Design of an Autonomous Aerial Herding Robot for A GPS Denied Environment

Kevin J. Sheridan
Purdue University, School of Aeronautics and Astronautics

Jason K.C. Lo
Purdue University, School of Aeronautics and Astronautics

Mitchell J. Lee
Purdue University, Polytechnic Institute, Department of Computer Information Technology

In this paper, we present our solution to the seventh mission of the International Aerial Robotics Competition. Our system is computer vision based with assistance of an inertial measurement unit and range finder allowing it to operate in a GPS denied environment. Our system is capable of herding target robots via blocking. To accomplish this, a visual inertial odometry algorithm was developed allowing our system to quickly and robustly estimate its body frame velocities. We also describe a method of globally localizing our system and ground robots. Further, we greedily evaluate a uncertainty aware Markov Decision Process to decide which target to herd at a given state. Finally, A minimum time actuator constrained trajectory generation algorithm was implemented to execute herding maneuvers.

Nomenclature

P = Covariance Matrix
$\boldsymbol{x}$ = State vector
T = 3-D Transformation Matrix
K = Intrinsic Matrix of a Camera
$\mathfrak{g}$ = Lie algebra of se(3)
$\boldsymbol{u}$ = normalized pixel position
$\mathbf{r}$ = position vector
$\mathbf{X}$ = Sigma point

I. Introduction

"The Association for Unmanned Vehicle Systems International Foundation's International Aerial Robotics Competition is the longest running collegiate aerial robotics challenge in the world, having celebrated its quarter century anniversary in 2016. The primary purpose of the IARC is to move the state-of-the-art in aerial robotics forward through the creation of significant and useful mission challenges that are considered 'impossible' at the time that they are proposed" [1].

Mission 7

The objective of Mission 7 of the International Aerial Robotics Competition (IARC) is centered around autonomously herding 10 autonomous ground robots via touch or block in a 20m x 20m GPS denied environment without the help of external sensors or hardware. Additionally, 4 obstacles robots are added which carry vertical poles no greater than 2 meters tall.

As shown in figure 1 the ground robots also known as target robots are initially arranged in a 1 meter circle and the obstacle robots are initially arranged in a 5 meter circle. Each of the target robots run the same exact program which is based on an internal timer inside each of the target robots. The rules which each of the target robots are as follows:

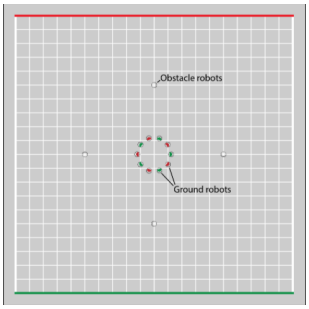
1) move in a forward at 0.33 m/s.
2) every 5 seconds $+/-20°$ of gaussian noise is added to the angle of the target robot.
3) every 20 seconds the target robot rotates 180° clockwise.
4) upon a collision from the front, the target robot rotates 180° clockwise.
5) upon a collision from the top, the target robot rotates 45° clockwise.

Similarly, the obstacle robots each follow the same program which is defined as follows:

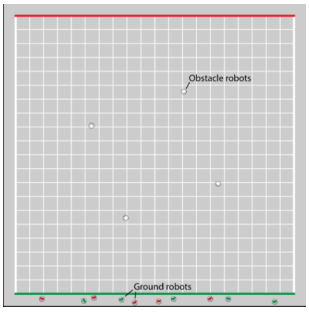
1) move along a 5 meter radius circle clockwise with a tangential velocity 0.33 m/s.
2) upon a collision from the front, stop for 2 seconds.

The goal of an aerial robot is to herd each of the 10 target robots over the green boundary of the field via touching or blocking. When a target robot leaves the 20m x 20m grid it is permanently removed from the field. This must be accomplished in under 10 minutes autonomously.

(a) Actual Competition Floor with Robots



(b) Starting State of Mission 7



(c) Ideal Ending State of Mission 7

Fig. 1 Starting state and ideal ending state images courtesy of Team Ascend NTNU [2]

II. Solution

A. Airframe and Propulsion

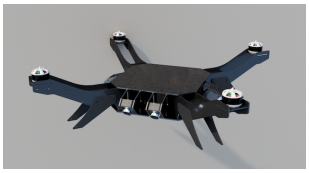


Fig. 2 CAD Airframe.

We chose to use a standard quadrotor design for our airframe. However, it was necessary to manufacture our own airframe because our electronics require more room than off the shelf airframes provide.

Our airframe was constructed out of 3mm carbon fiber plates, and Electronics are mounted with 3D printed brackets. The arms of our frame are swept up allowing the field of view of our cameras to be nearly unobstructed.

Our propeller-motor combination allows our robot to produce 25N per motor at full throttle. Our airframe is approximately 4.5kg when fully assembled.

B. Computation

We chose to use an Nvidia Jetson TX2 with a small form factor carrier board as our main computer. We chose the Jetson TX2 because it has a quad core A57 ARM Processor, dual core Denver Processor, and a 256 CUDA core GPU. This embedded computer is unparalleled because of its 256 CUDA cores which allow for massive parallelization.

We chose to use a PixHawk flight controller for its redundant EKF and stable code base. The PX4 firmware, which is flashed to the PixHawk, has many safety features and support for all of our external control inputs and state estimates.

C. Sensing

We use 4 30FPS PointGrey Chameleon3 cameras and 1 150FPS PointGrey Chameleon3 camera for all of our computer vision algorithms. We also use a LidarLite V1 as a range finder for height estimation. Finally, we use an IMU built in to the PixHawk flight controller with a gyroscope and accelerometer.

D. Safety

The IARC mission 7 requires every team to have a kill switch for their robot. This kill switch must be an independent system from the main computer and shutoff the motor power once triggered. The team decided that the best idea would be to cut off the power to the electronic speed controllers(ESCs). The circuit designed uses high power relays to switch the power on and off.

To achieve this, we wrote a program for a microcontroller that interprets the pulse width modulation(PWM) signal coming from our RC Controller and switches the relay through a MOSEFT.

Each individual component was tested and worked as expected. The entire layout of the power system was planned and put together.

E. Visual Inertial Odometry

The most challenging part of operating in a GPS denied environment autonomously is localizing robustly and efficiently. A good starting point would be to somehow estimate motion using computer vision. This is called Visual Odometry or Visual SLAM. Further when this is done with a single camera it is called Monocular Visual Odometry/SLAM.

1. Prior Work

True Monocular Visual Odometry is still an unsolved problem because of its lack of robustness and need for special initialization procedures. By adding inertial measurements, Monocular Visual Inertial Odometry has been shown to be capable of "power up and go" initialization by the creators of ROVIO [3]. ROVIO is based on a tightly coupled iterated EKF. It is well known and shown in their paper that the computational effort of an iEKF scales poorly with state size. Due to the nature of a tightly coupled method, ROVIO requires that the position of each feature be contained within its state estimate. This method requires too much compute time for our application. However, the robustness of this method is desired.

A semi-direct method (SVO) was introduced in 2014 [4]. In this method, speeds of only 3ms per frame were achieved with good accuracy. However, this method was not inertial fused giving poor robustness.

We seek a monocular method which gives both high speed and high robustness. Our method is based on the work of Forster [4] with the addition of a *tangent space EKF* and probabilistically weighted least squares pose optimizer.

2. Tangent Space EKF

We developed a method of efficiently fusing IMU measurements into a monocular visual odometry algorithm similar to SVO. Fusing IMU measurements into a monocular visual odometry algorithm is a very effective way of increasing its robustness. Further, it is common for an IMU to be rigidly mounted to an image sensor. Smart phones are a good example of this.

