Vision Controlled Autonomous System (ViCAS)
An investigation of inter-robotic control through vision

Frank Bogart, James Lee

Abstract - Currently, robotic systems require one human operator per robot. The operator cost can be reduced by moving to one operator per system by enabling autonomous inter robotic control.
ViCAS is an investigation of inter robotic control by using an Unmanned Ground Vehicle (UGV) to control an Unmanned Aerial Vehicle (UAV) indoors. Existing navigation methods can be implemented by the UGV and extended to the UAV aided by computer vision tracking techniques. This tracking is achieved by use of custom stereo vision algorithms to estimate the 3D location of the UAV relative to the UGV. The UGV localizes and navigates through the room by Real-Time-Appearance-Based Simultaneous Localization and Mapping (RTAB-SLAM) and handles UAV navigation by piloting the UAV to stay within line of sight while avoiding obstacles. This inter robotic control allows a single operator to control this two robot system.

Introduction - Two robotic systems working together as one can be of vital importance when saving money on operator cost and allowing for extended missions where an individual robot would not work. Such system is the example of a system with a UAV and UGV. UGV's tend to be very heavy in comparison to UAVs. This allows them to have large payloads and impressive battery and energy sustainability. For example, NASA's Curiosity[1] rover has a radioisotopic generator which powers the robot over a long life of time. These heavy and complex energy storage devices are not typical in UAVs since they have a much smaller weight regulation. UAV flight times average around 20 minutes and UGVs can last days to months with proper generators. This difference is emphasized since UAVs typically can not land and takeoff, they are constantly in the air, spending energy to stay up when hovering. A UGV can sit still and be in a low power state without its motors draining its power reserves.

This phenomena can be taken advantage of by creating a joint system of a UGV and UAV. The UGV can provide power and recharging capabilities to the UAV if the UAV can land on the ground vehicle. This is a very challenging task for a small UAV with limited processing power to do. Large processors require larger batteries which would impose more weight on the UAV. If the processing can be done on the UGV, this can save a lot of power and weight costs for the aerial vehicle. This would take advantage of the aforementioned advantages of a ground vehicle and could be processing in a stationary state.

The trickiest part about landing on a UGV autonomously is knowing where the platform is and where you are in relation to it. This is typically done with vision systems on the UAV, however, this would cause the UAV to be too heavy and large requiring additional processing power and hence batteries. Keeping the heavy vision sensors such as cameras and LIDAR systems off of the UAV and on the UGV would solve this problem. The location of the UAV in relation to the ground vehicle could be determined by cameras on the UGV and sent to the UAV for controls. This type of external contextual input is very important for distributed robotic systems to work together.

In our system, instead of sending the location of the UAV in relation to the UGV to the UAV, we do all the thinking and processing on the UGV and send actions to the aerial vehicle. Our system is designed to be indoors and this is why we are doing it this way. If we were sending the location to the UAV, it would have to do processing on board to determine the proper actions to move and fly and that would require more processing, battery… and in the end, more weight. This is bad because the copter if too big, would not be able to be flown indoors.

A lot of quadcopters and UAVs are capable of being autonomous independently since they have processors fast enough to allow them to analyze sensor data and navigate. However, they are larger, noisier, and blow more air around to stay up. It would be unacceptable to have a quadcopter with a 2ft diameter flying inside being a danger and nuisance to people. The propellers alone could harm and cut people if there was any accidents. The only foreseeable way to have an indoor, people friendly copter is to keep it small and thus keep the processing done elsewhere.

A prime example of processing being done elsewhere is done by the GRASP Lab at the University of Pennsylvania with multi nano quadcopter control. Each quadcopter is being tracked by offboard cameras positioned around the room and processing and control is done on an external computer[2]. The cameras being used are stationary, static to the room cameras using IR light reflecting off of surfaces added to the copters. This is only useful in a professionally setup lab setting since the cameras need to be measured out and calibrated to the room. In addition, each camera costs thousands of dollars. This is not acceptable for ordinary indoor/office use of small quadcopters.

