one was found, the KeyFrames in between were considered to be the path of a loop, and the thread performed a global bundle adjustment. This adjustments re-optimized the position of each KeyFrame in the loop with new constraints added by connected KeyFrames.
- Viewer - This thread ran concurrently with the other threads and displayed a visualization of the ORB SLAM algorithm. The visualization displayed the camera at its current location, the estimated path in space that ORB SLAM produced, the placement of KeyFrames, and the graph constraints between KeyFrames.

Out of the four threads, the Tracking and Local Mapping threads were analyzed in depth, and the Loop Closing and Viewer threads were analyzed only from a high level. The reason for analyzing Tracking and Local Mapping threads in depth was that they perform computationally intensive operations most frequently. Tracking performs operations including feature identification and tracking on the receipt of each image, and Local Mapping performed analysis and optimization of the placement of each KeyFrame. Therefore, improving performance of these threads would be most essential for increasing the performance of the overall algorithm.

The analysis began with identifying sub tasks that appeared to be computationally intensive and then producing a function call hierarchy of these tasks for each thread in the in-depth analysis. Thread hierarchies, shown in Figure 28 and Figure 29, capture the task break-down of Tracking and Local Mapping threads and act as a framework for understanding the relationships between runtime results.

Functions or subtasks considered to be insignificant in terms of run time were initially left out of the analysis. As results uncovering significant run times of omitted collections of tasks were produced, some of the additional tasks within the omitted sections were explicitly identified and added to the analysis. Determining run time left out of the analysis within a particular parent task involved subtracting the total runtime of child tasks from their parent task. The result either demonstrated that the runtime unaccounted for was insignificant or that it was significant and new tasks would need to be added to the analysis. The setup for the data recording involved placing statements into the code that recorded the time of the start and end of each task.

To set up the code for time recording, the first step was to create wrapper functions for library calls to obtain time from a timer. The Boost timing library [30] was used to provide the timer to measure wall clock time during execution. Two recording wrapper functions were implemented for each thread. A pair of functions recorded start and end times for an event in each thread. The purpose of the wrappers was to be able to easily change the type of time being recorded. Analysis was implemented to use wall clock time, but this could be easily changed to CPU time for each thread. Next, for Tracking and Local Mapping, the time recording calls were placed at the start and end of each task identified in the call hierarchies. For the other two threads, the record statements were only placed in the main loop of the each thread. The time recording functions outputted the data to four CSV files, one for each thread.

After running the code with the time recording functionality, two CSV files were produced. In each file, there was a large list of data entries. Each data value consisted of a sequence number (a number n for the nth recorded event timestamp), a cycle number identifying the cycle that was currently executing in the thread, an event label that corresponded to an event in the hierarchy, a status label to mark the "start" or "end" of the event, and a timestamp. The data was processed in MATLAB. The processing involved finding the time difference between the start and end of each occurrence of every event and building a call hierarchy tree structure for each thread. The structures were then converted to tables that listed the analyzed subtasks and the cycle duration values for each subtask to allow further statistical analysis.

6.10 AR HUD DEMO AT MIRROR LAKE

To demonstrate the utility of augmented reality in visualizing robot perception and better understanding one's environment, a brief AR demo utilizing the ODG HUD glasses was partially implemented. An octomap of the lakebed at Mirror Lake in Devens, Massachusetts was supplied as a .bt file, and it was loaded using an octomap server running on the Jetson, which published the octomap as a marker array ROS data type, shown in Figure 13. A wireless access point was created on the Jetson so that Android devices and the glasses could communicate with the Jetson ROS master. The Jetson utilized the IMU heading on the helmet to establish the orientation of the user wearing the helmet and glasses. The Hector SLAM-provided position of the user was initially used to allow the user's translations to impact the view of the map. However, head rotations were not corrected for in Hector SLAM, so a static position was assumed along the lake shore. A node was modified to allow the RVIZ goal selector to relocate the user position within the world.

An Android app, which ran on the HUD glasses and Android Tablets, was configured. A TF tree that rotated the octomap into the correct coordinate frame and placed the user frame in a suitable location on the lake shore was established. Because the Android app could only display about two thousand markers, the ROS utility RQT reconfigure was used to reduce the octomap's depth, effectively reducing the resolution and number of markers. Secondarily, it forced the octomap marker array to republish, which sometimes did not automatically start publishing to connected subscribers. The Android app then subscribed to the octomap marker array and the position derived from the IMU. On the day of competition, the equipment was transported to Mirror Lake and the system was setup as it had been before.

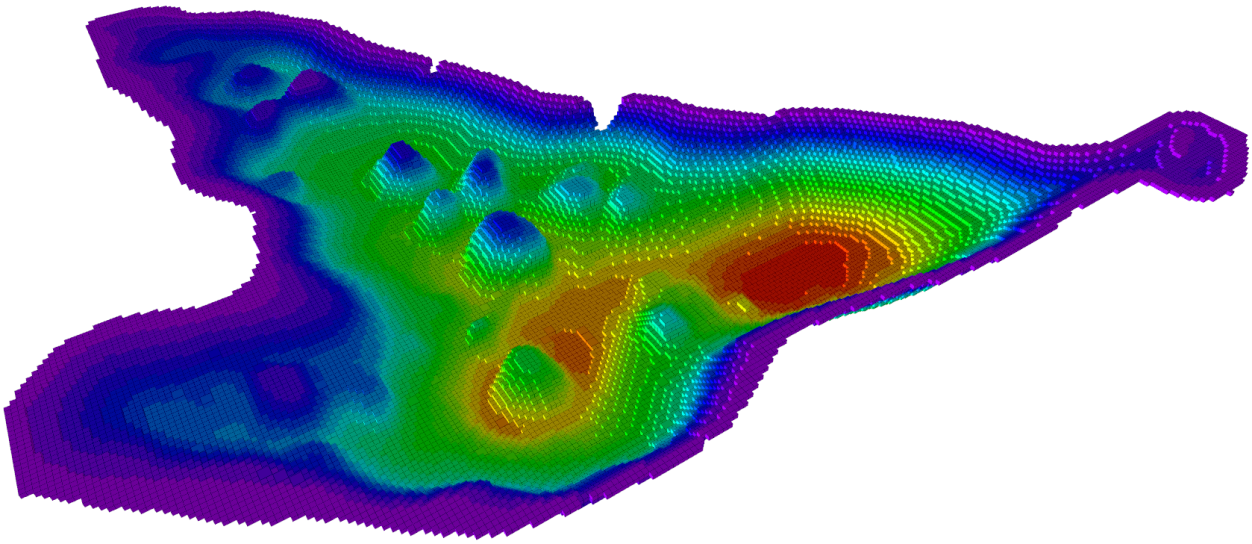

Figure 13. Octomap of the 3D lakebed of Mirror Lake.

7. RESULTS AND ANALYSIS

7.1 DATA COLLECTION AND SYSTEM SETUP

Approximately 185 GB of data was collected from indoor tests utilizing motion capture. Table 6 displays the characteristics of tests conducted in the motion capture studio using the cart, and Table 7 displays test characteristics when a person was wearing the helmet.

TABLE 6

Motion Capture Test Data Characteristics: Cart Tests, Filename Prefix indoor_sept.

Motion Pattern	File Name	Distance Traveled (m)	Angular Distance Traveled (rad)	Max Linear Speed (m/s)	Avg. Linear Speed (m/s)	Max Angular Speed (rad/s)	Avg. Angular Speed (rad/s)
Static	static_10Min_2016-09-02-15-09-21-461	0.00633325	0	0.0201546	0.0016986	0.226606	0.0250932
Linear	linear_2016-09-02-15-33-38-715	42.1126	3.1785	4.65061	0.488986	0.727079	0.0649068
Rotation	init_rotation_2016-09-02-15-46-14-518	1.99394	2.20568	0.226317	0.0404396	2.96193	0.406501
CW Square	square_inside_2016-09-02-16-05-52-683	47.7179	50.1035	0.927823	0.330653	3.06238	0.202378
CCW Square	square_outside_2016-09-02-16-11-21-263	45.8427	35.71	0.899541	0.359645	1.36476	0.215678
CW Circle	circle_outside_2016-09-02-15-57-16-017/	72.1429	32.2752	9.15281	0.789296	4.14378	0.367158
CCW Circle	circle_inside_2016-09-02-15-51-57-828	61.0832	19.8713	20.6729	0.614096	10.6498	0.343363
Erratic	crazy_2016-09-02-16-17-10-667	55.2944	31.3353	1.58391	0.706952	2.09616	0.721339

Generally, the collected test data demonstrated sufficient coverage of potential system failure cases and provided some insight into the effectiveness of different position sensing systems. Examining differences between the cart tests and the walking tests, it was found that the walking tests typically had higher average rotational speeds and marginally faster average linear speeds. This indicated, as expected, that a human factor added extra movement to the tests. However, a closer examination of cart test data revealed an appreciable amount of jerk in the z axis, making those tests sometimes less smooth than walking tests. This was likely due to the cart striking the tile dividers on the floor. However, the effect played a little role in results and was overshadowed by some other issues.

TABLE 7

Motion Capture Test Data Characteristics: Walking Tests, Filename Prefix indoor_walking_sept.

Motion Pattern	File Name	Distance Traveled (m)	Angular Distance Traveled (rad)	Max Linear Speed (m/s)	Avg. Linear Speed (m/s)	Max Angular Speed (rad/s)	Avg. Angular Speed (rad/s)
Linear	linear_2016-09-02-16-26-20-474	23.1396	1.59972	5.30057	0.765232	0.577534	0.154217
Rotation	rotation_with_init_2016-09-02-16-28-00-401	5.83651	19.522	0.656048	0.173289	2.46237	0.907004
CW Square	square_inside_2016-09-02-16-05-52-683	48.6679	32.4979	1.56389	0.688767	2.71533	0.42802
CCW Square	square_outside_2016-09-02-16-34-45-926	48.6572	32.4907	1.5615	0.69202	2.54788	NA
CW Circle	circle_inside_2016-09-02-16-31-02-786	39.6156	11.9623	1.64826	0.899645	1.26868	0.469808
CCW Circle	circle_outside_2016-09-02-16-32-50-907	52.6086	14.3347	1.63899	0.936594	1.2675	0.494299
Erratic	crazy_motions_2016-09-02-16-39-22-782	49.5419	64.182	2.66819	1.16816	9.84431	1.86442
Head Turns	head_turns_2016-09-02-16-29-58-499	8.06952	30.3352	1.40889	0.258899	11.8876	1.24404
Jumping	jumping_2016-09-02-16-40-13-515	9.10063	3.03893	2.11614	0.342766	1.70767	NA

Motion capture proved to be a reliable truth estimate of position, and the placement of the retro reflective markers enabled the helmet to be rotated entirely upside down without loss of tracking. Yet sometimes the jitter in the reported pose indicated that further filtering needed to be applied to reduce instantaneous changes in position. This was most strongly manifested in the output of the mocap velocity data as a Twist message, composed of linear velocity and angular velocity. This jitter resulted in large velocity jumps which skewed the reported maximum speeds in Table 6 and Table 7. Application of a simple software low-pass filter or moving average filter could eliminate this problem and allow more accurate maximum speeds to be attained.

The Hector SLAM and ORB SLAM solutions were compared to the mocap data collected during the trials, with results displayed in Figure 14. Hector SLAM was fairly reliable as a truth system when rotation remained slow and limited to planar motion. Some slight error was noticeable between mocap and Hector SLAM, as the mocap data appeared to be slightly rotated with respect to the Hector SLAM path. This may have been caused by the tilt in the mocap ground plane with respect to the IR mocap cameras in the room. This non-zero error could also indicate that the mocap system was not properly calibrated, in which case a more standardized movement test would be needed to confirm these results.

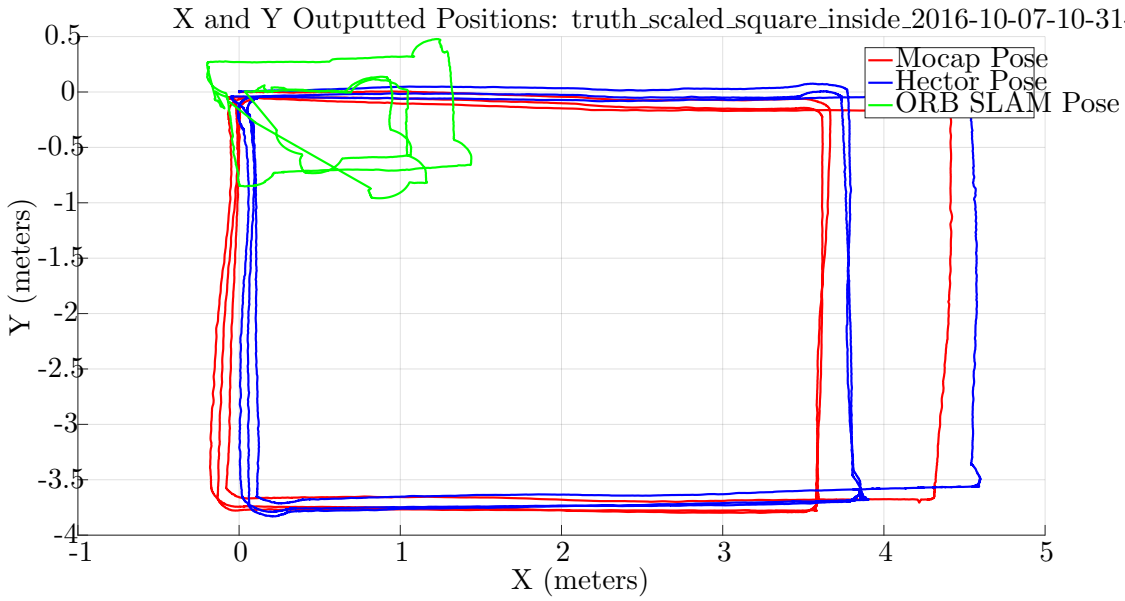

Figure 14. Square motion test on the cart showing plots of mocap, Hector SLAM, and ORB SLAM.