Further, our method adds an efficient way of removing the ambiguity between a rotation and translation while minimizing reprojection error.

Reprojection error is the classical method of pose op-

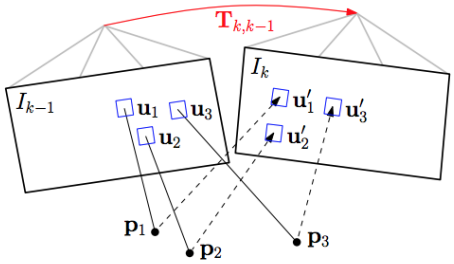


Fig. 3 The pose of our camera is optimized by minimizing an error called *reprojection error*. Reprojection error is defined as the error of feature positions and projected feature positions 6. Image courtesy of Forster [4]

timization and is called an *indirect method* [4] because it operates on feature positions rather than pixel intensities. Methods that operate on pixel intensities directly have shown better performance in certain scenarios when compared to indirect methods. However, due to time constraints our method is indirect.

$$K = \begin{bmatrix} f_x & \lambda & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \tag{2}$$

$$\mathbf{r} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3}$$

$$\mathbf{u} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{4}$$

$$s \begin{bmatrix} \mathbf{u} \\ 1 \end{bmatrix} = KT \begin{bmatrix} \mathbf{r} \\ 1 \end{bmatrix} \tag{5}$$

Equation 5 is called the pinhole model of a camera [5]. This is the model describes how a camera sees features in the environment. The K matrix defined in equation 1 is calibrated for each camera-lense combination and maps a feature position in a metric coordinates to pixel coordinates. T is a transformation matrix representing the inverse transform of the camera position. $\mathbf{u}$ is the homogenous pixel position of the feature. $\mathbf{r}$ is the metric position of the feature. s is effectively the depth of the feature in the camera frame.

Using the pinhole model of a camera we can define an error function for n features with n pixel measurements.

$$E(T, \mathbf{r}_1, ..., \mathbf{r}_n) = \sum_{i=1}^{n} \|[\mathbf{u}_i - \Pi(KT \begin{bmatrix} \mathbf{r}_i \\ 1 \end{bmatrix})]\| \tag{6}$$

$$\Pi(\begin{bmatrix} x \\ y \\ z \end{bmatrix}) = \begin{bmatrix} x/z \\ y/z \end{bmatrix} \tag{7}$$

Using equation 6 we now have an error function to minimize with a weighted least squares optimization method.

We know that the transformation matrix, T, has only 6 degrees of freedom. However, the matrix itself has 12 degrees of freedom with constraints on the rotation part.

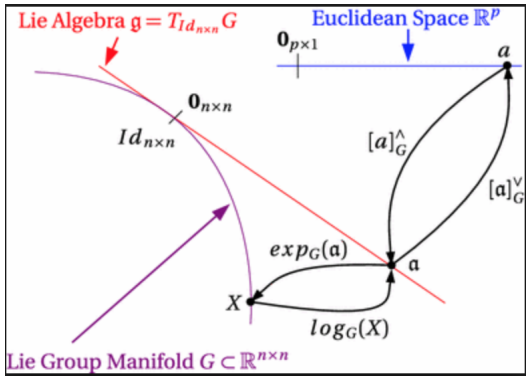


Fig. 4 The Lie Group SE(3) is a group which describes all 3D transformation matrices. This group can be described in its minimal form using its corresponding lie algebra, $\mathfrak{g} \in \mathbb{R}^6$. The lie algebra are part of a linear space which is called the tangent space. This space is centered around the unit SE(3) transform [6]. Lie Algebra can be transformed to and from the Lie Group with a exponential and logarithm map. Figure courtesy of [7].

$$\mathfrak{g} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} \tag{8}$$

$$\hat{\mathfrak{g}} = \begin{bmatrix} 0 & -\omega_z & \omega_y & v_x \\ \omega_z & 0 & -\omega_x & v_y \\ -\omega_y & \omega_x & 0 & v_z \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{9}$$

$$T = exp(\mathfrak{g}) = \sum_{n=0}^{\infty} \frac{\hat{\mathfrak{g}}^n}{n!} = I + \hat{\mathfrak{g}} + \frac{\hat{\mathfrak{g}}^2}{2} ... \tag{10}$$

The exponential map from a Lie Algebra to a Lie Group is shown in equation 10.

$$exp(A_{adj} \cdot \mathfrak{g}) = T \cdot exp(\mathfrak{g}) \cdot T^{-1} \Rightarrow A_{adj} \cdot \mathfrak{g} = log(T \cdot exp(\mathfrak{g}) \cdot T^{-1}) \tag{11}$$

Equation 11 gives a *linear and perfect* transformation from a lie algebra in one tangent space to a lie algebra in a tangent space transformed by T. Using this knowledge, we can transform the covariance matrix representing uncertainty in one tangent space into another.

Our Tangent Space EKF is a modification to the EKF which represents the pose uncertainty in the tangent space of the current pose estimate at all times. After any transformation of the state, the state space is transformed to the new tangent space with the inverse Adjoint map.

This new method allows us to exploit the fact that a small transformation can be treated as a simple addition to the prior tangent space. This allows for much more efficient pose optimization and uncertainty propagation.

$$P' = A_{adj}(-\mathfrak{g})FPF^T A_{adj}(-\mathfrak{g})^T \tag{12}$$

Equation 12 is the new uncertainty propagation equation for the tangent space EKF. F is the linear process applied to the tangent space. $A_{adj}(-\mathfrak{g})$ is the adjoint map of the inverse of the small transformation applied to the tangent space.

Equation 12 allows us to keep track of the instantaneous *twist* uncertainty. This is very helpful during the optimization step because we can use this new uncertainty prior directly as a weight during pose optimization.

$$\mathbf{x} = \begin{bmatrix} \mathfrak{g} \\ v_{body} \\ \omega_{body} \\ a_{body} \\ biases_{gyro} \\ biases_{accelerometer} \\ \lambda \end{bmatrix} \tag{13}$$

The state used in our method is shown in 13. v_{body} is the body frame linear velocity. ω_{body} is the body frame angular velocity. a_{body} is the body frame linear acceleration. λ is a metric scale factor used to scale accelerations to m/s^2. A metric scale factor is needed because monocular visual odometry has no way of knowing the scale of the world.

$$x_{t|t-\Delta t} = \begin{bmatrix} \mathfrak{g} + \begin{bmatrix} \omega_{body} \\ v_{body} \end{bmatrix} \Delta t + 0.5 \begin{bmatrix} \mathbf{0} \\ a_{body} \end{bmatrix} \\ v_{body} + a_{body}\Delta t \\ \omega_{body} \\ a_{body} \\ biases_{gyro} \\ biases_{accelerometer} \\ \lambda \end{bmatrix} \tag{14}$$

Notice that the process defined in equation 14 is linear. Note that tis is only true for a small Δt and around the tangent space.

3. Weighted Least Squares Pose Optimization

$$\Delta\mathfrak{g} = [P^{-1}H^T R^{-1}H]^{-1}P^{-1}H^T R^{-1}\mathbf{e} \tag{15}$$

Equation 15 shows the pose optimization iteration. H is the jacobian which maps a small twist into a small pixel delta. R is the covariance matrix representing uncertainty in the pixel position measurement. $\mathbf{e}$ is the reprojection error vector. P represents the prior tangent space uncertainty.

After each iteration the uncertainty should be transformed to the new tangent space. H can be computed very efficiently by exploiting the fact that the state is at the unit transform at all times.