We propose to use a stereo camera mounted on a ground robot with a large computer onboard for sensor and navigation processing for both UGV and UAV. A stereo camera allows the robot to see and track the copter in 3D just like the IR motion capture cameras do. It is a much cheaper solution but requires more processing power to create the 3D scene and has to do implement complex computer vision algorithms to track the object since there is no marker placed in contrast to using the IR cameras. This requires some research and effort to create the algorithms to track the specific object (UAV) and will be slower because of the added 3D processing but can be a cheap alternative to expensive IR camera systems.

With the use of one stereo camera on the robot, multiple cameras in the room are no longer needed. The robot can move from room to room piloting and recharging the copter as needed. The UGV can pivot, navigate and maintain the copter in its sights to fly it throughout the environment.

Technical Material: A description of the development process of the ViCAS - Before starting development of any tracking or control algorithms, decisions needed to be made as to which robots to use. We wanted to focus mainly on software development instead of mechanically/electrically developing the robots so we focused on purchasing commercial off the shelf systems. We opted to use a Turtlebot from the company Willowgarage[3]. The robot is a great choice for indoor use as it is small enough to not annoy people, yet large enough to accommodate a laptop computer onboard for heavy processing. It’s differential drive system allows the turtlebot to rotate in place to keep the copter in sight without having to move forward or reverse. In addition to these features, the turtlebot comes with a Kinect sensor. The kinect allows the robot to see in 3D. It projects an infrared pattern into the scene and records how it projects using an IR camera. Considering the pattern might be too diffuse to appropriately illuminate the small copter, a stereo camera was chosen to track the copter leaving the kinect for navigation and obstacle avoidance purposes. The robot also interfaces with the ROS software framework which is our framework of choice which will be introduced shortly.

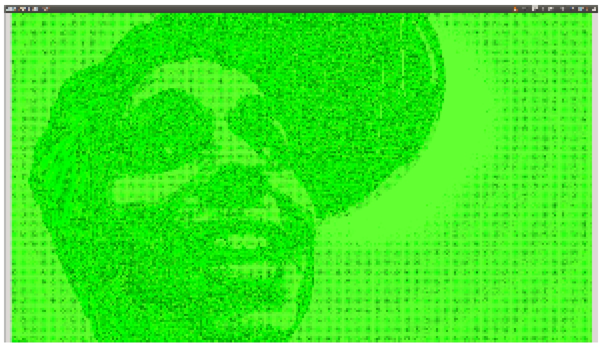
For the UAV choice, we chose to purchase a Crazyflie nano quadcopter from Bitzcraze. The copter is the size of the palm of your hand and can be flown from a computer out of the box. This allows us to connect the usb RF transmitter to the laptop on the turtlebot. The crazyflie also comes with an accelerometer and gyroscope which can send data back to the turtlebot computer. These are important when knowing the orientation of the quadcopter which is very difficult to discern by computer vision methods alone. By knowing the orientation of the copter, a command to go straight or right turn can be aligned with the turtlebot to coincide with the correct translation or rotation. Both these sensors interface with ROS as well as publishing driver commands such as roll, pitch, yaw, and thrust to pilot the copter. The copter also comes with on-chip firmware for auto stabilization to control the motors appropriately. This simplifies the control code to only rotation and translation commands instead of direct motor frequencies which could have timing and processing issues.

As mentioned previously, both systems are ROS capable which helped with communication and flexibility between turtlebot and crazyflie. ROS stands for Robot Operating System[4] and is a software communications framework. It includes a publish and subscribe topic interface with custom messages.

These messages and topics allow for any system to communicate using their message format. ROS is also comprised of nodes which sit between topics to do local processing. For example, on the turtlebot there exists a node for computer vision tracking, navigation, copter control and many more. The node previously developed for the crazyflie provides an interface with the Bitcraze crazyflie SDK to pull and publish data from the onboard sensors over the usb RF transmitter/receiver. This allows the node to be ran and do all local control processing on the turtlebot computer rather than on the crazyflie itself.