In addition, Hector SLAM sometimes lost accurate positioning during walking trials. This demonstrated that truth from Hector SLAM may become invalid on tests outdoors that included quick rotations or non-yaw rotation common during careless walking. Additionally, in outdoor tests, it was discovered that Hector SLAM failed near hills if the LIDAR's spinning axis was not perfectly orthogonal to the ground plane. In this instance, Hector SLAM believed it was following the curve of a wall since the scans swept the slope of the hill as motion progressed forward. This

caused the position solution to rotate off the true path and spiral towards the hill. This case should be studied further, but it may be necessary to add IMU orientation to correct the Hector SLAM solution.

In walking tests, it was also discovered that inertial information could be used to provide position estimates through pedometry, a method where position estimates are obtained through counting steps and estimating stride length. Outdoor tests revealed that on packable terrain such as leaves and with different walking behaviors, such patterns were more difficult to distinguish. In all, this did not rule out the possibility of falling back to pedometry estimates if visual odometry was unavailable for long periods of time, and it should be considered in further system research.

7.2 TRANSFORMING COORDINATE SYSTEMS

The different coordinate frames of the system were successfully determined and related through the ROS TF library. Figure 15 shows the full TF tree that was implemented, but several of the system maps were statically defined and not based on initial position changes between collection runs. Ideally, the maps would be defined based on common reference points in the real world, such as GPS, and this would allow the map positions to be moved more easily.

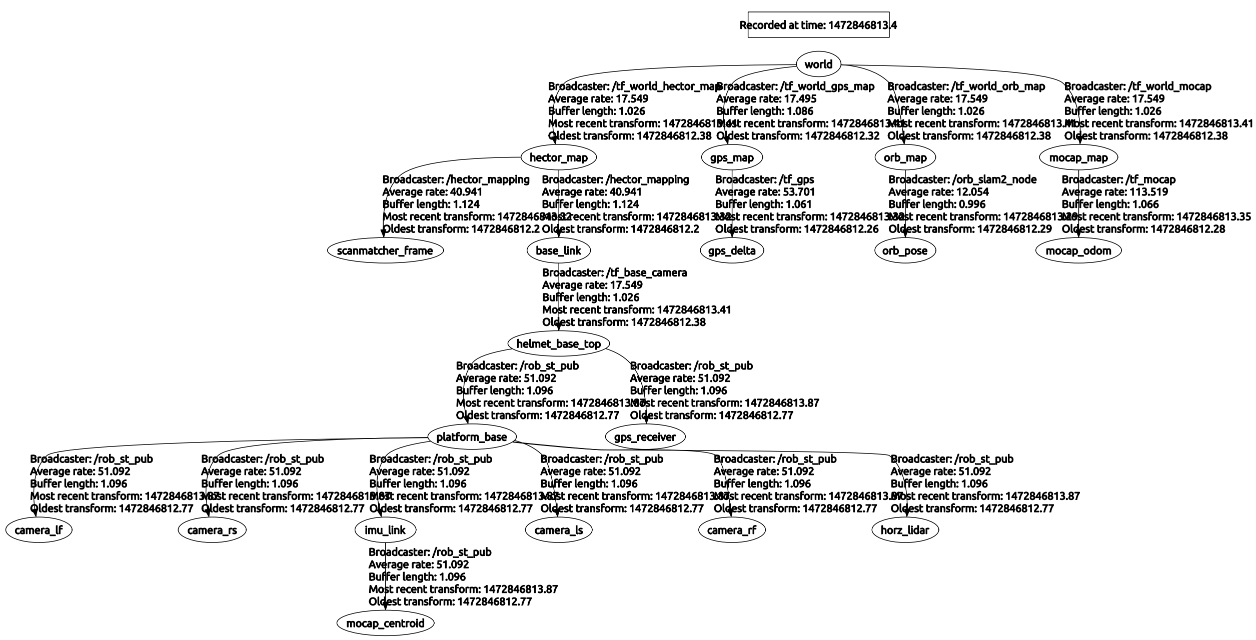

Figure 15. TF tree of helmet mapping system.

The coordinate frame tests were used to check accuracy of the transformations that were implemented. Figure 16 shows the results of running the position algorithms and data with the TF tree configured.

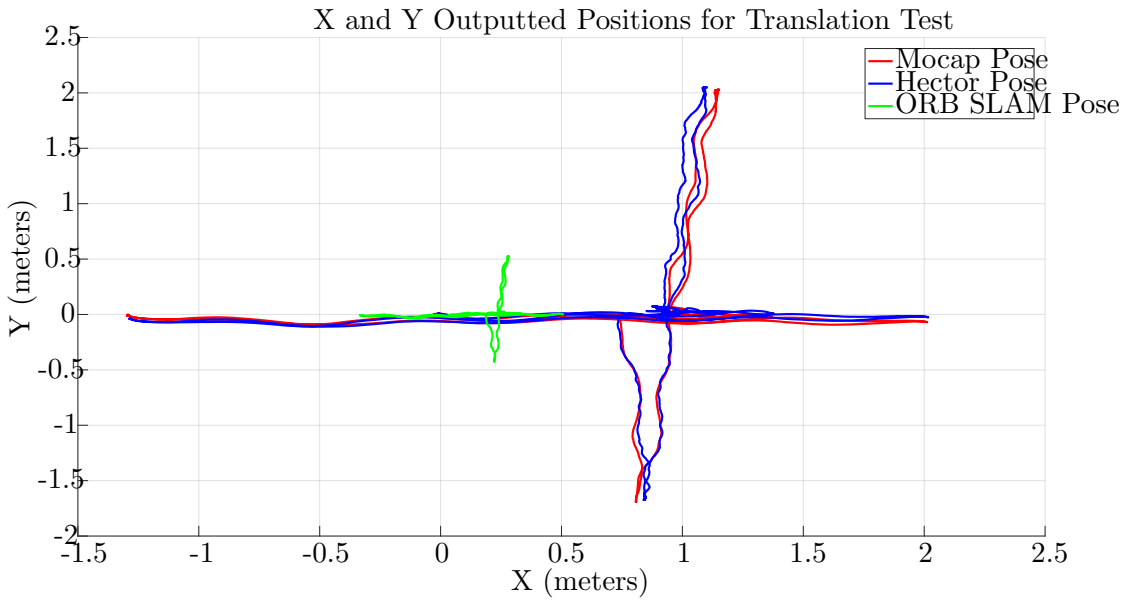

Figure 16. X,Y positions of truth systems and ORB SLAM without scale correction applied.

Disregarding tracking errors, each of the systems report similar translational and orientation behavior. This indicated correct implementation, though setting up the system and understanding the necessary transformations was more time consuming than expected.

7.3 DEAD RECKONING WITH THE IMU

Dead reckoning using the IMU was implemented to both correct scale and scale drift in the SLAM algorithm and to provide a purely inertial estimate of position when the SLAM estimate was unavailable. Refer to subsection 6.4 for implementation details. Results were generated from the rotating helmet and linear movement trials.

The graph in Figure 17 shows the raw accelerometer data before gravity and DC bias compensation for the rotating helmet trial. In this trial, the helmet was rotated 360° around each axis. The graph displays the varying effect that gravity had on the three axes as the helmet was rotating but not linearly accelerating.

The graph in Figure 18 shows the gravity and DC compensated acceleration data for the rotating helmet trial. This indicates gravity compensation was able to calculate the gravity vector relative to the IMU and subtract it from each of the three axes, even while the orientation was changing. After the gravity vector and other DC biases were subtracted, the acceleration plot for each axis was centered at approximately zero.

The graphs in Figure 19 and Figure 20 show the integration of compensated accelerometer data plotted against mocap (truth) for the rotating helmet and linear movement trials respectively.

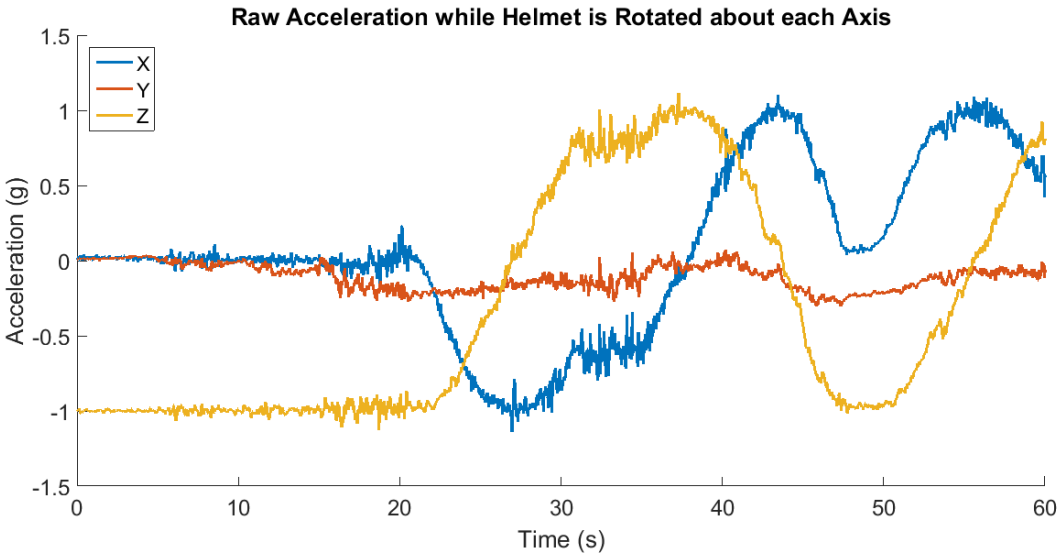

Figure 17. Raw acceleration as the helmet is rotated 360° about each axis.

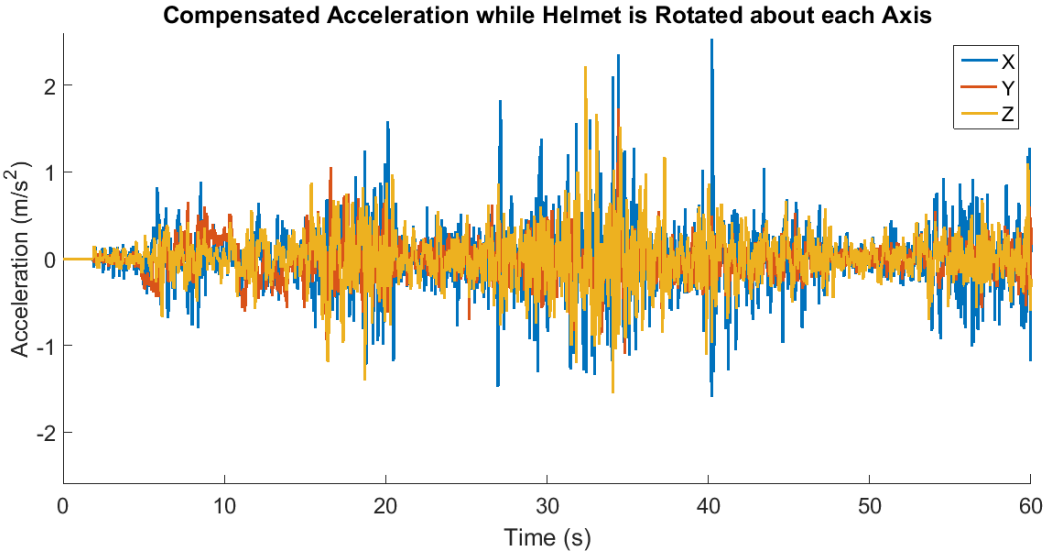

Figure 18. Compensated acceleration as the helmet is rotated 360° about each axis.

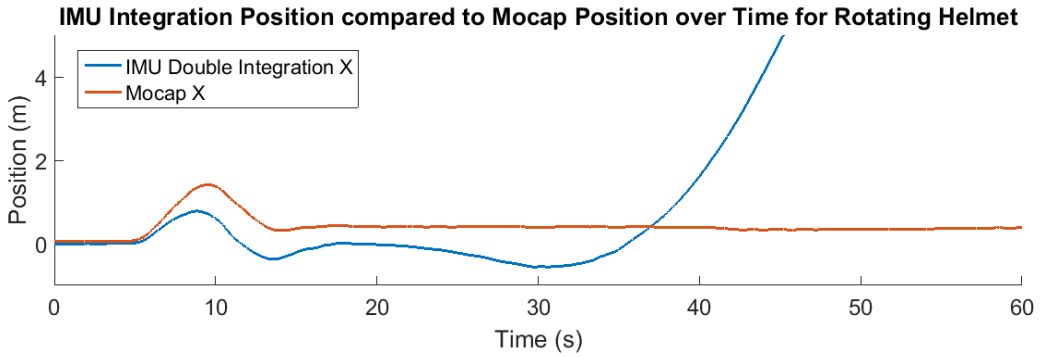

Figure 19. IMU position integration along X-axis while helmet is rotated 360° about each axis. The inertial position integration was compared to truth data provided by mocap. An intial movement forward and backward allowed ORB SLAM to initialize.