4. Results

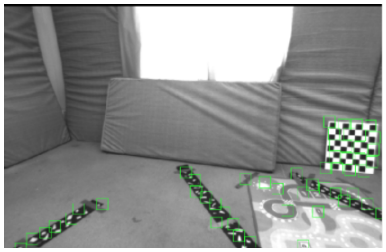


Fig. 5 This image shows the features being tracked by our Visual Inertial Odometry Algorithm on the EuRoC Vicon Room dataset [8].

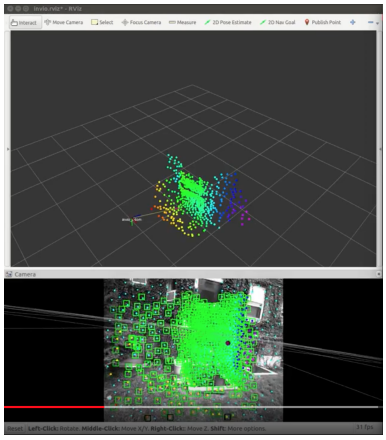


Fig. 6 Monocular Visual Odometry on our camera. The colored points are the 3D positions of features.

In the future, we will compare our odometry estimates to ground truth and determine exactly how much this method drifts.

F. Obstacle Detection

Another challenge that must be overcome in Mission 7 is avoiding obstacle robots. To reiterate, the obstacle robots have vertical heights up to 2 meters and are initially located at a 5-meter radius, 90 degrees apart from each other. These robots move in a clockwise circular motion with the fixed radius of 5-meter until collision with another object, which causes them to pause momentarily.

The fundamental challenge lies in depth estimation. Our current solution to this problem is to obtain a 3D point cloud using a pair of stereo cameras, extract clusters that represent individual objects, and compute the centroid of

each cluster. These centroids will be used in the state estimator to estimate the pose of obstacle robots. We must estimate the pose of obstacle robots so that we can accurately predict the future collisions of target robots and plan trajectories.

A dataset from ETH Zurich's EuRoC MAV dataset [8] was used to aid testing of the centroid computing function (referred to as PCProc below). A dataset fixer program was created to calibrate the coordinate frames of the stereo cameras and vicon motion capture frame. This involved calculating the transformations between different reference frames within the quadrotor used to collect the dataset. Some minor challenges were encountered during calibration since Euler angle sequence used in the dataset was not known so trial-and-error was the only approach to solving the problem. A visualization of the point cloud from the dataset was obtained with the help of this dataset fixer and is shown below.

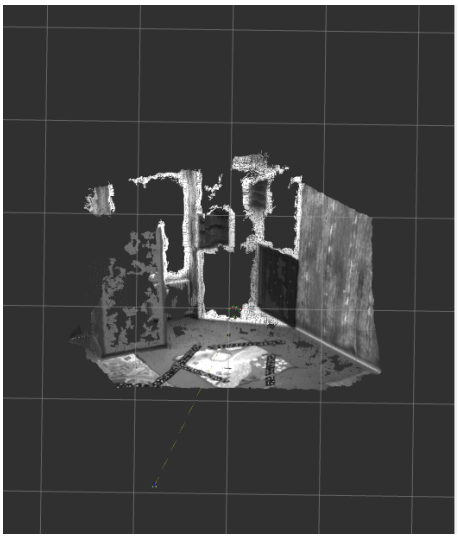


Fig. 7 Point cloud generated with the EuRoC MAV dataset.

PCProc is a function that takes in a point cloud, extracts clusters, computes and publishes the centroids of each cluster. After researching clustering algorithms, We found that there is an algorithm called Euclidean Cluster Extraction in the point Cloud library [9] that extracts clusters by performing a breath-first search on each point with a constraint on the maximum distance between nearest neighbors. The maximum distance is a threshold that determines how far two points must be for them to be considered as points from the same cluster. Note that a K-d tree was used to assist finding the nearest neighbors of a point for performance reasons. A detailed algorithm is shown below.

Finally, centroids are obtained by taking the mean of each cluster's pixel coordinates. The centroids are then published to other programs in our system for use in state estimation and trajectory generation.

Algorithm 1 Algorithm to extract clusters

```
Create an Octree (3-d tree) representation for the raw point cloud P;
Create a queue of points that needs to be check, Q , and an array of clusters C;
for every point p in P do
    Add p to Q;
    for every point p in Q do
        Find nearest neighbors pn
        if pn-p less than threshold radius of max d and not visited then
            Add pn to Q
        end if
    end for
    Add Q to cluster array C
end for
when all points in P has been visited, the algorithm is completed and all clusters has been extracted
```

G. Target Detection

1. Overview

Similar the rest of our system, our system uses computer vision to detect target robots. Each target robot will have either a red or green plate on it. These plates are the key features that our system uses to identify target robots. We are using a blob detection algorithm to identify clusters of red and green pixels. Once these pixel clusters are identified, they are then filtered and the pixel position of the centroids for each cluster are extracted.

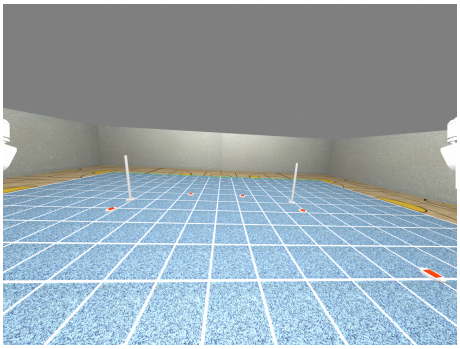
If we make the assumption that the z height of each target is constant, we can use these pixel positions as a measurement update of target robot state.

2. Current Progress

Currently, the blob detection algorithm can generally detect green blobs and some red blobs. Detecting red blobs is slightly more complicated than detecting green blobs as red colors occur at two distinct ranges within Hue Saturation Value (HSV) spectrum. This means that the program has to detect scan for two different color ranges when looking for red blobs. Currently, the program can only detect colors within the upper HSV range because the dataset we tested on only had red colors within that range.

3. Issues

The blob detection algorithm needs to be further tuned. The algorithm has problems with falsely identifying stray pixels as clusters. In past implementations however, the



(a) Footage of our aerial robot's perspective from out dataset

(b) A filtered image showing red blobs as white spots

Fig. 8 Our blob detection algorithm detecting red blobs

algorithm was only able to detect a few of the clusters that in the frame.

4. Future Work

We need to plan to further tune our algorithm by changing different tuning parameters and experimenting with different filters

H. UKF State Estimator

1. Motivation

In order to choose the optimal target robot to pursue at a given time, our robot must be able to predict the future states probability distribution of all the robots on the field. A common solution to this problem would be to implement the Extended Kalman Filter (EKF) to estimate the state of the ground robots undergoing their non-linear process. However, the robots are programmed rotate or stop at set time intervals or upon collision. This makes the EKF an unsuitable solution as the process is now discontinuous. The EKF requires that the process be differentiable in order to linearize it at a given point [10]. Therefore, we decided to employ another type of Kalman filter known as the Unscented Kalman Filter (UKF). The UKF works by sampling the distribution, P, along its eigen vectors around the current state estimate $\boldsymbol{x}$.Each one of these samples are known as *sigma points* denoted as $\boldsymbol{X}_i$. Each of these sigma points are then passed through the discontinuous process. Subsequently, a gaussian is then fitted to this new distribution. This process is known as an Unscented Transform (UT) [11]. Because the UKF uses sampling to propagate uncertainty rather than linearization, it is able to handle high non-linear processes.