With the individual robots decided on, we were free to work on software algorithms. Up first was the tracking algorithm. This makes logical sense because the 3D location of the crazyflie must be known before any control systems code can be developed. As stated above, the sensor chosen to be used to track the copter is a stereo camera. A stereo camera is made by two cameras synchronized together with a constant known distance between them. It works very similarly to the human eye system. From each camera, features are found in the scene and attempted to be matched in the opposite camera image. From a feature pair, known as a correspondence, the 3D depth of the point can be calculated from knowing the translation between each camera.

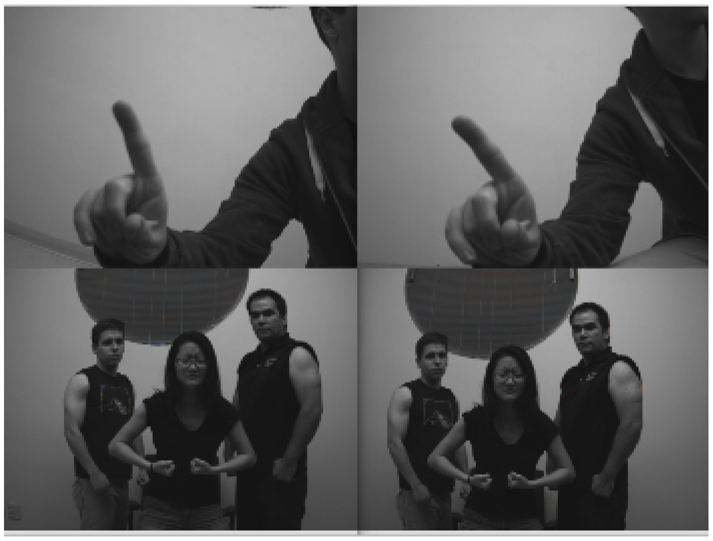
The cameras used were leopard imaging machine vision cameras. These cameras provided some setup issues with linux at first since they were designed to be used with Windows. A ROS node needed to be developed to pull images from each camera and publish them as ROS messages. This could be done with existing nodes since the cameras are UVC (universal video class) compliant, however, as discovered by speaking with the company the raw data is encoded in a GR bayer format but expressed as YUYV. Some image decoding needed to be done at the driver level first and hence using pre-written ROS nodes to speak with the camera was out of the question. A new driver was written using libv4l2 (video for linux) to pull the raw data from the cameras. With the raw data now at our fingertips, it could be correctly decoded into RGB rather than a RG-bayer pattern. A GR bayer pattern works by storing all pixel values on one layer of the image in contrast to an RGB image which has a layer for red, blue and green respectively. Hence the GR bayer pattern has a resolution much larger than the standard RGB image.


The raw image produced by the leopard imaging cameras was presented in a GR-bayer pattern. Needs to be decoded to RGB before computer vision algorithms can be applied.

An algorithm to decode the raw Gr-bayer pattern was present in a sample program used to view the cameras in linux presented to me by the company. After using their decoding method, the image was properly decoded, however, strangely in grayscale and not RGB. This was okay for me since I did not plan to use color information for tracking. It should be investigated in the future how to properly decode the pattern since the sample display program correctly showed the colors.

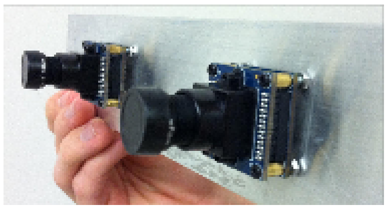
Next up when creating the stereo camera was to have both cameras be synchronized through ROS. This means that each timestamped image has the same timestamp. This can be achieved either in software via filtering or in hardware by triggering the second camera via an external strobe input. The software approach was chosen in our custom driver, but did not work to synchronize the cameras. There was noticeable lag between each camera when viewing them side by side. Instead, I ended up adapting the code to decode the GR bayer pattern into one of the existing UVC ros drivers to pull and decode the data. This was nice since the drivers in the package already had a stereo node which synchronized the cameras properly via software filtering and timestamp matching. This resulted in two images which matched in time.