In the rotating helmet trial, the error between the integration and truth remained under approximately 0.6 meters in the first 25 seconds. The linear movement trial exhibited significantly more drift at 25 seconds, an error of approximately 5.8 meters. The linear movement trial reached a drift error of 0.6 meters at approximately 7.2 seconds, much earlier than the rotating helmet trial. This suggested that drift error accumulated much more rapidly in certain conditions than in others. In this case, drift accumulated more quickly when the helmet was linearly accelerated back and forth along a single axis than when the helmet was rotated in place.

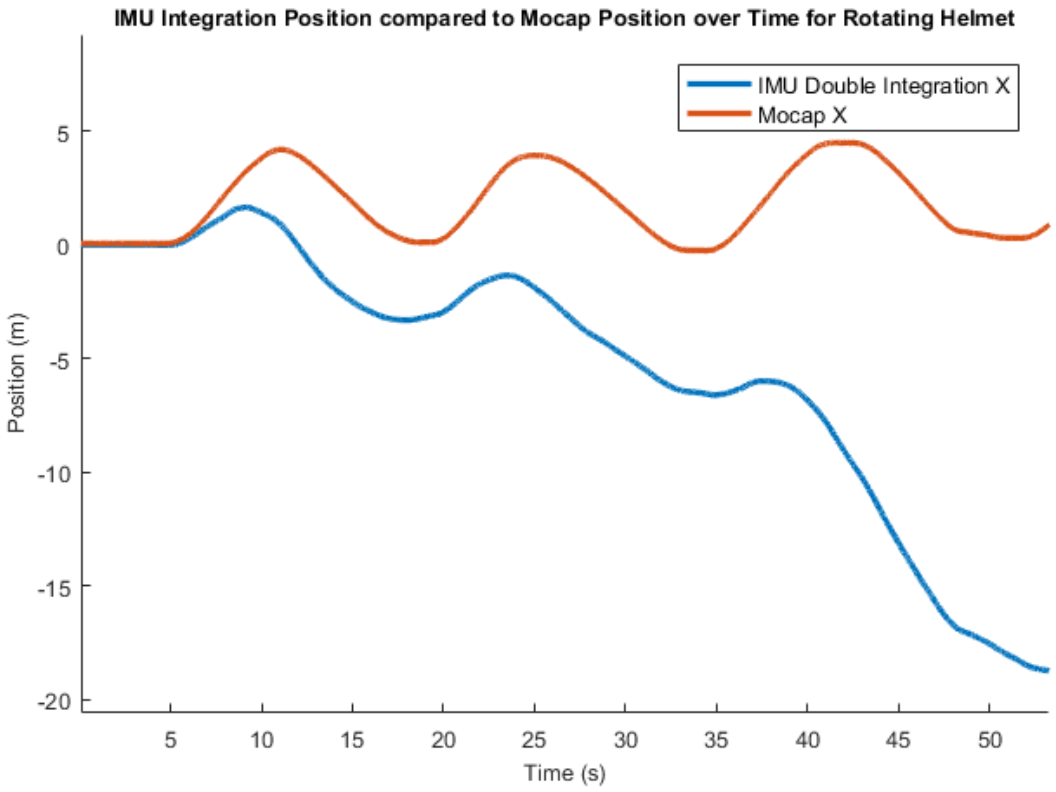

Figure 20. IMU position integration along the X-axis during linear movement. The linear movement involved moving the helmet back and forth along a single axis. The inertial position integration was compared to truth data provided by mocap.

7.4 DETERMINING MAXIMUM ROTATIONAL RATES

Plots were created of the angular velocity of the helmet as reported by motion capture and the IMU, which can be seen in Figure 21 and in further detail in Appendix 1.

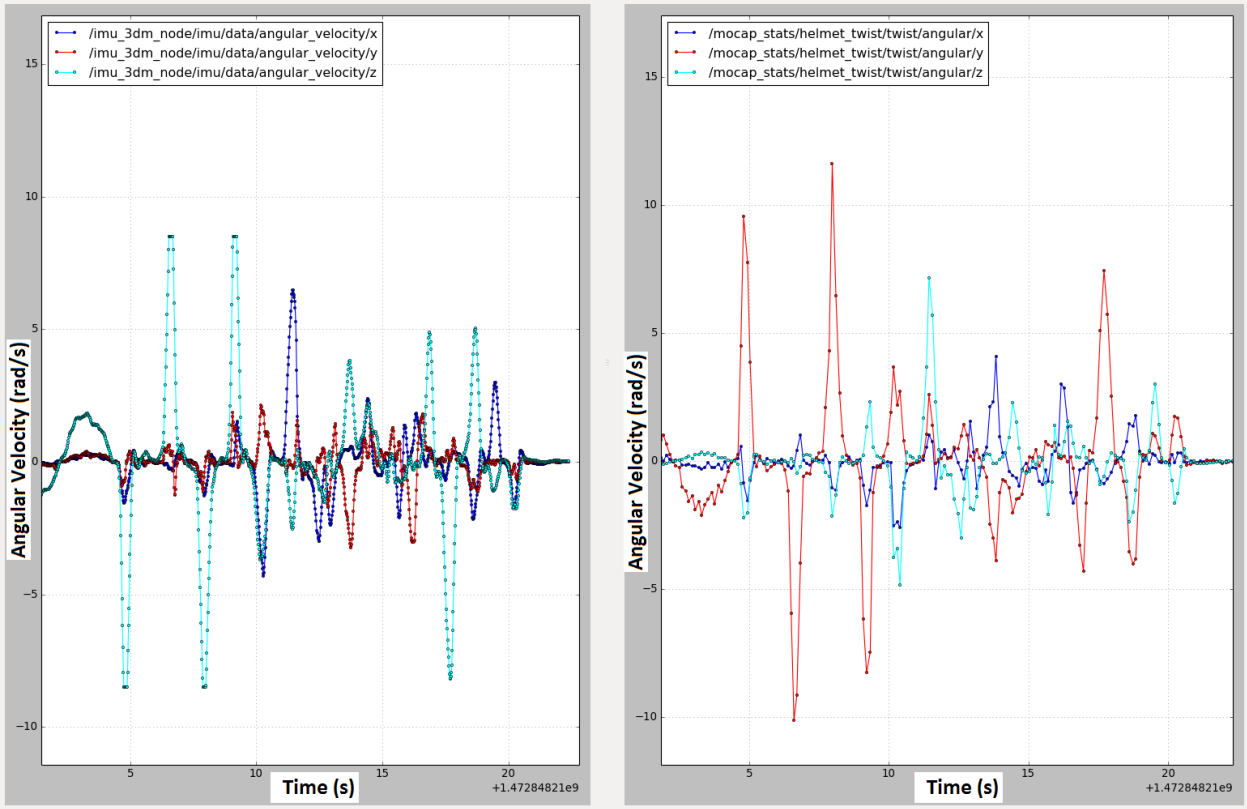

Figure 21. Plots comparing IMU-reported angular velocity to that recorded by mocap. Note how the IMU readings saturate.

From these data, it was easily discovered that the gyro saturated at approximately 8.5 rad/sec, which surpassed the maximum angular rate of 5.236 rad/sec specified for the model. The motion capture angular velocity continued providing measurements reaching 12 rad/sec. This initially seemed to indicate the hardware was insufficient for the testing situations. However, the IMU orientation as reported by the EKF running in the INS was monitored, and it maintained an accurate and absolute orientation throughout these fast rotations. This indicated that while the gyro did exceed its rating, the EKF in the INS system continued to use the changing magnetometer data and knowledge that the gyro was saturated to continue providing accurate orientation estimates. Thus, while the accuracy of the system could improve with a faster gyro, it was suitable for this project to use the EKF-calculated heading. However, if the camera data is processed at a rate greater than 53Hz, it may be necessary to upgrade the gyro so that a more complicated multi-rate data problem is not encountered. In the testing during this project, tracking execution did not

exceed this rate, though it is possible that with GPU acceleration in ORB SLAM it could easily be reached.

ORB SLAM testing demonstrated that the motion blur caused by fast head motions did not have a significant impact on tracking ability. Rather, the most prominent failure-inducing variable was the amount of frame overlap, which was related to the helmet rotation speed. Tracking typically continued with motion blur, but at reduced performance that sometimes led to tracking loss events. There was also a noticeable difference between tracking in new scenes compared to previously mapped areas. In response, later tests were conducted such that rotations occurred only on newly mapped areas, which described the case most likely to be encountered during outdoor localization. Unfortunately, not enough time was allocated to properly conduct the proposed methodology for determining the rotational rate at which ORB SLAM lost tracking in new areas.

These brief tests and observations provided a general assessment for the system's rotation tracking ability. For a system mounted on a person's head, ORB SLAM's rotational tracking was not sufficient for all cases, but the IMU reported accurate orientations. ORB SLAM's tracking ability could be improved by utilizing a spherical camera model rather than the standard pinhole camera model. This would allow objects at the far extremes of the fisheye lens to be used, effectively increasing the frame overlap percent in all cases and increasing the maximum rotational rate.

7.5 SLAM SCALE CORRECTION

With the initial baseline estimate injected into ORB SLAM initialization, the scale was significantly corrected. Figure 22 compares mocap and ORB SLAM outputted XY position with scale corrected using mocap data. Figure 23 shows the same comparison but with inertial position estimates to correct scale in the square tests. Similar charts on outdoor data appear in Appendix 2.

The scale adjustment resulted in a good correction, with a max instantaneous error of about 10cm. Inertial scale correction in the square test was also successful with about 20cm error before rotation, and deviations from the truth were caused by scale drift errors following the tight rotation. This indicated the inertial estimate was accurate enough in cases were movement started quickly before inertial dead-reckoning estimates accumulated errors. The mocap scale correction was also applied to a case involving a circular path, as shown in Figure 24.

These results showed a much larger error. At most, during the initial walk around the circle, the highest error was 95cm before loop closure was achieved. Once the loop closure completed, further tracks around the circle produced errors of 1.3m. While the effect of scale drift was evident by the track that spiraled towards the middle, the loop closure correction did not reduce the error. Furthermore, initial error was unlikely to be caused by scale drift, indicating the scale correction in this test was not as effective. This indicated the position change in multiple directions was not properly being computed, though the arc length as opposed to straight-line distance was not applicable. Further tests are necessary to determine the cause of this failure.

Another issue that sometimes occurred was that the scale correction seemed to vary from different evaluations of the same dataset. Further examination is necessary to determine whether

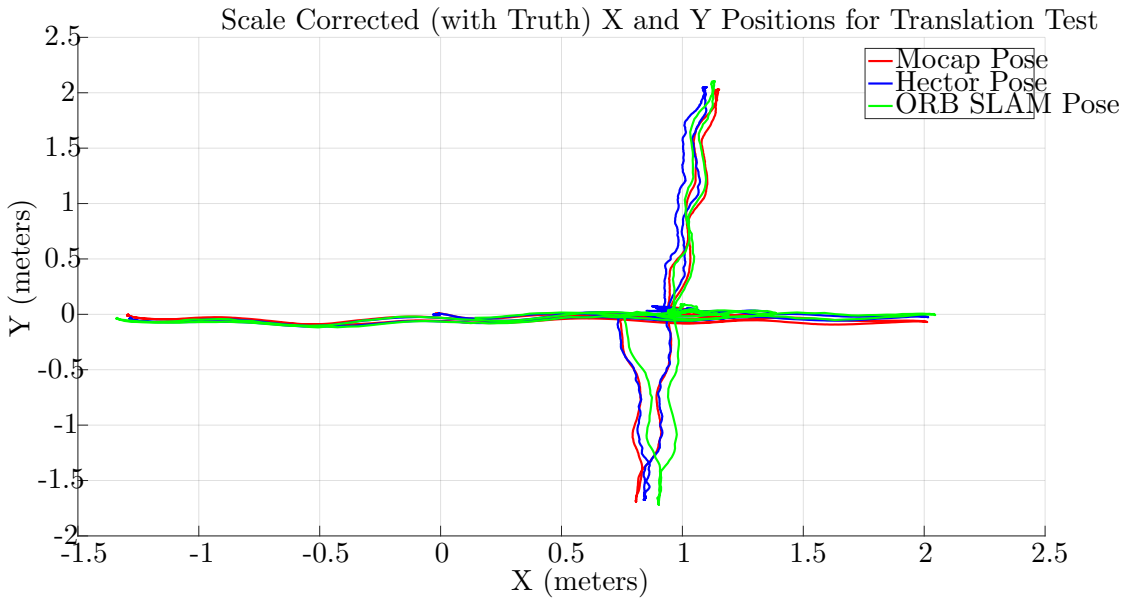

Figure 22. X,Y positions of truth systems and ORB SLAM with scale correction applied to X,Y,Z test. Mocap data used for scale adjustment.