2. Unscented Transform

The first part of the UKF uses the UT to propagate uncertainty. First, we initialize our tuning parameters. These tuning parameters allow us to control how far the sampled sigma points are from our mean. n represents the number of dimensions within our state, $\boldsymbol{x}$. We initialize α, β,κ and λ such that:

$$\kappa \geq 0 \tag{16}$$

$$\alpha \in (0, 1] \tag{17}$$

$$\beta = 2 \tag{18}$$

$$\lambda = \alpha^2(n + \kappa) - n \tag{19}$$

From there, we can calculate our sigma points $\boldsymbol{X}_i$ using our current state estimate, $\boldsymbol{x}$, and probability distribution, P. To do this, we define a matrix, M, shown in equation 20.

$$M = \sqrt{(n + \lambda)P} \tag{20}$$

Our initial sigma point $\boldsymbol{X}_0$ will always be our current state $\boldsymbol{x}$. We then take our M and subtract as well as add each row from $\boldsymbol{x}$[12]. We should end up with $2n$-1 sigma points:

$$\boldsymbol{X_0} = x \tag{21}$$

$$\boldsymbol{X_i} = x + M_i \qquad for\ i = 1, ..., n \tag{22}$$

$$\boldsymbol{X_i} = x - M_{i-n} \qquad for\ i = n + 1, ..., 2n \tag{23}$$

M_i is the ith row of the matrix, M. We also need to calculate weights for the sigma points. We generate two types of weights, mean weights $w_m^{[i]}$ and covariance weights $w_c^{[i]}$.

$$w_m^0 = \frac{\lambda}{n + \lambda} \tag{24}$$

$$w_c^0 = x + M_i \tag{25}$$

$$w_m^i = w_c^i x = \frac{1}{2n + \lambda} \qquad for\ i = 1, ..., 2n \tag{26}$$

Transform each of our sigma points $\boldsymbol{X}^i$ as well as $\boldsymbol{x}$ through our process $g(x)$:

$$g(\boldsymbol{X_i}) \quad for\ i = 0, ..., 2n \tag{27}$$

$$g((x)) \tag{28}$$

Using the transformed sigma points we can now propagate the error. This is first done by generating a mean from the sigma points called $\boldsymbol{x'}$. We will use $\boldsymbol{x'}$ to calculate P'.

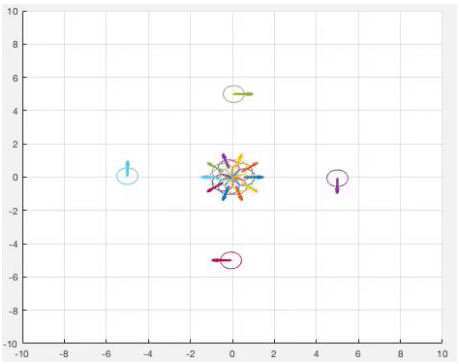
$$\boldsymbol{x'} = \sum_{i=0}^{2n} w_m^{[i]} g(\boldsymbol{X}^i) \tag{29}$$

$$P' = \sum^{2n} i = 0 w_c^{[i]} (g(\boldsymbol{X_i}) - \boldsymbol{x'})(g(\boldsymbol{X_i}) - \boldsymbol{x'})^T \tag{30}$$

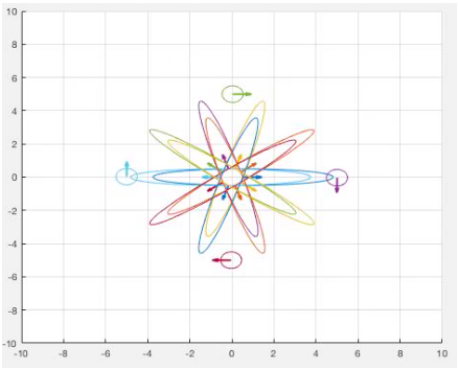
After the UT, a standard Kalman Update can be applied to the process estimate to produce the optimal estimate.

3. Results

Our program can accurately model the uncertainty associated with each robot's state on the field until a discontinuity is reached.

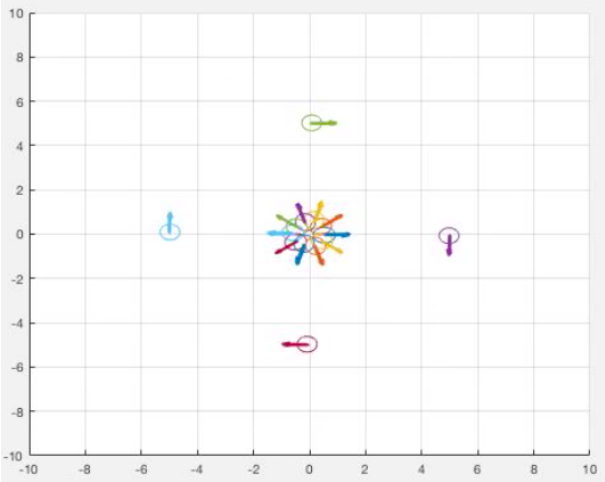


(a) Robots at time 0 with Unscented Transform only

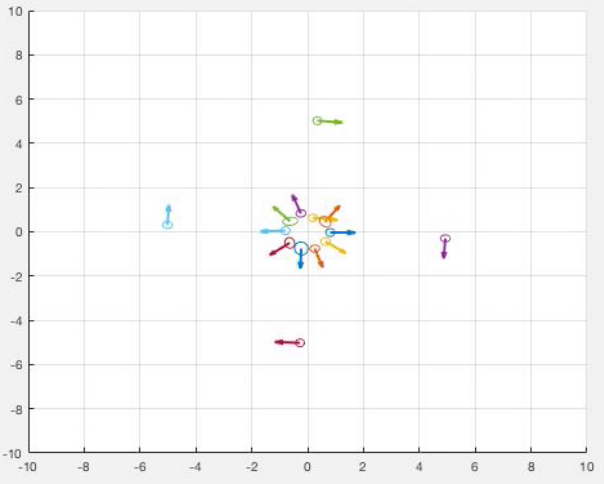


(b) Robots at time 0.1 with Unscented Transform only

Fig. 9 The probability distributions for each robot grow very quickly and so we quickly become uncertain of the robot's position. This is most likely due to the high timer internal timer uncertainties that each target robot is initialized with. About half the sigma points for each corresponding robot are initialized with an internal timer less than zero. The process does not allow sigma points to move forward if their timers are less than zero. Meanwhile, the other half of the other sigma points that were initialized with positive internal timers move forward stretching the probability distribution.



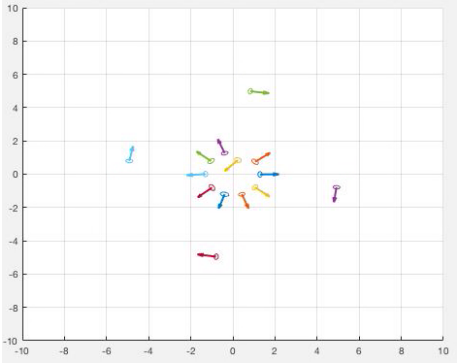
(a) Robots at time 0.1 with full UKF implemented



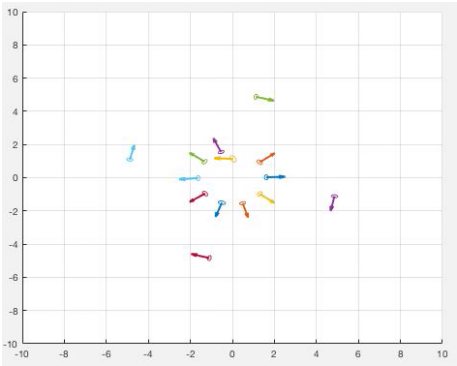
(b) Robots at time 1 with full UKF implemented

Fig. 10 With the Kalman Update added, the uncertainties collapse down quickly on to one point. All graphs have an x standard deviation of 0.25, y standard deviation of 0.25

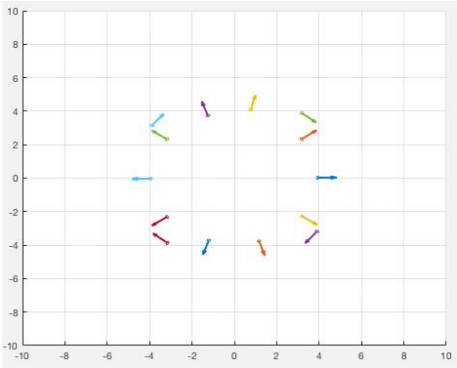
If we contrast Figure 9 with Figure 10, we can see that the full UKF can quickly collapses towards the most probable state.