Next, the two cameras needed to be rigidly connected together into a stereo rig so the distance could be calculated for triangulation. A mounting method needed to be decided. The distance between two


Top images: Time offset. The finger on the right is out of synch with the left and arrives later. The bottom images are properly synched.

cameras in a stereo configuration is known as a baseline. As the baseline changes, the 3D data's "focus" changes. For example, if you move your finger too close to your nose, you are unable to focus on the finger and your depth perception of it begins to fail. Same goes for a stereo camera. If the baseline is too large, depth information can be lost at close range. So finding a good stereo baseline is important. At first, the stereo rig was designed to have an adjustable baseline to configure to the user's choice of focus. However, this added complexity to the design and proved difficult. It also had the chance of not staying rigidly fixed. It is very important when creating a stereo rig to not have the cameras move post calibration. With a dynamic baseline setup, there runs the risk of the cameras moving slightly out of alignment and having to recalibrate both cameras. Hence, a simpler stereo rig design was proposed and implemented using a single piece of aluminum. Aluminum is strong enough to not warp or bend during normal robot movement while soft enough to drill and mold with power tools and was an obvious first choice.

The two cameras were given a baseline of 4 inches which provided enough depth of focus for 3D data (any object less than 2 feet could not be detected). We assumed the copter would be further than this distance at all times. With measurements made, holes were drilled and standoffs in place for the cameras to be attached. A hole behind each camera was also drilled to give access to the camera's USB 3.0 port.


Stereo camera rig finished with a 4inch baseline

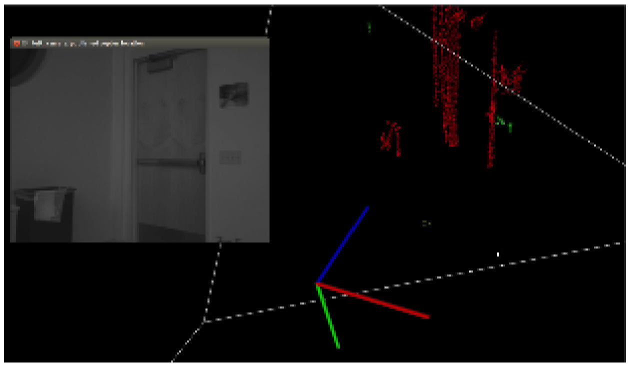
After creating the camera rig and synchronizing the cameras, camera calibration is required. Camera calibration finds the exact measurements of the baseline. More explicitly, it finds the rotation and translation required to move the right camera frame into the left camera frame. This is typically done with a checkerboard pattern held in front of both cameras by the user. Algorithms are used to find the corners of the black and white checkers and matched between the left and right frames. The translation in pixels can be found directly from the algorithm. The mapping from pixel space to real world units (meters) is given when the physical width of each checker is defined. In our case, the width of each checkerboard was .054 meters. By knowing that each checker is a perfect square, any lens distortion can be calculated and corrected to produce a rectified image. Calibration is also able to find the physical pixel widths to do this conversion as well as camera centers and focal lengths.

Once camera calibration is complete the cameras are able to produce a disparity map of the scene in front of them. This is a mapping of the distance between each pixel and the matching feature in the other camera. Together, these matching features are called a correspondence. The amount of shift from each correspondence is the disparity of each feature. This information can be displayed as a disparity image. At first, the disparity image did not seem very accurate. There was a lot of perceived noise and/or not detected objects. By modifying some parameters in the stereo matching procedure, the image could be improved.