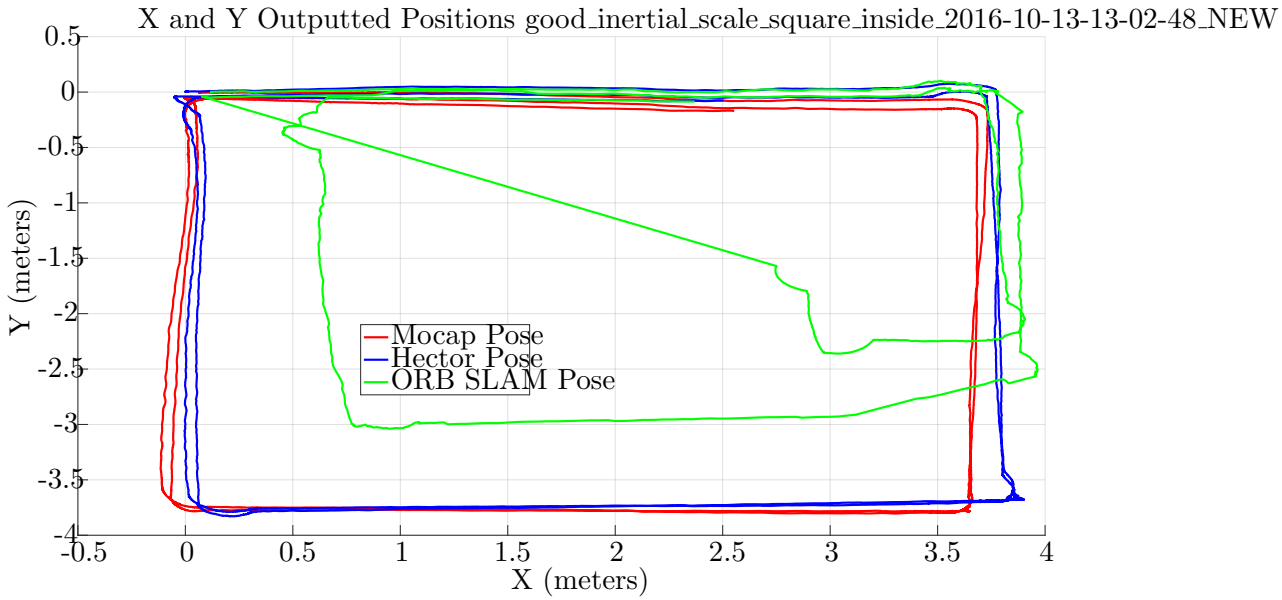

Figure 23. X,Y positions of truth systems and ORB SLAM with scale correction applied to Square Inside test. IMU data used for scale adjustment.

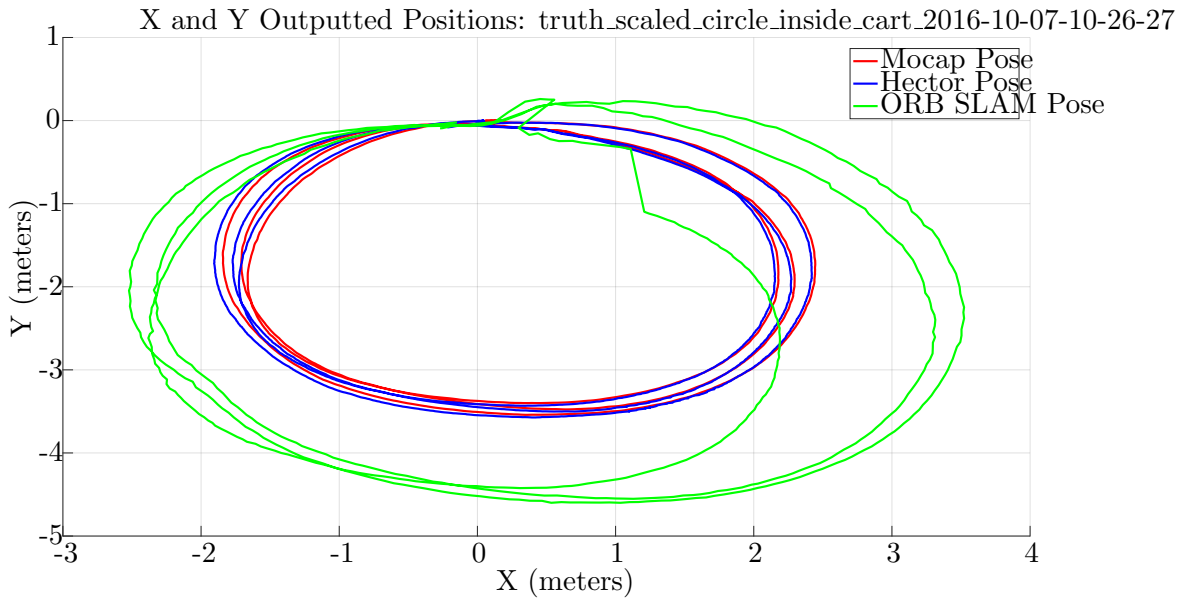

Figure 24. X, Y positions of truth systems and ORB SLAM with scale correction applied to circle test case. Mocap data used for scale adjustment.

this was due to non-deterministic variations within the SLAM algorithm itself or if the scale baseline estimates were suffering from other averaging or timing issues.

7.6 SLAM SCALE DRIFT CORRECTION

Scale drift correction in the ORB SLAM algorithm was implemented to maintain an accurate scale estimate in the SLAM algorithm once the initial scale was established. Refer to subsection 6.8 for implementation details of scale drift correction. The results below show how error accumulated in the position that SLAM estimated with both scale and scale drift correction in place.

Figure 25 shows the path traveled in the XY plane as estimated by ORB SLAM and as mocap system and Hector SLAM reported as truth during the Circle Inside cart trial. The graph shows that, at first, the scale drift correction using mocap data improved the accuracy of ORB SLAM compared to when scale correction was applied (Figure 24). On the left side of the graph, which displays the first half of each path around the circle, the ORB SLAM Pose (shown in green) was tightly clamped to the truth values that mocap and Hector SLAM provided. However, during the second half of each path around the circle, the ORB SLAM pose started to diverge from the truth values. On the right side of the graph, not only does the green line diverge, but it tends to exhibit large jumps towards the true track. In the upper right part of the circle, the green line spikes towards the more accurate mocap and Hector SLAM estimates closer to the center of the circle and returns to an erroneous position many times. The likely cause of this behavior was inconsistencies that emerged in the algorithm as a result of the scale drift correction. The technique used to correct scale drift was able to apply small and potentially only temporary corrections to the position estimate.

Corrections that were in large disagreement with what ORB SLAM would normally estimate were not accepted well. The reason was that the applied correction was unable to correct the feature points on the map, which were a core component of the algorithm. The features suggested that the camera was in one location, but the inertial correction suggested something different enough to create undesired behavior. Typically, the estimate from ORB SLAM was strengthened as more features and key frames were placed. This was a possible explanation for larger inconsistencies on the second half of the circle, and undesired behavior caused significant changes on the second half of the path around the circle but not on the first half.

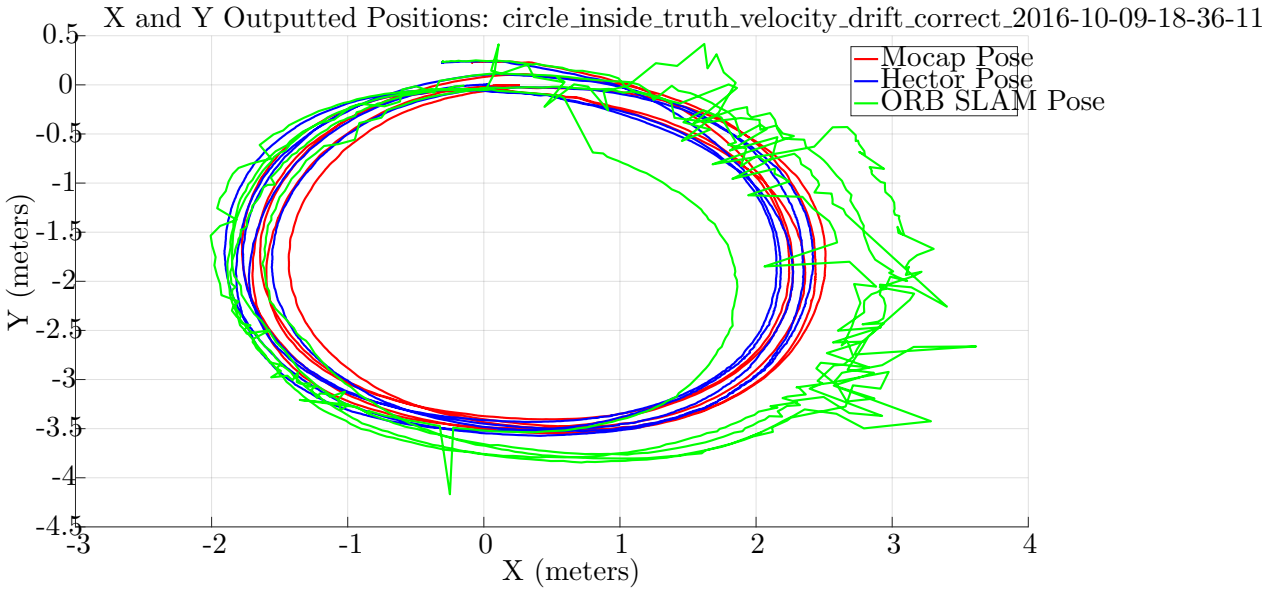

Figure 25. X,Y positions of truth systems and ORB SLAM with scale drift correction applied to circle test case. Mocap data used for adjustment.

Another noteworthy effect was that of loop closure. During one of the paths around the circle, the ORB SLAM pose estimate diverged from truth near the bottom of the circle. The estimate then remained closer to the center of the circle than during any other path around the circle. In the graph, this was shown as the part of the green line that existed inside the right side of the circle. When the estimate was sufficiently close to the starting point at the top of the circle, loop closure snapped the estimate back to the start of the circle and similar behavior as described above occurs again on the next path around the circle.

7.7 ORB SLAM PERFORMANCE EVALUATION

The goal of analyzing the performance of ORB SLAM was to determine the run times of different tasks and subtasks within the algorithm in order to prepare for future work on a real time implementation of the system. Refer to subsection 6.9 performance evaluation data collection and

processing. The following results show the run times for significant tasks within the ORB SLAM algorithm. Tasks were presented in call hierarchies, one for each of the two threads whose subtasks were analyzed. To consider the algorithm's behavior in a variety of scenarios, indoor and outdoor data collected during four trials were aggregated together and analyzed. The following trials were included in the analysis:

- 6 Normal - an outdoor forest trail recorded at moderate walking speed, no loop closure
- 7 Fast - an outdoor forest trail recorded at fast walking speed, no loop closure
- Circle Inside - indoor trial with constant simultaneous translation and rotation, loop closure
- Square Inside - indoor trial with alternating translations and 90° rotations, loop closure

The data collection was conducted on a desktop computer with the technical specifications shown in Table 8. In addition, the data collection was performed on an unmodified version of ORB SLAM (except for the addition of event recording functionality) because it was assumed that the modifications made to the algorithm to improve the pose estimate had a negligible effect on run time.

TABLE 8

Technical specifications of the desktop computer that the run-time performance analysis data was collected on.

Component	**Description**
Processors	Two Intel Xeon(R) CPU E5-2687W, each with 18 virtual cores clocked @ 3.10GHz.
Graphics	Two Quadro K5000/PCIe/SSE2 and an additional NVIDIA card for screen connections
RAM	188.9 GiB
Primary Disk	1 TB Samsung 850 Pro SSD
Operating System	Ubuntu 14.04 LTS 64-bit

To determine which of the threads in ORB SLAM had high potential for improvement, an analysis on each thread's overall task was completed. The bar graphs in Figure 26 and Table 9 display average duration for a processing cycle of each thread.

The 5th and 95th percentile values for each thread's durations were also shown in the graphs to provide good and bad performance boundaries that were resistant to outliers, which were considered to be in the outlying 5%. The Loop Closing thread had a very small average cycle duration because loop closure occurred infrequently in the data sets, and it was important to note that the main task of the thread continued cycling even when loop closer was not needed. The Local Mapping thread

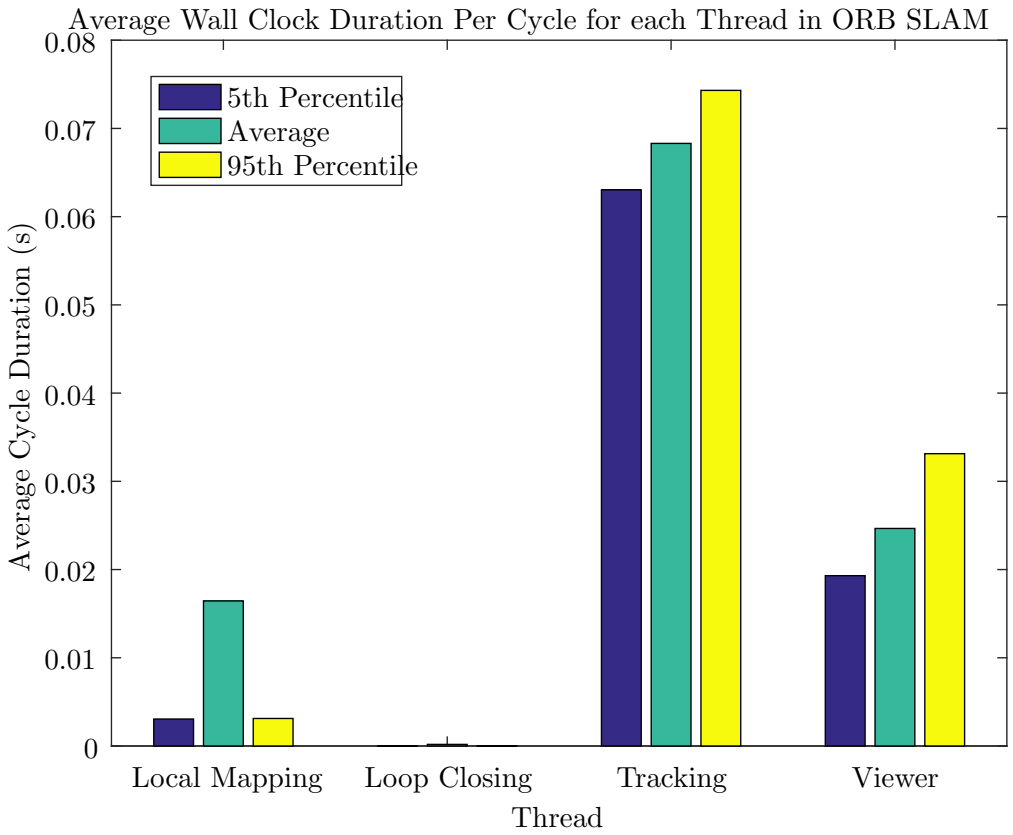

Figure 26. Average Wall Clock Duration Per Cycle for each Thread in ORB SLAM. The 5th percentile and 95th percentile durations for each thread are also shown to provide boundaries for good and bad performance cases that are resistant to outliers (values in outer 5%).