(a) Orange arrow has the wrong trajectory

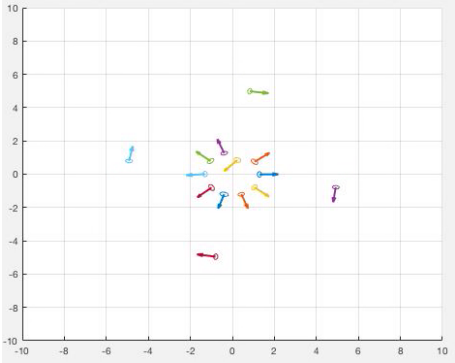


(b) Orange arrow is gradually correcting is trajectory

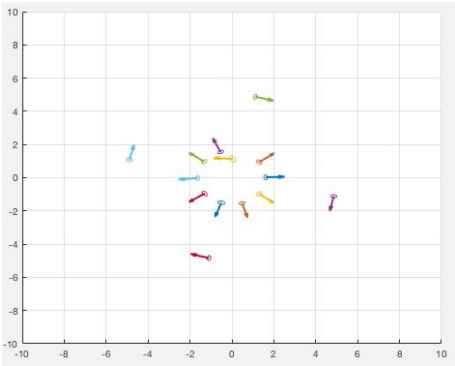


(c) Orange arrow has completely corrected is trajectory

Fig. 11 The Unscented Kalman filter will gradually estimate the most likely state based on noisy measurements of robot positions.



(a) Two colliding robots



(b) Robots after collision

(c) Robots after collision with Unscented Transform only

Fig. 12 The UKF is successfully able to track robot positions even after they collide. However, because of the poor representation of this process by a single gaussian distribution, performance is degraded.

When robots collide, they are supposed to either travel in the opposite direction or stop depending on the type of robot. As we can see in Figure 12c, the UT by itself had trouble accounting for these sudden shifts in movement and the probability distributions would spike as a result. The main reason for the large spikes is that some sigma points would collide before others due to positional variance amongst the points. The probability distributions would never recover as the initially collided states would recover and continue while the other corresponding sigma points would till be stuck in a collided state. However, the full UKF corrects itself with measurements. We can see these results in Figure 12a and Figure 12b

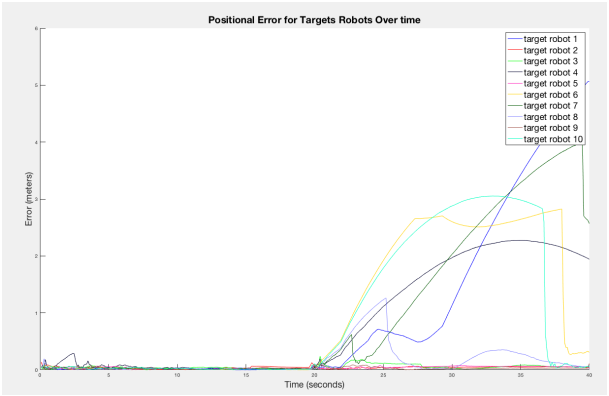


Fig. 13 Positional error between the true state and the UKF output for target robots

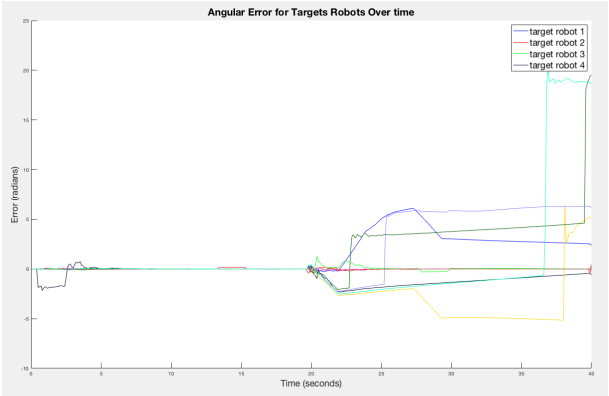


Fig. 14 Angular error between the true state and the UKF output for target robots

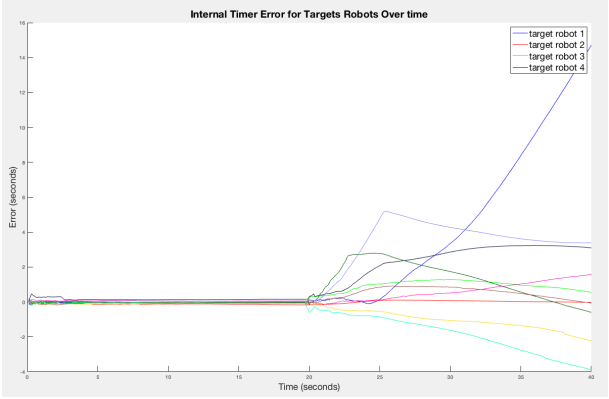


Fig. 15 Internal Timer error between the true state and the UKF output for target robots

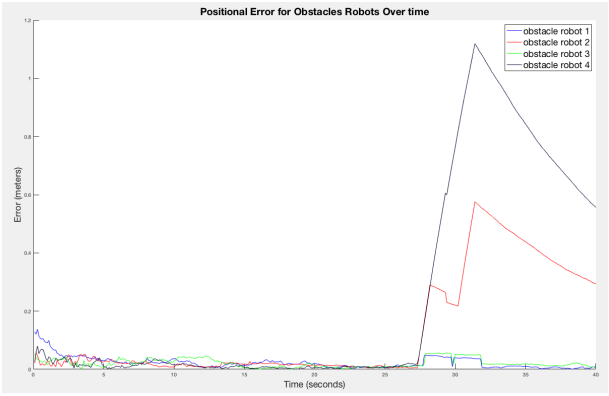


Fig. 16 Positional error between the true state and the UKF output for obstacle robots

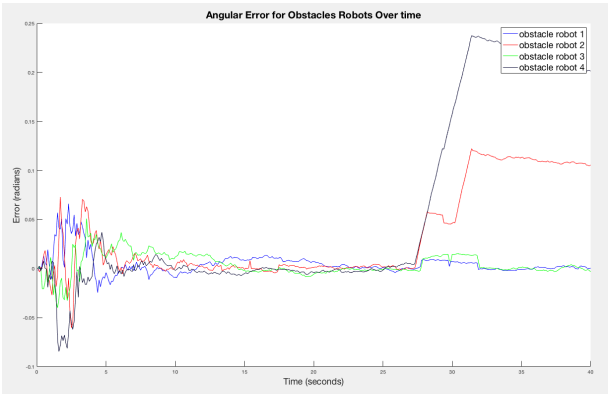
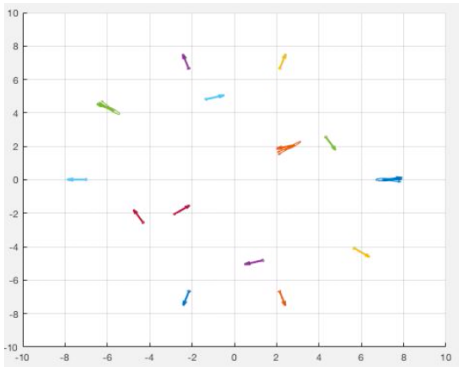