There were a few notable parameters that really improved the disparity map. For example, the texture threshold was set to be too low initially and noise from each image was matched to noise in the other. By setting it to a value of 500, noise in the disparity image was decreased. This is because the texture (or contrast) of background noise is low compared to hard objects like a lightswitch on a white wall. The disparity of each pixel is not exactly the depth. To get the depth, each pixel goes through triangulation to be projected into the scene. After this step, a point cloud can be produced from the disparities.

A point cloud is akin to a 2D image, but in 3D. For every pixel that has a matching correspondence, the depth can be calculated and the x and y locations can be calculated from knowledge of the focal length and pixel widths of the cameras. However, for scenes which lack features (ie: a white wall), 3D data can be be inferred.

With a proper representation of the 3D scene, algorithms were written to track the copter's location. The library of choice for 3D processing was the Point Cloud Library (PCL) which has a lot of useful functions for manipulating point clouds. The first thing that is done in the tracking algorithm is to threshold based on distance to reduce the search volume for the copter. In other words, ignore any 3D points which are too close or too far away. Our system ignores all points closer than 0.8 meters since the camera's 4inch baseline can not accurately represent anything below that. Also, we do not intend to fly the copter anywhere further than 4 meters since the copter is so small and would no longer resemble a copter but rather a small dot in the scene.


The point cloud produced by the stereo camera. Red clusters are too large. Yellow bad shape, and green are good candidates.

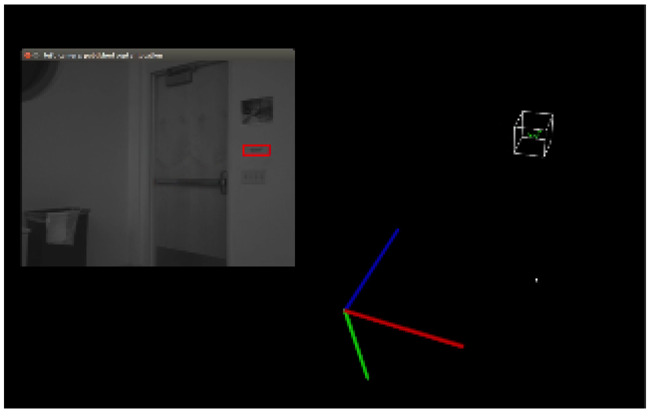
Since the point cloud coming from the stereo camera is pretty sparse (most of the 3D volume does not include points), we opted to extract clusters of points from the scene. This gave us another parameter to threshold with. If a cluster had too many or too little amount of points, then it could not be a copter. For example, at 2 meters away person typically has around 500 points associated with their body while the crazyflie has around 40. This gave a pretty good indication of what was a good copter candidates, but not good enough. For example, if a person was wearing a white shirt (which lacked features), then instead of 500 points, they could contain around 250. The same could be said for chairs and other objects. We needed another metric besides just the number of points. We decided to make a crude calculation of perceived surface area. This is the surface area of the object as if it were a flattened 3D object rather than a true object having grooves and warped textures which would increase the actual surface area. This calculation allowed us to threshold out larger clusters which passed the point requirement, but were still obviously too large like a textureless door or empty picture frame. Our copter is roughly .015 square meters and anything above or below that amount by a threshold was determined to be a bad candidate and was not included.

We could narrow the search space further by eliminating clusters that did not appear to be the shape of the copter. At any given shot of the copter, the width of it was at max 4.4 times larger than its height. In other words, it was closer to being a square than say a lamp. This allowed us to remove clusters which were too long or skinny to be the copter.

Any candidate which met the cluster size, surface area, and shape requirements was determined to be a good copter candidate. A final check is to make sure it "looks" like a copter. To do this, we created a mathematical model of the copter and compared any given frame of it to the model. To do this, we first needed images of each candidate cluster. Through ROS, the tracking algorithm subscribed to both the point cloud and image topics of the stereo camera. With each new message, a synchronized point cloud and camera image was given. From each good cluster candidate, a 3D bounding rectangle was computed and projected into the 2D image. This gave us a bounding box around where candidate cluster's 2D image location was. The projection was done by the camera's camera calibration matrix that was determined during camera calibration. This is essentially the reverse of creating the 3D points. Instead of re-projecting each pixel into the scene using the disparity/depth, the 3D points are projected into the 2D image.