TABLE 9

Thread cycle durations for each of the threads in ORB SLAM. The average durations, 5th percentile durations (good cases), and 95th percentile durations (bad cases) are shown. In addition, the expected maximum processing cycle frequency, which was calculated from the average for each thread, is shown.

	Duration per Cycle (s)			
Thread	**5th Percentile**	**Average**	**95th Percentile**	**Expected Max Frequency (cycles/s)**
Local Mapping	0.003057	0.01644	0.00312	60.82725061
Loop Closing	0.000003048	0.0001818	0.00001874	5500.550055
Tracking	0.06304	0.0683	0.07431	14.64128843
Viewer	0.0193	0.02466	0.03313	40.55150041

had a small but not insignificant average cycle time and may have some room for improvement. One property that Loop Closing and Local Mapping had in common was that their average cycle durations were larger than the 5th and 95th percentile values. Seemingly non-intuitive at first glance, this indicated an upwards skew in the distributions of cycle durations for Loop Closing and Local Mapping. In other words, the values in the upper 5% of values were significantly larger than values in the lower 95%. This increased the average beyond the percentiles while keeping the percentiles low. This result was expected because the Loop Closing and Local Mapping threads each performed an infrequent yet lengthy and computationally intensive task, which skewed the duration distribution upwards. What this meant in terms of run time analysis was that the overall average of duration was likely insufficient to make conclusions about whether the processing for these threads can execute frequently enough throughout operation of a real time system. Even though the overall average may suggest that real time operation satisfied a desired standard, local averages taken from smaller time periods could reveal that under certain conditions, the threads would not be able to process fast enough in local time periods. The Viewer thread had a larger average duration than Local Mapping, but the viewer can be disabled for a real time implementation since it is only used for visualization during development or direct demonstration of ORB SLAM.

The Tracking thread had a large average cycle time, so further analysis was done on the subtasks of this thread. Figure 27 shows the distribution of cycle durations for the Tracking thread. The distribution had a mean of 0.069 seconds and a standard deviation of 0.010 seconds. The histogram demonstrated that there was some spread in the Tracking thread's cycle durations among the four data sets with differing conditions. The amount of spread was not expected to create a problem which could arise from events where Tracking cycles take longer than normal (occur at the high end of the distribution). However, in the event that the high end of the distribution occurred too frequently under certain conditions, lowering the maximum processed frame rate, more optimizations or analysis may need to be done.

As shown in Table 9, the expected maximum frequency for processing cycles of the Tracking thread was $14.64\frac{cycles}{second} \approx 15\frac{cycles}{second}$. Therefore, when running ORB SLAM on the same system on which performance data was collected, one could expect ORB SLAM to accept a maximum of approximately 15 frames per second before dropping frames. A primary goal of implementing a real time system worn by a person is to significantly increase this maximum framerate by decreasing processing time of each cycle of the Tracking thread and ensuring the rate is nearly always met. In addition to the more detailed analysis conducted on the tracking thread, a secondary analysis was also done on the subtasks of the Local Mapping thread.

Before the Tracking and Local Mapping thread were further analyzed, call hierarchies of possibly computationally intensive tasks were constructed through inspection of the ORB SLAM implementation. Figure 28 shows the call hierarchy of the tasks analyzed in Tracking, and Figure 29 shows the call hierarchy of tasks analyzed in Local Mapping.

Tracking thread cycle duration totals among the four trials for different subtasks are shown in Figure 30 and Table C.10. Each task's total cycle duration was represented as a bar in the graph. Each bar is split into time that was further broken down into child tasks (blue), and time spent on other tasks, which were not broken down into subtask children in the analysis (yellow). The most

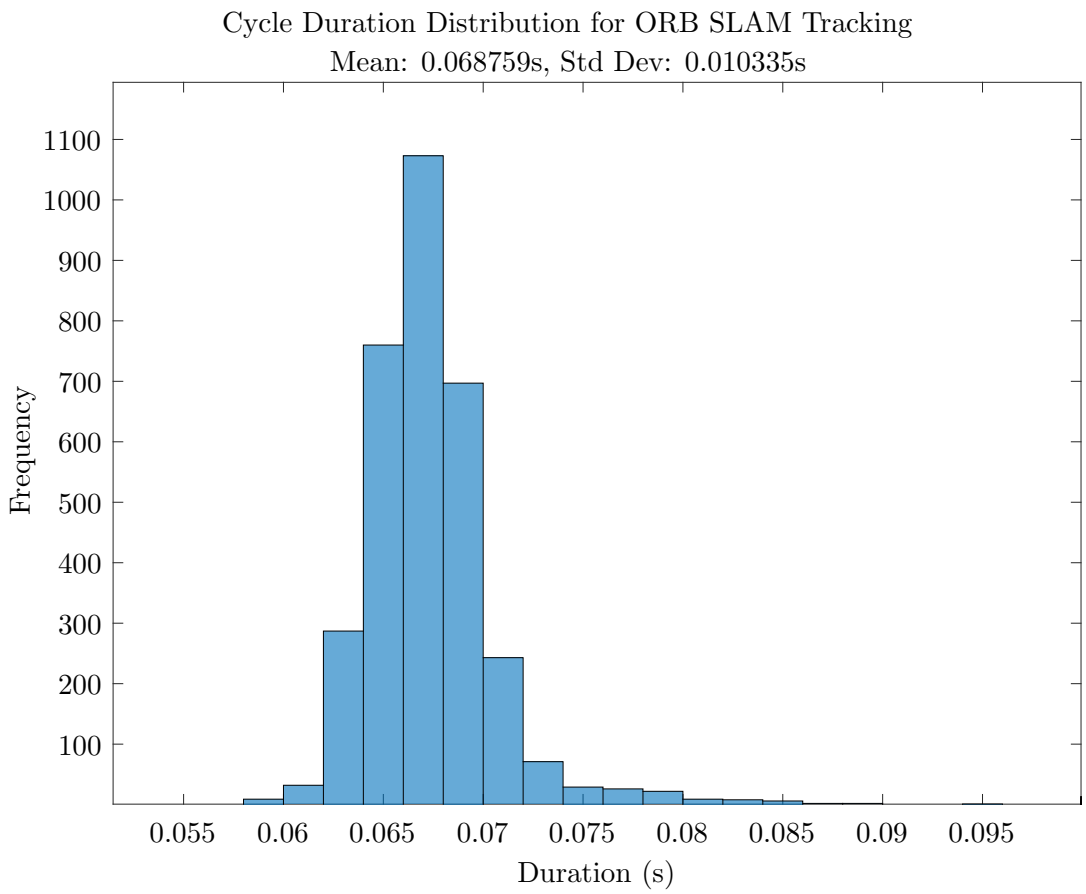

Figure 27. Distribution of Cycle Durations in ORB SLAM Tracking Thread

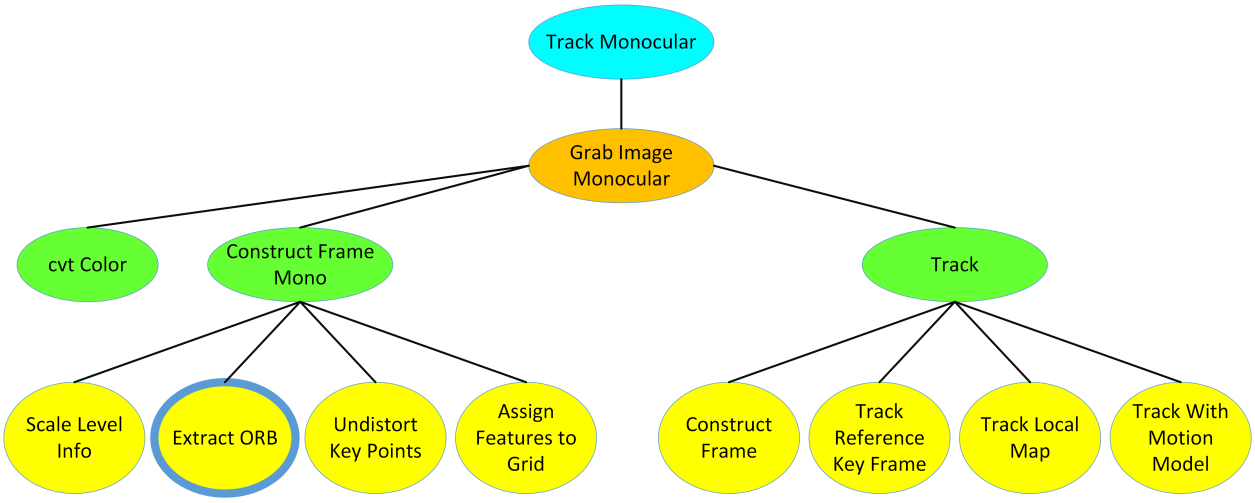

Figure 28. The call hierarchy of analyzed tasks in the Tracking thread of ORB SLAM

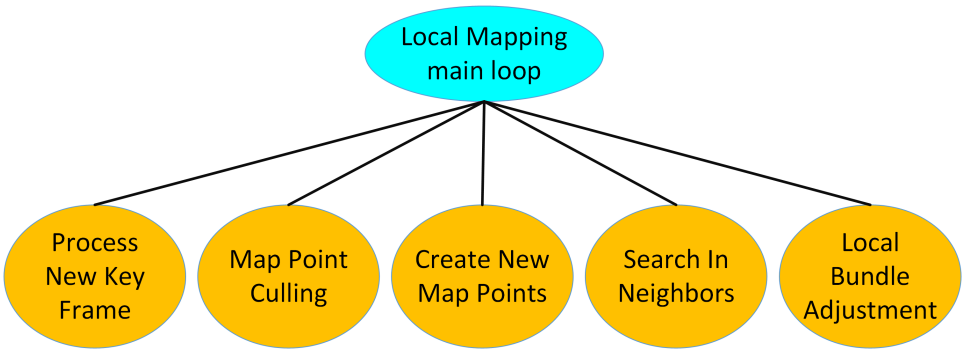

Figure 29. The call hierarchy of analyzed tasks in the Local Mapping thread of ORB SLAM

significant piece of information that this graph revealed was that the Construct Frame Extract ORB task consumed a significant portion of run time. The Construct Frame Extract ORB task executed for a wall clock time of 195 seconds while the overall Track Monocular task executed for 230.5 seconds. This indicated that the Construct Frame Extract ORB task, according to the trials in the analysis, consumed approximately $\frac{195s}{230.5s} = 0.8460 = 84.60\%$ of the total wall clock time for the Tracking thread. This remarkable discovery provided a significant opportunity for future work on a real time implementation, and this was further explained in Discussion subsection 8.4.

Figure 31 and Table C.11 show the total amount of time spent on each task among the four trials for the Local Mapping thread. The results indicated that the Create New Map Points subtask was the most significant task in terms of run time within the main loop of the Local Mapping thread. This subtask, according to the analysis, consumed approximately $\frac{113.4s}{285.35} = 0.3974 = 39.74\%$ of the total wall clock run time in a cycle of the Local Mapping main loop on average.

For additional information on the run times of particular tasks in the Tracking and Local Mapping threads, refer to Appendix 3. The appendix consists of tables that contain numerical wall clock run-time data and the percent of run time that each task consumed.

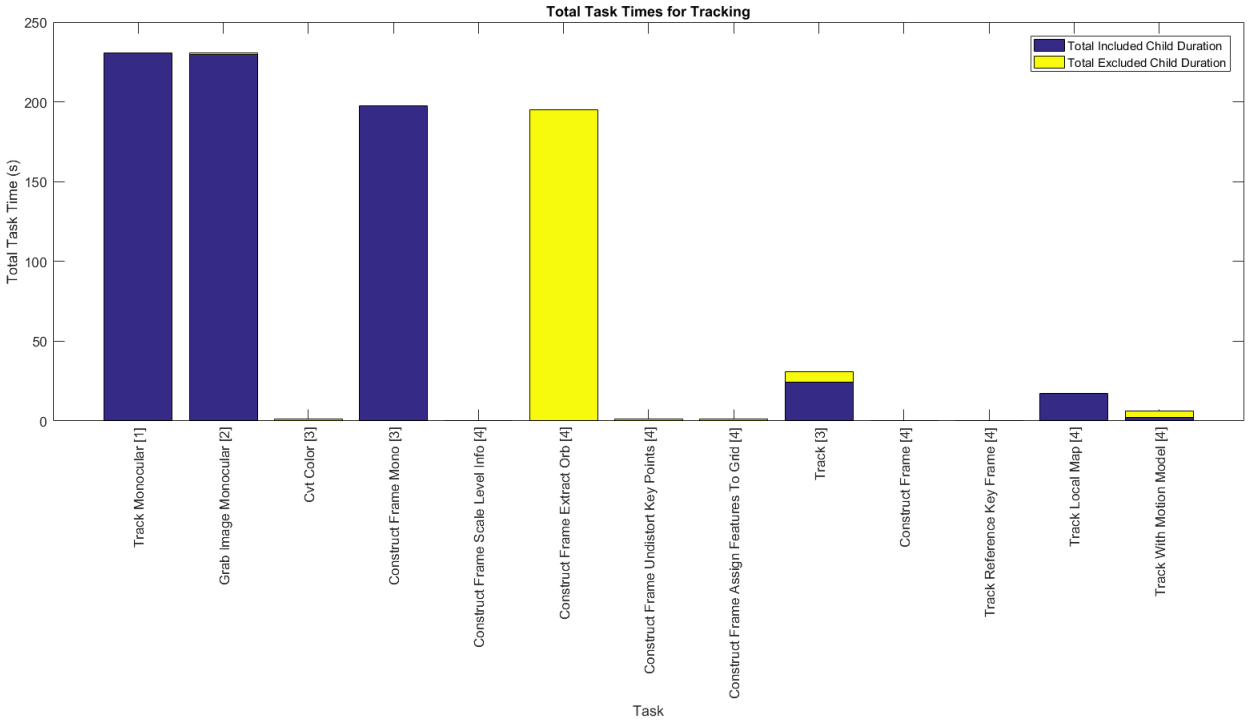

Figure 30. Total time spent on subtasks within the analyzed call hierarchy of ORB Tracking thread. Data is aggregated from four trials with differing conditions. Displays analyzed subtasks of the Tracking thread in ORB SLAM on the x-axis and total amount of time the algorithm spent on each task on the y-axis. The blue section of a bar indicates run time that was further subdivided and shown elsewhere in a child task. The yellow section of a bar indicates run time that was not divided any further. Therefore, a purely yellow bar represents a leaf in the analysis call hierarchy, and the sum of all yellow sections results in the total time spent on cycles of the thread. A bar with blue and yellow sections only had some of its subtasks (represented collectively by blue) analyzed.