Fig. 17 Angular error between the true state and the UKF output for obstacle robots

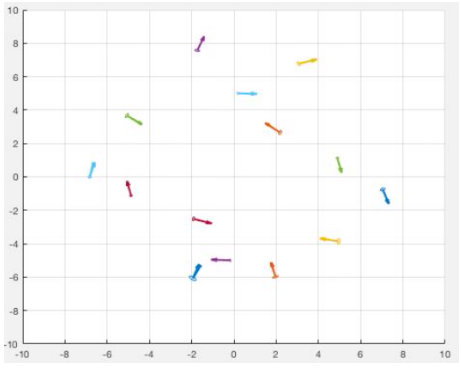
Figure 13 - Figure 17 show some common trends. The error is higher during the initial time steps but soon reaches near zero. However, the error spikes up again towards the end. According to Figure 13 - Figure 15, it takes about five seconds for the error to reach near zero for target robots. The error spikes up at 20 seconds which is when the target robots are programmed to turn and travel in the opposite direction. Looking at Figures 16 and Figure 17 it takes about ten seconds for the errors to reach near zero and the errors spike up again at about 25 seconds. This is strange as the obstacle robots aren't programmed to do anything different at 25 seconds.

4. Current Issues

One of the main issues that the program currently has is that it is unable track robot states once the target robots start to their programmed rotations at 20 seconds. As mentioned earlier, the error for all target robot dimensions increases rapidly at exactly 20 seconds. We currently are analyzing why our program loses its accuracy during this event.



(a) Target robots at time 20 starting a rotation



(b) UKF results a few moments after the rotation

Fig. 18 The UKF completely losing accuracy of robot positions after the programmed target robot rotation begins

Another issue is that we are do not know how consistently the UKF can maintain its ability to track the robots after a collision. Although our initial testing does indicate that the UKF can track robots after a collision, we have not yet been able to test it with a second set of collisions due to the fact that the UKF run for more than 20 seconds without

losing accuracy.

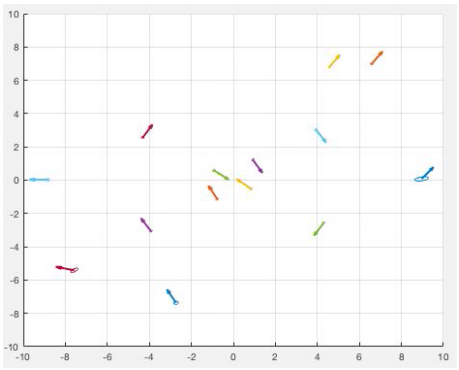


Fig. 19 The UKF state estimate at time 40 seconds

The biggest issue however, is that we have determined our process is non-Gaussian. As a result, several modifications will have to be made to the UKF as the UKF is traditionally optimized to handle problems with a Gaussian distribution. Some of these modifications are discussed in the next section.

5. Future Work

The UKF still needs a lot of tuning before it is ready to be put into production. We plan on improving the robustness of the algorithm by further adjusting the tuning parameters as well as adjusting initial covariance and error values to better match reality. We also plan on making the the Kalman updates at less frequent and random intervals. Currently, the Kalman Update runs every time step. This is not very realistic as measurement updates occur sparsely in real life.

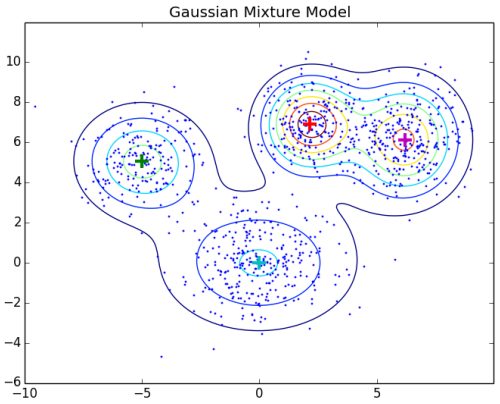


Fig. 20 A Gaussian Mixture Model with a multi-modal distribution. Image taken from Zhu's Machine Learning Garden Blog[13]

We are currently considering implementing a Gaussian Mixture Model (GMM) which uses the UKF. The GMM will produce multi-modal Gaussian distributions or 'branches'. Each time the process reaches a discontinuity (e.g. programmed rotation), the probability distribution will split in two. One branch will represent the probability distribution of the robots undergoing the discontinuous process (e.g. rotating) and the other branch will represent the probability distribution of the state if the discontinuity never occurred (e.g. moving forward).

Lastly, we plan on fitting probability distributions to a greater amount of sigma points for greater accuracy. Our current implementation allows $2n$-1 sigma points to be generated but more sigma points can be added by adding hidden dimensions to our state vector (x) [10].

I. Localization Based on Grid Nodes

The most challenging aspect of this competition is localizing globally with respect to the grid. Figure 1 shows an image from the real competition at Georgia Tech. The glare on the grid at low altitudes makes it impossible to sense grid lines and their colors at a far distance.

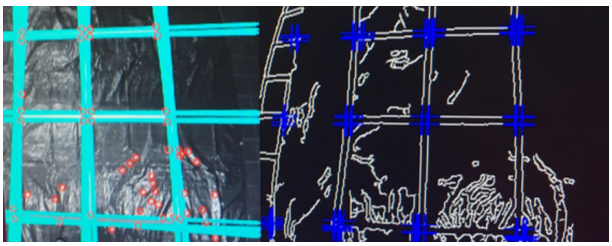
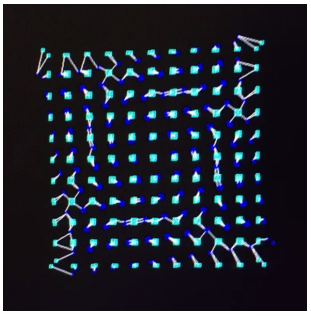
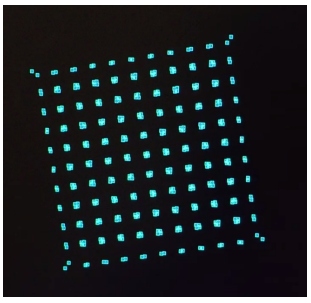


Fig. 21 Each of the methods we implemented relied on accurate and robust detection of line intersections. We call these intersections *Grid Nodes*. This figure shows results from our Grid Node detection algorithm on real sensor data. This method is based on the Hough Transform for line detection and a KNN clustering algorithm.



(a) Grid before 3D-2D ICP alignment



(b) Grid before 3D-2D ICP alignment

Fig. 22 We use a modified Iterative Closest Point (ICP) algorithm. Typically ICP is used to align two point clouds by computing a transformation which minimizes the squared position error between the closest points between point clouds. However, in our implementation, we optimize the *camera* pose by minimizing the 2D position error between detected grid nodes and projected grid nodes. This method excels in this scenario because the grid nodes are not unique.