From the 2D image location of each candidate a model of the copter was trained using positive (images of the copter) and negative (images of candidates not being the copter) images of the copter. The model creation process was leveraged by the OpenCV (open computer vision) library. The model used is called a Classifier, or more precisely a Cascade Classifier During the training process, stages are created modeling the features of the copter. If a candidate is compared against the model, it first is compared against the first stage, then second, third, etc. It must pass all stages to be considered a copter, otherwise it is rejected along the stages. Training was initially done on 1500 positive and 1500 negative images of the copter. The training data produced a good classifier, however the images were taken at a variety of ranges with some taken beyond the 4meter limit. This caused the classifier to be skewed too far towards being a small copter (just a dot). This would make any small generally spherical dot in the scene be mistaken for a copter. The classifier was retrained using images of the copter close to the camera with the home of extracting more features from the copter images. A similar model was created to the previous classifier without the small black dot false positive phenomena. This was done with only 600 positive and 900 negative images.

If the classifier determines that the candidate cluster is the copter, than the algorithm publishes the 3D center of the cluster over the nodal network to the control code. Once the algorithm determines that the copter has been found, a region of interest (ROI) is placed around the previous copter location. This ROI reduces the search volume for the next iteration. Since the pointcloud/image data is coming in at a fast enough rate, it can be assumed that the copter has not moved too much from its previous location. Hence, the algorithm searches in the nearby area. This allows the algorithm to process less 3D data with less errors and faster time. When the copter is not found (classifier rejects all candidates), then the 3D ROI expands by a constant amount along all dimensions. This accounts for any direction the copter may have moved while it was not in sight.


Copter found: Region of interest (ROI) placed around previous copter location. When copter is not found, the region expands incrementally.

Once the 3D center of the copter has been sent to the control code, a PID loop is used for control system. A target is some 3D point in space where the quadcopter is desired to be at relative to the turtlebot. An error is calculated between the target and the 3D center of the copter and used in the PID. There are separate PIDs for each axis of the flight dynamics. Also extra PID loop is added for thrust. The controller can make a judgment on how to adjust the appropriate thrust commands to perform a desired pattern. A little bit different parameters were used in the PID loop for the yaw. Instead of the 3D points, a specific yaw value was the target and the input was the current yaw value received from the IMU on the quadcopter.When testing the PID codes, a problem during take-offs was noticed. During the take-off, more thrust was needed to lift off from the ground. However with this excess amount of thrust, the quadcopter moved in the frame of the stereocam too fast for the tracking algorithm to make any adjustments with the PID. Hence, we investigated if a higher frequency tracking algorithm would improve the control.

The second tracking algorithm implemented makes use of the fact that when launching the copter, the turtlebot will not be moving. When the cameras are not moving, background subtraction can be utilized to find any movement in the scene. This functionality is only processed on a single camera's image feed. Over time as frames come in, a background is defined by subtracting subsequent frames from a model to see how they have changed over time. Once a big change has occurred, ie: a person walking into the frame, then the change is labeled as being foreground until there is minimal change and hence fades into the background. These small changes are identified as potential copter candidates and are again compared to the model of the copter in the cascade classifier. If the copter is found in the foreground of the image, then its 3D center must be published over the network. Since background subtraction is typically done on a single image stream from one camera, there is no way to get the 3D center from the data alone.