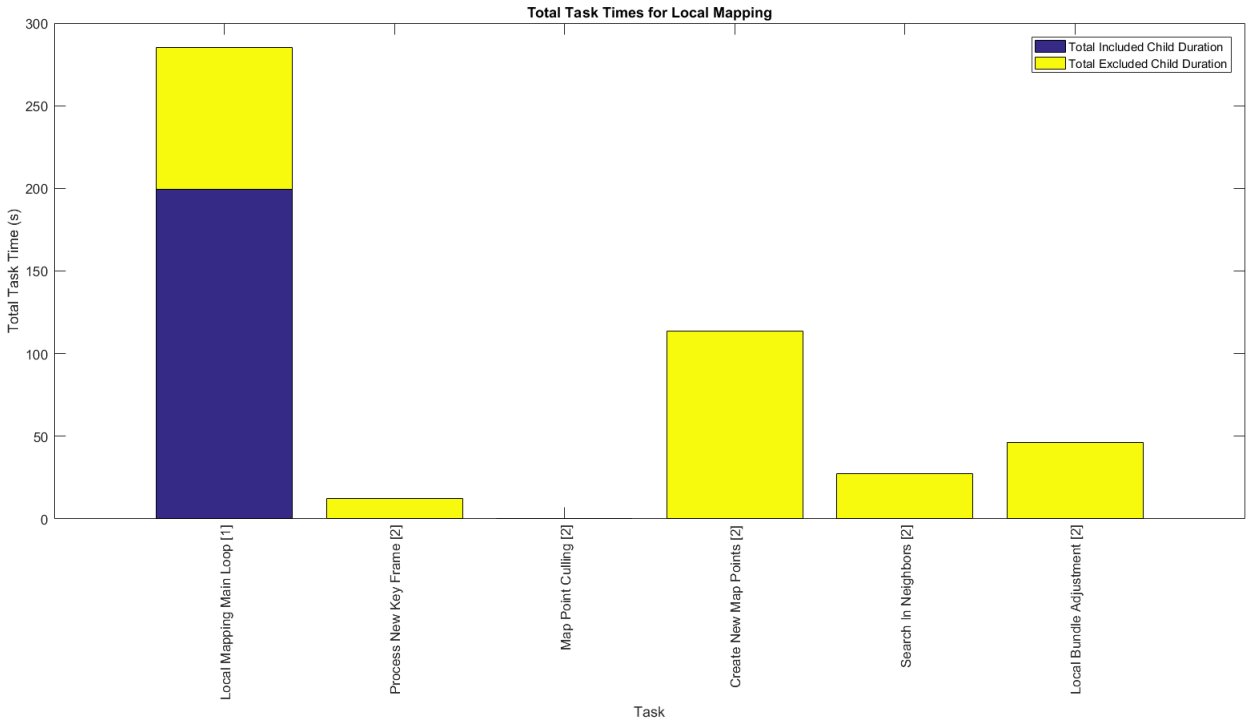

Figure 31. Total time spent on subtasks within the analyzed call hierarchy of ORB Local Mapping thread. Data is aggregated from four trials with differing conditions.

7.8 AR HUD DEMO AT MIRROR LAKE

The AR demo at the lake was, in terms of performance, a failure. It took a couple hours at the competition to setup what was working days before. Additional time was spent trying to determine why the octomap was not displaying as expected. Since the basic display of a simplified map was working in the lab, it was believed that when the actual user position was staked on the lake, any rotations could be completed in the field. However, while the lakebed and user position and orientation were correct in RVIZ and indicated the user was properly located along the lake and looking at the lakebed marker array, the transform did not seem to have an impact on the user's perceived orientation and position on the Android application. This resulted in the user not having a correct perspective of the lakebed octomap. After testing several configurations and moving the map around, the issue was determined to be that the Android application did not account for TF to determine proper positioning of the markers. This and other issues were found that day:

- Android application did not utilize TF
  - Maps needed to stay in the world frame.
  - Pose must carry the positional data in the world frame.
  - Implication: pose must be used to rotate around the map frame so that the map appears correct to the user.
- Android app could only display 2000 objects.
  - If the resolution was changed, the map needed to be manually cleared.
  - There would need to be special handling for clearing and reloading the map as the ID's of the markers would change. Completing this behavior would be difficult.
- It required approximately 30 seconds for a 14-level deep lake octomap to load on the Android application.
  - The app was entirely unresponsive.
  - However, after the loading period, rendering was generally smooth.
- A cropping node for only keeping desired portions of the map and discarding other portions of the map that were outside the range of view was not implemented.
  - This could solve some loading time and display issues.
  - However, there was no way to handle cropping the markers and reloading the map on the glasses, which was necessary if changes occurred.
- The application automatically clipped far markers, which was not ideal for viewing the entire lake.

While the performance of the demo was not acceptable, the knowledge gained from the experience was a success. The issue relating to TF was not unexpected in hindsight, but it did

indicate that special nodes for properly reorienting the pose were necessary. Just because markers and positions were correct in RVIZ did not mean they would be correct on the Android application. Additionally, there would need to be extensive work in optimizing the display of so many markers on the current Android application.

Through tests utilizing the Glasses IMU and the Android tablet IMU, there were substantial drift issues in the orientation estimates from both systems. The glasses tended to supply a fair orientation estimate in most cases, but if the glasses were rotated quickly while worn on the head, the view would start drifting and never recover. This would indicate the system implementation was very gyro-dependent and did not properly fuse the magnetometer estimate, unlike the Microstrain IMU on the helmet. The tablet IMU was fairly accurate and resumed the right position if rotated too quickly, but it exhibited small drifts about the true heading of the device. This caused some unintended motion and produced an error between the displayed location of the markers and the actual heading of the user.

8. DISCUSSION AND RECOMMENDATIONS

8.1 REFINING THE APPROACH

The initial design of the system differed from the design that was ultimately pursued. Figure 1 outlined a combination of approaches including dead-reckoning, pedometry, visual-inertial odometry and GPS to improve the position estimate. After comparing these approaches, a primary approach, deemed most essential, was identified. The chosen approach was to rectify the scale in the ORB SLAM algorithm using input from inertial sensors and to loosely couple inertial position estimation with the algorithm. Although a more modular approach which could enable the SLAM algorithm to be chosen independently from inertial corrections, it could not support assisting internal optimizations in SLAM with external information. This could theoretically result in a better estimate than a higher-level sensor fusion.

The advantage to coupling the inertial data with SLAM was that it can place restrictions on the distances that SLAM considered. Concretely, this was done by applying corrections to the estimate of current velocity in the motion model of ORB SLAM (scale drift correction), and this approach could not be achieved with a strictly modular approach as in the original system design. The SLAM algorithm used an initial baseline estimate that established translation and rotation between the first pair of key frames. Normally, SLAM did not have enough information to settle on accurate and consistent baseline values, resulting in a poor initial scale. SLAM scale also drifted as features entered and exited the current frame. However, with the additional information that the IMU provided, SLAM may be able to periodically adjust the scale to partially prevent drift.

As the focus of the project narrowed, the core component of the system, which was the direct output of the robust position estimate, moved away from an abstract notion of sensor fusion. The ORB SLAM algorithm coupling with inertial data became the focus. However, large opportunities for future work lie in a sensor fusion node to provide optional but beneficial assistance to the proposed visual-inertial system.

8.2 INTEGRATING MULTIPLE SENSOR ESTIMATES

The approach in this project, discussed in the previous subsection 8.1, enabled the SLAM algorithm to function properly during typical operation of the system. Before the solution was put in place, ORB SLAM produced unscaled data that was, by itself, useless for the final goal of providing pose to augmented reality applications. What the solution did not provide was a way to reinitialize the ORB SLAM algorithm if tracking failed. The solution also did not provide a way to keep the system functioning at its best with the available resources while SLAM was not operational.

As illustrated in Figure 1, a sensor fusion node could receive data from three sources: GPS, Pedometry, and one or more instances of Visual Inertial Odometry. One could think of the GPS input as a course initial input among the three inputs. Even though GPS may not be available at any time, it provides absolute position when available. After adding GPS to the fusion process, pedometry estimates could be added. In theory, the pedometry node would be less resistant

to quadratic integration drift since it would use acceleration values directly to count footsteps. Nonetheless, the node may be implemented with an average stride estimate, which would cause the accumulation of outputted changes in position to drift from truth by a constant factor. This would result in linear drift. Odometry estimates could be used to further refine the position estimate beyond the discreet increments inherent to footsteps and GPS fixes (or provide the pose estimate by itself if GPS and pedometry are unavailable). A Kalman Filter may be an ideal way to combine these three positional inputs.

Once the sensor fusion process is implemented, its output could assist the ORB SLAM algorithm in two ways. The first way could be in the Reinitialization and Map Storage modification to the SLAM implementation (Figure 10). When ORB SLAM lost tracking of features, it attempted to reinitialize and regain tracking by recognizing previously mapped features. In the future, a new system could be developed that allows seamless integration of multiple position estimates to assist ORB SLAM with recovering from tracking loss. Suppose that ORB SLAM begins tracking and then fails. For tracking to resume, ORB SLAM needs to relate the current position with the previously saved map. The Map Storage part of the modification now stores the current map. Meanwhile, the sensor fusion node could utilize purely INS driven techniques to persist an estimation of position. The sensor fusion node can deliver the change in position between the point at which SLAM failed and the point of reinitialization to the Reinitialization and Map Storage modification in ORB SLAM. When the modification receives the position estimate, it takes note of the translation and rotation between the stored map and the newly started map. An alternative or additional approach would be to maintain a single map and utilize the existing ORB SLAM functionality to identify common features between the map before failure and after manually reinitializing. This would have to properly add constraints that ORB SLAM places on key frames between sessions of persistent tracking. In other words, constraints and loop closures could cross between any number of continuous tracking sessions if such an alternative approach were pursued. Future work could include tests to determine whether this alternative produces desirable behavior or propagates errors due to unreliable constraints across tracking sessions.

The other way in which the sensor fusion approach can assist the ORB SLAM algorithm is intended to occur during normal, uninterrupted operation of the algorithm. When GPS is available, the sensor fusion node could fuse GPS and pedometry to periodically correct the position of ORB SLAM. Absolute position correction would be a possible future modification to the ORB SLAM algorithm that could receive estimates of absolute position from sensor fusion and update the position in ORB SLAM. Implementing this modification would require uncovering a section of the algorithm that can accept a spontaneous correction in position without resulting in later inconsistencies when the information propagates. This may be particularly difficult because the correction might not agree with the constraints that SLAM places on the key frames for bundle adjustment and loop closure or with feature points being placed on the map. In this situation, after an absolute position correction, it would only be a matter of time before loop closure or local bundle adjustment triggers the outmoded constraints to undo the correction. A potential solution to this would be to leverage the global bundle adjustment that loop closure uses. Similar to the way loop closure poses a constraint two close keyframes, the absolute position correction could place a constraint that the most recent key frame is near the absolute position received from sensor fusion.

Global bundle adjustment could, in theory, seamlessly accept this new constraint and re-optimize previous constraints in accordance with the new constraint during loop closure.

8.3 SYSTEM HARDWARE AND SOFTWARE IMPROVEMENTS

The system hardware was very good, but there were a few ways system performance could be improved by hardware changes. First, the camera lenses should be swapped with versions that allow the sensor to see the entire FOV of the lens. Currently, the camera sensor cannot see the full sphere of the fisheye lens; the bottom and top are cropped off. While this was acceptable for the pinhole camera model currently utilized and increases resolution for objects in the frame, for more effective tracking at high rotational rates, the full 190° FOV should be utilized with a spherical camera model.

Secondly, the black and white cameras should be tested to determine if they provide better results than the color cameras. The color cameras were used as the color space conversion on the black and white cameras was too slow to record video on the Jetson at a faster rate than the color camera. While color provides additional and useful information that is critical for people to quickly distinguish objects in the map, mono sensors typically exhibit less noise than color sensors. This would reduce drift error.