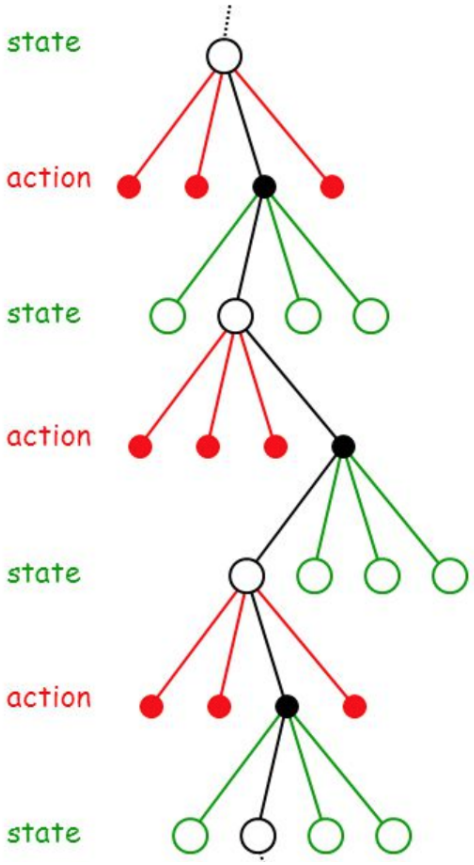


Fig. 23 A Markov Decision Process Visualized. Figure courtesy of [14]

J. High Level Decision Making

Previously, we used Neuro Evolution, a machine learning method, to train a neural network to pick a target and how many times to touch it. A video was made about that method and posted here: `https://www.youtube.com/watch?v=UO3ZPPVxk8E`

However, this semester we decided to prefer blocking over touching target robots. This is both simpler to execute and more effective in terms of rotating target robots.

We can think of this decision process as a Markov Decision Process (MDP). In our case for every state we will have ≤ 11 possible actions. 10 blocks and do nothing. For every action we get a new future state. The number of possible action sequences for n actions is 11^n in the worst case. This does not scale well ad it is effectively impossible to evaluate this MDP past a few actions.

We have decided to use our Unscented Transform described in section II.H to determine the state and its uncertainty after a given action sequence.

A greedy method will be implemented which picks the action sequence which maximizes the probability of

K. Minimum Time, Actuator Constrained Trajectory Generation

Once A goal is decided on and our state is being estimated, we now must plan a trajectory to the target robot. Our trajectory generator was based on work by Cutler [15]. In that paper an actuator constrained trajectory generator was described.

We extended this work further by developing an efficient trajectory optimizer which travels through way-points in a minimum time. It must be mentioned that we have not proven that this method is minimum time, but the results appear to be minimum time.

Cutler derives a function which computes open loop control inputs for a given multi rotor and parametric polynomial trajectory. Our method uses this function to determine if a given trajectory represented by a nth degree parametric polynomial trajectory is feasible.

1. Trajectory Optimization Method

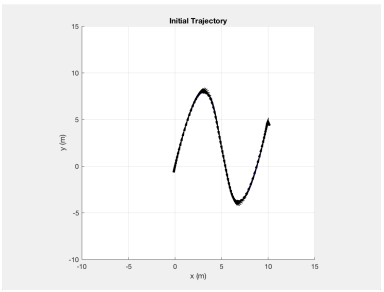
Given a set of waypoints, initial and final velocity, acceleration, jerk, snap, our method is capable of finding the optimal waypoint times which can be used to setup a linear system and solve for the coefficients of the nth order parametric polynomial representing the trajectory.

Algorithm 2 Trajectory Optimization Algorithm

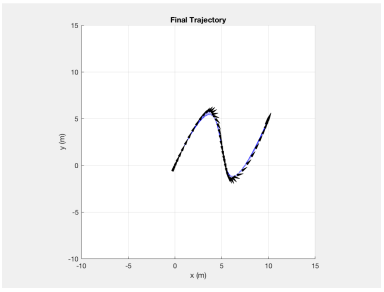
```
Waypoints stored in a list W;
Times of each waypoint initialized linearly and stored in a vector T;
for every waypoint in W, i represents the ith Waypoint do
    k = 1
    high = high scale
    low = low scale
    for desired number of iterations do scale the zeroth to the ith element of T by k. Recompute the trajectory with these new times.
        if trajectory is feasible then k = (k - low)/2
        end if
        if trajectory is not feasible then k = (high - k)/2 + k
        end if
    end for
    scale the zeroth to the ith element of T by low.
end for
    the result T vector is the optimal time vector.
```

The algorithm is based on a binary search and exploits the fact that as the change in time between waypoints decreases, the motor forces required become extreme.

2. Results

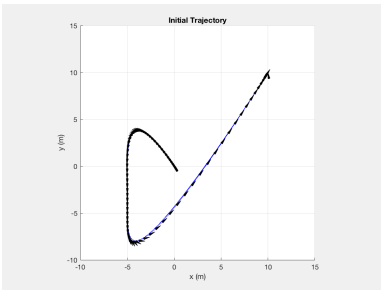


(a) Initial Trajectory. Flight time = 10 seconds

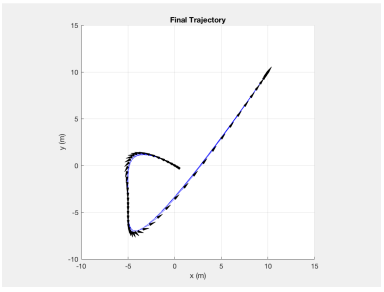


(b) Final Trajectory. Flight time = 5.82 seconds

Fig. 24 4 Waypoint trajectory before and after optimization. Arrows represent the velocity vector along the trajectory.



(a) Initial Trajectory. Flight time = 10 seconds



(b) Final Trajectory. Flight time = 6.92 seconds

Fig. 25 This trajectory includes 5 waypoint constraints and actuator constrains. Velocity and acceleration are unconstrained directly.

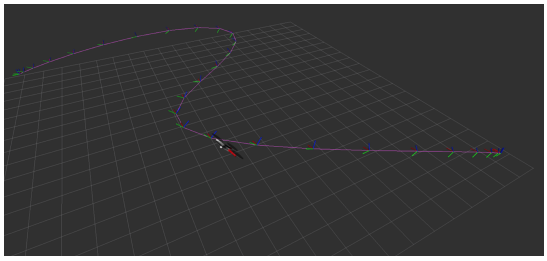


Fig. 26 A C++ version was also implemented. While the MATLAB implementation takes nearly 2 seconds to compute a Trajectory, the C++ implementation requires less than 50ms to compute. This speed improvement allows our system to generate a new trajectory over 10 times per second. Visualization software courtesy of OSRF [16]

.

III. Simulation

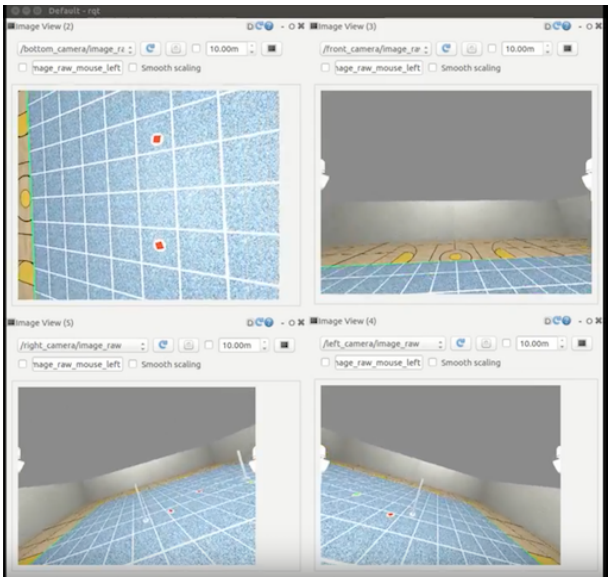


Fig. 27 Simulated camera images generated by the our Gazebo Simulator during a flight.

A full scale test of mission 7 at Purdue University is not feasible due to cost and space requirements. Instead we elected to create a highly accurate simulator. To d do this, we used Gazebo Sim [17] for generating fake sensor measurements and ground robot interactions. We used the Rotors Simulator [18] to accurately model the dynamics of a quadrotor, and finally we used a SITL simulator developed by the same group responsible for the PixHawk flight controller [19].