To solve this, the algorithm subscribes to both the left camera image stream and the point clouds. Once the copter has been found, since the point cloud is organized the same way that the images are (640x480 data blocks), it is easy for each pixel to be looked up in the point cloud. The copter cluster is re-created this way and the copter center is published over the network. This algorithm results in an average of 18Hz, double the speed of the clustering algorithm. In theory, once the copter is hovering within a defined region of space, the algorithm can be switched back to the clustering algorithm since the velocity of the copter would be stable enough for the lower frequency tracking algorithm to control it. In other words, on copter takeoff, the turtlebot would use fast background subtraction to track the copter and once under control switch over to clustering to allow the turtlebot to move forward throughout the room.

Control results with background subtraction definitely updated quicker. The results of the control system was much more clear with the faster tracking algorithm. After seeing faster response of the control system, another check was added in the PID loop for thrust. The velocity of the quadcopter was taken into consideration in the control system. Before, only position of the quadcopter affected the PID correction. However, a check for the velocity of the quadcopter was added to make the PID more fine tuned. The velocity of the quadcopter was calculated using the IMU on the quadcopter. The IMU gave back data about its vertical acceleration and the acceleration was summed over time to get the velocity. With this additional data, checks for different cases depending on which way the quadcopter was traveling and the which way the PID correction was trying to make the quadcopter move were added and changed the thrust accordingly. For example, if the quadcopter was already moving and the correction was trying to make it move even higher faster, the adjustment to the thrust would not as big as the adjustment applied to thrust if the quadcopter was moving down. Adding this check in the control system reduced the overshooting of the PID.

Work was also done to enable to turtlebot to move within the room while tracking the copter with the goal of waypoint navigation. Waypoint navigation would allow an operator to tell the system where to move in the room instead of each individual robot. This implies allowing the turtlebot to localize itself within the room as well as the crazyflie. If the turtlebot knows where it is within a map of the room and it knows where the crazyflie is in relation to itself, then in turn it can place the crazyflie in the map as well.

To allow the turtlebot to localize itself within the room the process of SLAM (Simultaneous Localization and Mapping) was implemented. Instead of writing this process ourselves, we chose to utilize an existing ROS package. After some research on SLAM packages, the RTAB-MAP package[5] proved the most promising. This stands for real time appearance based mapping. It is an RGBD-SLAM approach which uses color camera information in addition to depth which seemed a perfect match for the turtlebot's kinect sensor.

SLAM works by first understanding the robot's motion (odometry) and relating its movement (creating a path) by comparing it against a series of landmarks (robust features) within the room. It does this by a process called bundle adjustment which takes in a series of image frames, finds the features, and attempts to match them within a given global model of the room. Once a bundle is complete, the robot updates its position within its generated map given that the landmarks are stationary and not moving. The robot's odometry can be determined by a variety of methods. In our case, we relied on the turtlebot's wheel encoders to give us an estimate of its odometry. Typically, for best results, the odometry is determined by a combination of sensors: wheel encoders, gyroscope, and cameras (visual odometry matching features over time). However, the turtlebot's gyro was disconnected was never re-connected due to time constraints since it would have to be soldered and integrated into the software driver system.

To test the effectiveness of RTAB-MAP, it was first tested by teleoperating the robot through the environment to create a map. Without the use of gyroscope information for orientation, the map appeared skewed because of wheel encoder drift. We attempted to fuse visual odometry to correct, however, our visual odometry had inherent drift as well. The SLAM investigation was dropped to focus on improving visual tracking and copter control.

Milestones - The project was distributed into three major milestones 1) tracking the copter, 2) autonomously piloting the copter, and 3) following and piloting the copter. These milestones were introduced at the beginning of the quarter. Development began in parallel with milestone 1) being done by Frank and milestones 2) being done by James. While the stereo camera was being built and tracking algorithm developed, the copter was being built and communication established. By week 5, a tracking algorithm was finished well enough to track the copter with high accuracy behind a white wall. At this point, the copter was almost ready for control code to be developed. In the mean time, the tracking algorithm was refined with a lot of time spent training cascade classifiers. About a week was dedicated to creating the first classifier alone. Once a proper tracking algorithm was produced, waypoint navigation and SLAM was investigated for the turtlebot.