Thirdly, a new IMU that can handle the faster head rotations of a person without saturating the gyro should be purchased. A model that can record angular rates up to 13 rad/sec should suffice. The EKF in the INS system worked very well, but if faster rates are needed to properly compare camera motion to IMU rotation, it will be needed. Relatedly, the ROS driver for the IMU should be modified so that IMU data (linear accelerations, angular velocities) will be reported at higher sampling rates in order to more closely match the camera capture rate.

Fourthly, the Jetson TX1 processor seemed to be one of the best options for low-power embedded processing. However, the slower ARM processor is likely to be a bottleneck for the completed system, and achieving fast ORB SLAM execution rates may be difficult without significant development time spent optimizing the algorithm for the limited hardware. At least one group has possibly achieved an ORB SLAM execution rate of up to 20 FPS on the Jetson [31], but additional overhead of extra sensor fusion nodes could reduce this. Furthermore, it would seem improbable that this system could handle two instances of ORB SLAM running with two separate cameras at once if a sensor fusion approach utilizing two instances was considered. Likely, NVIDIA will release a new model soon which should have improved processing capability, but these limitations may still apply.

Finally, there could be errors in the mocap calibration, which would account for the discrepancies between Hector SLAM and mocap poses. More controlled tests should be conducted so that the actual movements can be verified between the two systems. An industrial robot arm would be most suitable for this, but moving the helmet carefully along a track or other guided method would suffice.

8.4 ACHIEVING REAL TIME IMPLEMENTATION ON JETSON TX1

The goal of the ORB SLAM run time analysis was to prepare for future work on implementing the position estimation system to run in real time on the Jetson TX1 board and eventually, on the ODG Glasses. The results showed that the crucial Tracking thread, which processed incoming frames, likely had the most room for improvement. The subtask within the Tracking thread that had the largest duration on average was the ExtractORB function. This task consumed 84.60% of the total run time of the Tracking thread (computed from data aggregated from four trials with differing conditions). The job of the ExtractORB subtask was to extract ORB features from each frame for feature tracking. This occurred during the construction of a Frame object in the algorithm and the call to this function occurred on line 136 in file $ORB\_SLAM2/src/Frame.cc$ of the open source ORB SLAM 2 implementation, available on GitHub. [29] [8]

ExtractORB was a bottleneck with significant opportunity for future work to support a higher frame rate in ORB SLAM and run in soft real time on the Jetson. To decrease the run time of the ExtractORB task and the average cycle duration of the Tracking thread as a result, ExtractORB can be GPU optimized. According to the NVIDIA VisionWorks web page, the VisionWorks API includes the FAST Corners and the FAST Track computer vision primitives. [1] More research on the capabilities of the VisionWorks API will be necessary to determine if the other ORB feature extraction primitive, Rotated BRIEF is available through the API or if the API could be used for secondary optimizations to ORB SLAM. Investigating the applicability of and leveraging these primitives to GPU optimize the ExtractORB task is recommended.

In addition, if later testing indicates that the run time of the Local Mapping thread needs to decrease, the run time analysis revealed that the Create New Map Points subtask in the thread consumes a large amount of run time. This task consumes approximately 39.74% of the run time of the Local Mapping main loop on average. Other potential opportunities for optimizing Local Mapping can be found in Table C.11.

8.5 REFLECTIONS AT MIRROR LAKE

The issues discovered with the AR platform during tests at Mirror Lake (subsection 7.8) indicated that this remains a nontrivial problem to solve. Though the theory was easy for implementing the HUD glasses, the actual implementation was complicated by several issues. Development is needed for making the application more robust and easy to operate. Power consumption also remains an issue, both on the HUD glasses and on the Jetson board, which limits the potential field time of the system. Once the system is functioning properly, the accuracy of the AR overlay needs to be established. One method of verifying this would be to compute the percent overlap between the actual object and the displayed version of the object, either through video recording looking through the HUD or through other calculations. In addition, the scale and range of the object would need to be tested.

There also remains a data interaction issue to solve. While the user could in theory view the map collected by the robot and objects found in the map, there is no established way for the user to quickly select objects in three-dimensional space. The best method for selection on

a HUD has not yet been determined, but hand-held options have been considered. Eye-tracking systems are miniaturizing and declining in cost, and binocular systems allow for three-dimensional gaze acquisition. Some companies are manufacturing hardware that can fit within existing AR and HUD systems, which opens up possibilities for fielding binocular eye tracking in the helmet mapper system. [32] However, some experimentation should be conducted to determine the suitability of such interaction mechanisms. Accuracy of depth perception with binocular vision is limited to a particular range, so long-range selections may have to rely on a secondary mechanism.

8.6 ATTAINING THE FINAL GOAL

A great deal of work remains in making the system robust to sensor failure and improving estimations through all possible avenues. As previously discussed, sensor fusion can adequately handle fusing the proposed visual-inertial odometry solution with GPS and pedometry estimates. A package known as Robot Localization is a potential candidate for quickly utilizing previously existing packages for better localization. [33] In addition, the following assumptions were made concerning the correction of the ORB SLAM algorithm and need to be addressed in the future:

- System initialization is expected to be quick. If too much time is taken between system start and the selection of the first keyframe pair, the inertial position estimate will drift. In these cases, other sensors should be utilized for baseline estimation or further enhancements should be made to properly synchronize the camera motion with inertial data. However, Google Tango is able to accurately establish scale with just the IMU and camera using a static calibration procedure. Whether the scale is established before or after major movement is unknown.
- The inertial sensor measurements were captured at a slower rate than the camera images, but the camera images are not processed faster than the sensor data. A multi-rate problem may exist but has not manifested itself because the image processing time is longer than the sample interval for both the INS and camera.
- Camera calibration was assumed to be correct and accurate. However, there are signs of incorrect calibration, such as points in the middle of free space. This would indicate calibration was not as accurate as it could be.
- Transforms in the ROS interface are currently hard-coded for mocap and the right front camera, and to complete IMU scale correction, these transforms were temporarily modified. Given more time, a proper lookup between message frame id and the target frame should have been conducted, which avoids this time-stressed approach. Additionally, since ORB SLAM and Hector SLAM define the map origin at system start, the delta of the mocap pose was taken so that it had a similar effect. In the future, this should not be done as it results in a loss of transformation data and makes it more difficult to adjust map positions with other data such as GPS and earth-centric IMU orientation frames.

Map coordinate frames must also be standardized to allow multi-robot and person systems to function cooperatively. The maps must be staked on known earth-centric locations to allow easy

integration with existing data. HUD interfaces need additional development to improve capability and usefulness.

Finally, while the system properly estimates position, taking this position and creating a dense map of the environment has not yet been fully attempted. Tests have been conducted with feeding ORB SLAM positions into LSD SLAM to improve the map creation, so while this capability exists, it remains to be well-proven. In addition, it seems unlikely that LSD SLAM could be fully implemented on the Jetson without reducing map quality, as the program proved very memory and computation intensive. Hopefully, this will prove a non-issue as hardware is more efficiently utilized and new, lighter-weight algorithms are developed

9. CONCLUSION

This project achieved accurate pose determination using only passive sensors in GPS-denied environments. Collection of data during the various test cases proved invaluable for identifying issues and testing algorithms. Coordinate systems were successfully matched and transformed, and an inertial dead-reckoning calculation proved crucial. Extensive testing to understand system limitations allowed assumptions to be justified and provided a basis for improving the system. Utility of the ORB SLAM solution was significantly improved by tightly coupling inertial data with visual tracking. Scale issues inherent to monocular SLAM were solved by properly estimating an accurate baseline, and scale drift mitigation was attempted by augmenting the ORB SLAM motion model with external sensor data. Through additional development, the proposed position estimate system will enable localization-driven augmented reality for intuitive interaction with autonomous systems.

A APPENDIX A

Additional plots showing the saturation point of the gyroscope included in the helmet mapper INS.

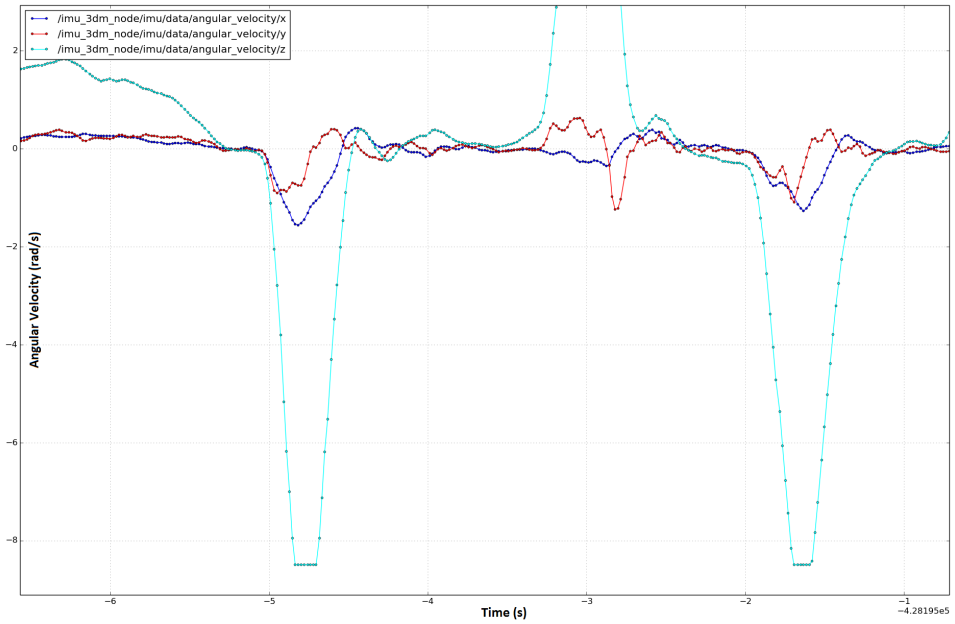

Figure A.32. Gyro Saturation Detail During Head Yaw Movement

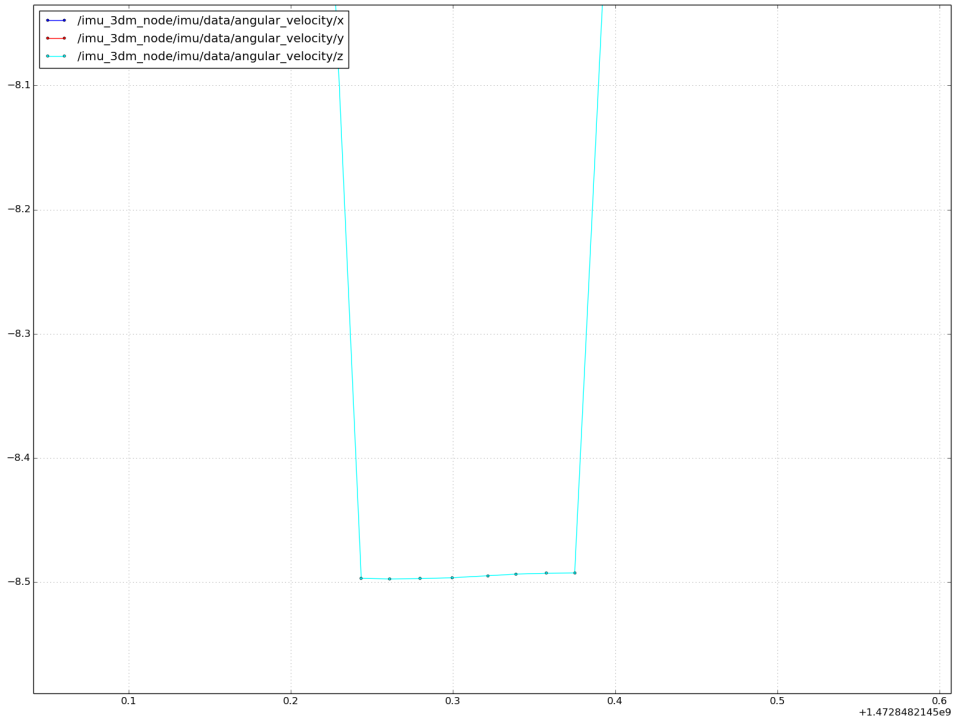

Figure A.33. Gyro Saturation Occurring at 8.5 rad/sec

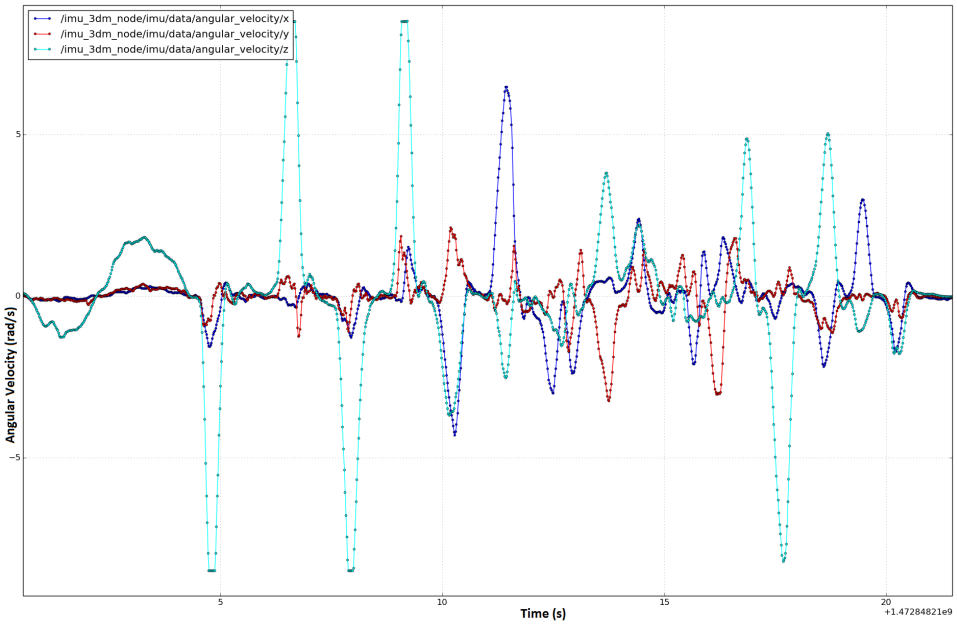

Figure A.34. Angular Velocities During Entirety of Head Turn Testing

B APPENDIX B

Additional inertial-based scale correction results in outdoor data sets.