Roadblocks appeared during this phase. It was difficult to install RTAB-MAP since it was conflicting with the version of PCL which I was using. After joining the forums which the algorithm was maintained on and getting in contact with the author, I found a workaround by obtaining the binaries and forcing the library to link with the ROS version of PCL. A second roadblock occurred when trying to get a good map from RTAB-MAP using the turtlebot. The map was poor because I could not properly calibrate the wheel encoders. The calibration procedure of the turtlebot depends on knowing a reference angular displacement which is given by the gyroscope. However, the gyro was disconnected and could not reconnect it to the turtlebot. The gyro also contributes to the odometry and would have drastically improved the map produced by RTAB-MAP.

In addition to roadblocks introduced by SLAM and navigation, ROS threw a big monkey wrench into the thick of things. During a normal Ubuntu update, a lot of ROS packages were updated. Before the update, the first tracking algorithm (clustering) would publish the copter center, when locked on, at around 24Hz. After the update, it fell to 5Hz. After adding some timing print statements throughout the code, it was determined that the slowing was happening in a ROS to PCL conversion function for point clouds. After a week of posting the problem on the ROS forums with no replies, I decided to open up the source code for the conversion to understand it. I was able to tweak some numbers and force the conversion to happen a certain way which sped up the algorithm to 15Hz, but not back to the 24Hz original speed.

Of the three milestones that we set out to accomplish, we were able to fully complete the tracking algorithm and partially finished controlling and system navigation. The control code was incomplete due to the unexpected and inconsistent behavior to the changes made in the thrust of the quadcopter. Same behavior from the quadcopter was expected in the beginning given some thrust value. However throughout the project, it was noticed that the result of a certain thrust value depended on various factors like the life of battery and the breeze in the air. Without any consistency in the thrust which was heavily used in the control system, a complete control code could not be finished.The navigation milestone was abandoned because it required the control systems of the turtlebot/crazyflie to be completed. It would be impossible to move both the turtlebot and crazyflie through the room without the turtlebot being able to keep the crazyflie in front of itself. Efforts to navigate the room were transitioned to efforts to controlling the copter and increasing the frequency of the tracking algorithm due to the ROS update induced problems.

Conclusion - With a system like ViCAS in place, a single operator could operate the joint ground and aerial system. The system could be extended to multiple aerial vehicles with multiple cameras and multiple recharge stations on the vehicle. For this to take place, the operator interacts with the base system, in this case, the ground vehicle since it can support high computing power and telecommunication systems.

The ViCAS system contributed to this effort by developing tracking algorithms using low cost sensors, ie: stereo cameras, compared to active infrared sensors. Stereo cameras can be mounted on a moving robot and are not required to be stationary in fixed positions. A rudimentary method of control using PID loops was established to keep the copter in the frame of the camera with mild success. The copter is able to stay in the frame for a few seconds but drifts away and tracking is lost. Possible future work might include improving the control system code using alternative control method to take into account the copter's velocity and flight dynamics. The gyroscope of the turtlebot could be attached and re-integrated into the ROS software system of the turtlebot. This would allow for proper odometry calibration which could improve the use of RTAB-MAP based SLAM. From this generated map, the copter could be placed within the map in relation to the turtlebot. A planar laserscan approach using the map data could be implemented to incorporate obstacle avoidance and local cost maps for the copter.

The ViCAS team was able to create a Vision Controlled Autonomous System. We were not able to control it to the degree we originally anticipated, but with a robust tracking algorithm and a two way communication system established between UAV and UGV, a first phase control system was setup between the two.

References:

1. http://en.wikipedia.org/wiki/Curiosity_%28rover%29
2. https://youtu.be/w2itwFJCgFQ?t=78
3. https://www.willowgarage.com/turtlebot
4. http://www.ros.org/about-ros/
5. http://wiki.ros.org/rtabmap_ros