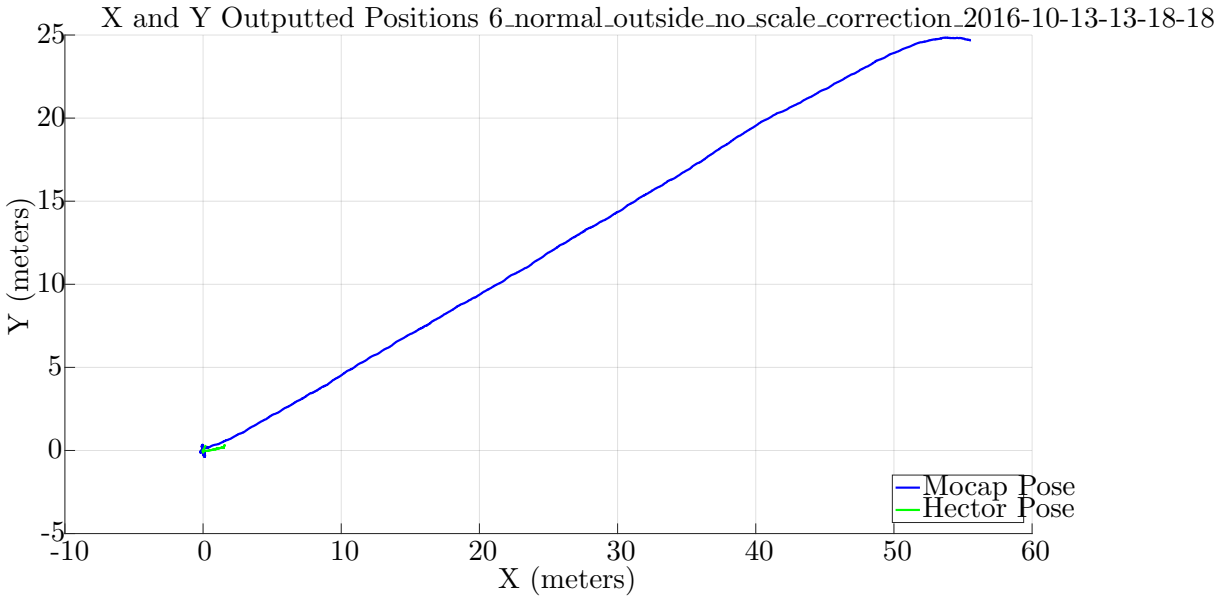

Figure B.35. Outdoor Test Results of ORB SLAM with No Scale Correction. The results have very different scales. Note: Legend is incorrect: Blue is Hector Pose, Green is ORB Pose.

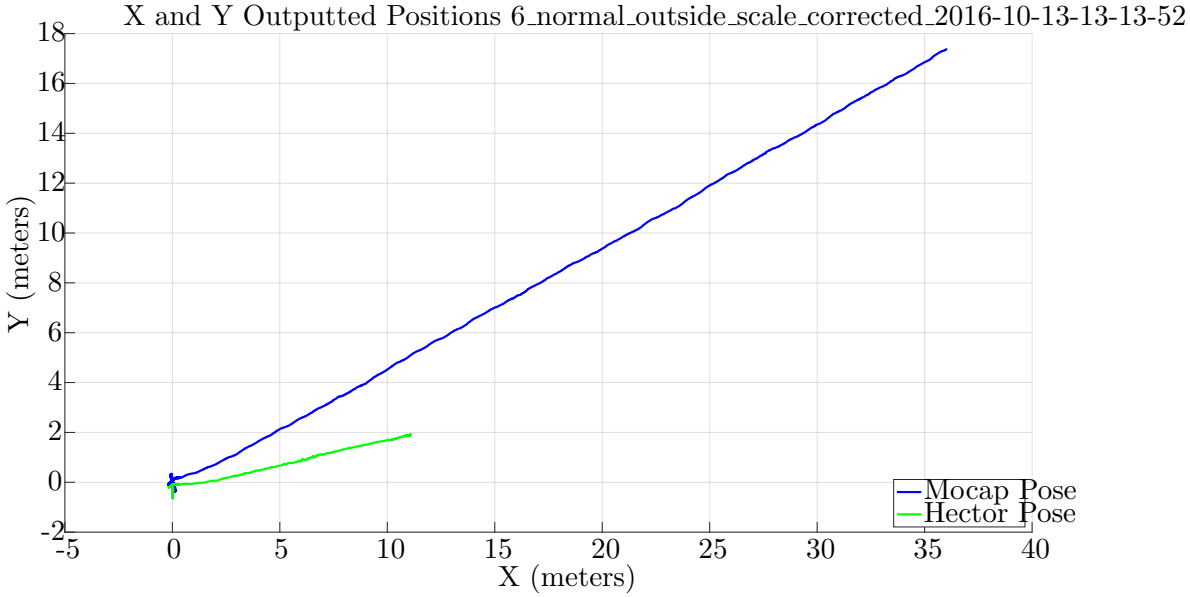

Figure B.36. Outdoor Test Results of ORB SLAM with Inertial Scale Correction. The difference in solutions is improved but still apparent, indicating further changes are necessary. Note: Legend is incorrect: Blue is Hector Pose, Green is ORB Pose.

C APPENDIX C

Numerical data for the ORB SLAM run-time performance evaluation are shown below in Table C.10 and Table C.11. Data was aggregated from four trials with differing conditions. Note that the tables report data down to a call hierarchy depth of 4, but the analysis covered a depth of 6. The table report total times spent on tasks throughout the four trials. The values have no absolute significance but are important in relation to each other. Below are descriptions of each column in the tables:

- Hierarchy Depth - shows the parent, child structure of the tasks.
- Total Included Child Duration - the total amount of time spent executing all of a task's child tasks that were also analyzed (due to further subdivision of tasks in the analysis call hierarchy).
- Total Excluded Child Duration - the total amount of time spent on child tasks that were omitted from the analysis. If a task is a leaf in the hierarchy tree, all of its time will be under Total Excluded Child Duration, since it was not subdivided any further and has no child tasks that were also analyzed.
- Total Time on Task column - the sum of the time for both included and excluded child durations.
- Percent of Total Cycle Time - used to show how much of the total wall clock run-time the task consumes on average.

TABLE C.10

The task durations of each analyzed task within the Tracking thread.

Task	**Hierarchy Depth**	**Total Included Child Duration (s)**	**Total Excluded Child Duration (s)**	**Total Time on Task (s)**	**Percent of Total Cycle Time**
Track Monocular	1	230.5	0.07054	230.57054	100
Grab Image Monocular	2	229.5	0.9641	230.4641	99.95383625
cvt Color	3	0	1.302	1.302	0.564686191
Construct Frame Mono	3	197.5	0.07541	197.57541	85.68978934
Construct Frame Scale Level Info	4	0	0.01356	0.01356	0.005881064
Construct Frame Extract ORB	4	0	195	195	**84.57281663**
Construct Frame Undistort Key Points	4	0	1.443	1.443	0.625838843
Construct Frame Assign Features to Grid	4	0	1.021	1.021	0.442814594
Track	3	24.37	6.282	30.652	13.29397936
Construct Frame (copy contructor)	4	0	0.3835	0.3835	0.166326539
Track Reference Key Frame	4	0.02929	0.0001038	0.0293938	0.01274829
Track Local Map	4	17.29	0.1913	17.4813	7.581757843
Track With Motion Model	4	2.346	4.133	6.479	2.809986046

TABLE C.11

The task durations of each analyzed task within the Local Mapping thread.

Task	**Hierarchy Depth**	**Total Included Child Duration (s)**	**Total Excluded Child Duration (s)**	**Total Duration (s)**	**Percent of Total Cycle Time**
Local Mapping main loop	1	199.3	86.05	285.35	100
Process New Key Frame	2	0	12.18	12.18	4.268442264
Map Point Culling	2	0	0.06178	0.06178	0.021650605
Create New Map Points	2	0	113.4	113.4	**39.74066935**
Search In Neighbors	2	0	27.3	27.3	9.567198178
Local Bundle Adjustment	2	0	46.36	46.36	16.24671456

REFERENCES

[1] NVIDIA. VisionWorks. https://developer.nvidia.com/embedded/visionworks, 2015.

[2] Jurgen Sturm. Tracking and mapping in project tango. https://jsturm.de/publications/data/sturm2015_dagstuhl.pdf, 2015.

[3] LTD Hokuyo Automatic Co. UTM-30LX-EW. http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/utm-30lx-ew/.

[4] Inc. Point Grey Research. Flea3 2.0 MP Color USB3 Vision. https://www.ptgrey.com/flea3-20-mp-color-usb3-vision-e2v-ev76c5706f-3, 2016.

[5] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable SLAM system with full 3D motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

[6] Open Multiple View Geometry. Pinhole camera model. http://openmvg.readthedocs.io/en/latest/openMVG/cameras/cameras/.

[7] National Instruments. 3D imaging with NI LabVIEW. Technical report, National Instruments, August 2016.

[8] R. Mau-Artal, J. Montiel, and J. Tardos. ORB-SLAM: A versatile and accurate monocular SLAM system. In *IEEE Transactions on Robotics*, volume 31, pages 1147–1163, October 2015.

[9] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision (ECCV)*, September 2014.

[10] J. Engel, V. Koltun, and D. Cremers. Direct sparse odometry. July 2016.

[11] A. Concha, G. Loianno, V. Kumar, and J. Civera. Visual-inertial direct SLAM. In *IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden, May 2016. IEEE.

[12] J. Gui, D. Gu, and H. Hu. Robust direct visual inertial odometry via entropy-based relative pose estimation. In *IEEE International Conference on Mechatronics and Automation (ICMA)*, Beijing, China, August 2015. IEEE.

[13] H. Strasdat, J. M. M. Montiel, and A. Davison. Scale drift-aware large scale monocular SLAM. In *Proceedings of Robotics: Science and Systems*, Zaragoza, Spain, June 2010.

[14] Wilfried Elmenreich. An introduction to sensor fusion. Technical Report 47, Vienna University of Technology, Austria, November 2002.

[15] Carolyn Mathas. Sensor fusion: The basics. http://www.digikey.com/en/articles/techzone/2012/apr/sensor-fusion-the-basics, 2012.

[16] E. Jones and S. Soatto. Visual-inertial navigation, mapping and localization: A scalable real-time causal approach. In *International Journal of Robotics Research*, September 2010.

[17] J. Barrett, M. Gennert, W. Michalson, and J. Center Jr. Analyzing and modeling low-cost MEMS IMUs for use in an inertial navigation system. Worcester, MA, May 2014. Worcester Polytechnic Institute.

[18] Google. Tango. https://get.google.com/tango/, 2016.

[19] NVIDIA. NVIDIA Jetson TX1 supercomputer-on-module drives next wave of autonomous machines. https://devblogs.nvidia.com/parallelforall/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/, 2015.

[20] Point Grey Inc. Streaming point grey cameras on embedded systems. Technical Report TAN2014009, Point Grey Inc., August 2016.

[21] NVIDIA. Jetson TX1 Embedded System Developer Kit. http://www.nvidia.com/object/jetson-tx1-dev-kit.html, 2016.

[22] Inc. Point Grey Research. Flea3 1.3 MP Mono USB3 Vision. https://www.ptgrey.com/flea3-13-mp-mono-usb3-vision-vita-1300-camera, 2016.

[23] LORD Sensing Systems. 3DM-GX4-45. https://www.microstrain.com/inertial/3dm-gx4-45, 2016.

[24] Hokuyo Automatic Co. UTM-30LX-EW. http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/utm-30lx-ew/, 2016.

[25] Osterhout Group. R-6 glasses. www.osterhoutgroup.com/presskit/R-6-TechSheet.pdf, 2015.

[26] Carnegie Robotics. MultiSense SL. http://carnegierobotics.com/multisense-sl/, 2016.

[27] jacobperron. Left-handed coordinate frame? issue 80. https://github.com/raulmur/ORB_SLAM2/issues/80, 2016.

[28] Thomas00010111. Publish camera pose as tf ros message pull request 102. https://github.com/raulmur/ORB_SLAM2/pull/102, 2016.

[29] J. M. M. Montiel Raul Mur-Artal, Juan D. Tardos and Dorian Galvez-Lopez. ORB-SLAM2. https://github.com/raulmur/ORB_SLAM2l, 2015.

[30] Boost C++ Libraries. CPU Timers. http://www.boost.org/doc/libs/1_53_0/libs/timer/doc/cpu_timers.html.

[31] Chen Yun Zhi. [GPGPU 2016 final] ORB-SLAM2 GPU optimization on Jetson TX1. https://youtu.be/p77hLLRfBGQ, 2016.

[32] Pupil Labs. Platform for eye tracking and egocentric vision research. https://pupil-labs.com/pupil/, 2016.

[33] Tom Moore. robot_localization package summary. http://wiki.ros.org/robot_localization, 2016.