IV. Conclusion

This semester, we accomplished nearly all of our goals which we set out to accomplish with the exception of a real test flight. We have designed and built a new airframe, implemented target and obstacle detection, improved our visual inertial odometry algorithm, Implemented a UKF in MATLAB to evaluate its performance estimating target and obstacle states, and setup a global localization method using grid nodes.

We now plan to focus our efforts towards test flights, and simulated test flights. In addition, we will attempt to develop a state estimator which is more suited for estimating robots undergoing a discontinuous process.

Appendix

Contributions This Semester

Kevin J. Sheridan

Visual Inertial Odometry

I implemented EKF based visual inertial odometry with poor results, and developed a new method of visual inertial odometry described in this report.

This was and is the most challenging part of the project for me. I have struggled to make robust Visual Odometry. However, with this new method, I have achieved robust odometry.

Design an Manufacture Airframe

I designed and manufactured our new airframe by milling carbon fiber and 3D printing mounts. This part of the project took a lot of time and effort due to many setbacks when machining the carbon fiber plates. We now have all of the carbon fiber plates necessary to assemble the airframe.

Trajectory Optimizer

I improved our old trajectory optimization algorithm. This was done in both MATLAB and C++. At this point I must only implement error checks and error mitigation methods. Periodically, the algorithm produces an infeasible trajectory and does not know how to handle it.

PixHawk Simulator

I integrated the Rotors Simulator and SITL Pixhawk simulator into our old simulation.

Grid Node Based Global Localization

The software necessary for this algorithm was developed last semester in a separate repository. I refactored this code at the beginning of this semester. This algorithm is yet to be thoroughly tested.

Mitchell J. Lee

I wrote the entire UKF in MATLAB. This part occupied the majority of my time. I also refactored our blob detection program. We had an existing blob detection program that had to be debugged. I was able to get the blob detection working as well as started tuning it. Some tuning work still needed to be done on it but it was a lower priority task.

Challenges

I had the most trouble with understanding and interpreting the math behind the UKF. Luckily, Kevin helped me out a lot with that. By far, the UT took the longest to implement. This is mainly because I made a lot of logic errors. For example, I was using $\boldsymbol{x}'$ as the new state instead of $\boldsymbol{x}$. I also found it a little hard to check my results and debug since I was working with such large matrices. I eventually learned how to get creative and developed my own methods of testing my results.

Things I learned

This was my first time working with ROS and MATLAB. It was also my first time developing a more math heavy application. It required me to develop a different mindset in order to approach the task. This project also gave me greater knowledge about the math involved in developing robotic systems.

Jason K.C. Lo

Obstacle Detection

Since I had no prior experience with Robot Operating System (ROS), it was a very steep learning curve for me learn about how it works and the features ROS supports. After some research, I gained a basic understanding of how different programs interacts with each other and how data is transferred in ROS.

The main thing I accomplished this semester is the obstacle detection program. As mentioned in the report, this program has three major functions, namely 3D point clouds reconstruction, cluster extraction, and centroid computation. For 3D reconstruction, I initially attempted a method called Structure From Motion (SFM). The idea of SFM is that if multiple frames are taken by the same camera and the motion of the camera is known, I can estimate the distance of points using epipolar geometry.

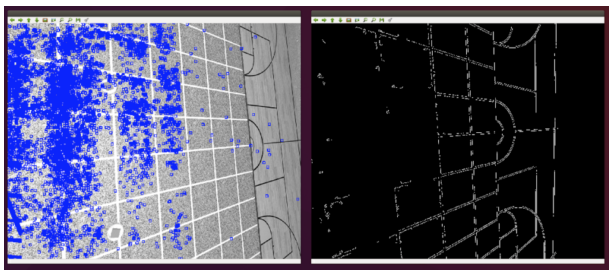


Fig. 28 Uneven distribution of feature in SFM.

I came across a few other problems during implementation of this program such as uneven distribution of features and sensitivity issues when detecting edges. A visualization of the problems is shown in figure above.

The feature distribution issue was solved by removing feature vector's size limit. I was able to solve most of rest problems by altering parameters and thresholds. figure below shows the SFM program tracking features across frames.

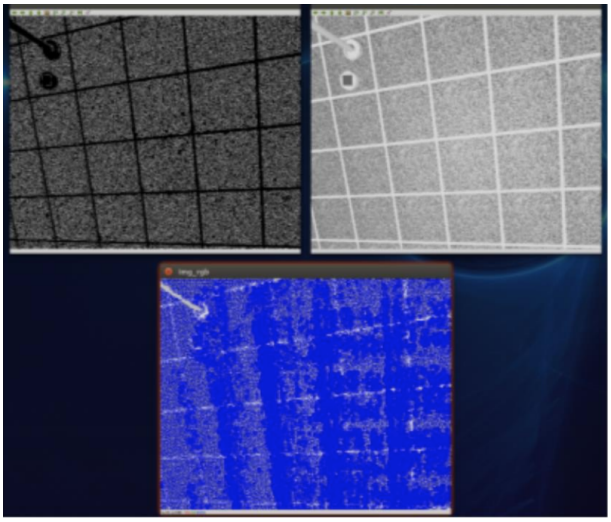


Fig. 29 visualization of SFM program's output .

However, we later discovered that SFM in this specific application may not yield the performance we expected. The SFM algorithm will break down when the aerial robot performs a momentary stop. Further, SFM relies on a good odometry estimate which is not guaranteed. For these reasons, the team decided to switch to using a pair of stereo cameras. The most notable benefit is that stereo pair gives two frames of the same object from different orientation at approximately the same instance. Not only does this approach work with dynamic obstacles, it is more reliable as it doesn't break down even when both our robot and the obstacles are static.

I then wrote the function PCProc to process the point cloud coming from the stereo cameras

I worked with Kevin Sheridan to create the dataset fixer program that calibrates the coordinate frames of the stereo cameras for the EuRoC dataset.

Kill Switch

Apart from obstacle detection, I also implemented the kill switch that was required by IARC. Initially I designed a circuit that uses transistors to act as switches between the Pixhawk flight controller and the electronic speed controller(ESC). A Teensy micro-controller was used to control the base current of the transistor and thereby switches it on and off. A schematics of the initial circuit and the finished circuit board are also shown.

During a test, I discovered a major problem that later led to a redesign of the entire circuit. Although the initial

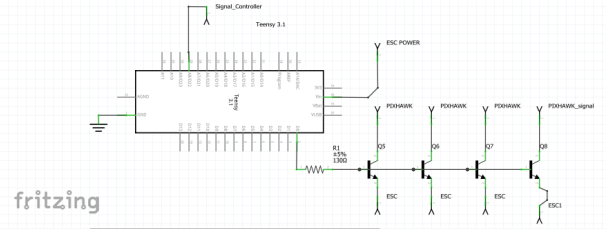


Fig. 30 Initial schematics of kill switch.

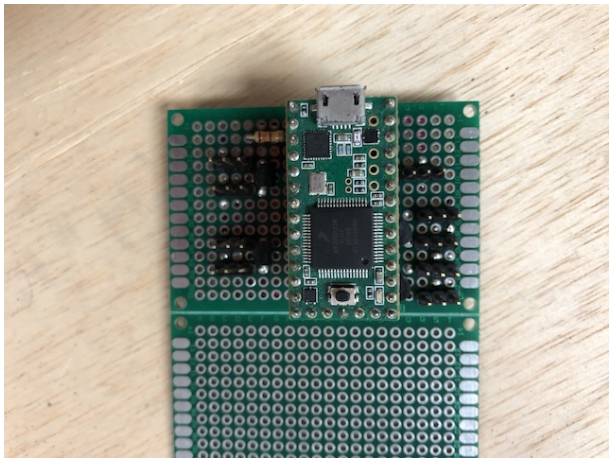


Fig. 31 Front view of the kill switch circuit board.

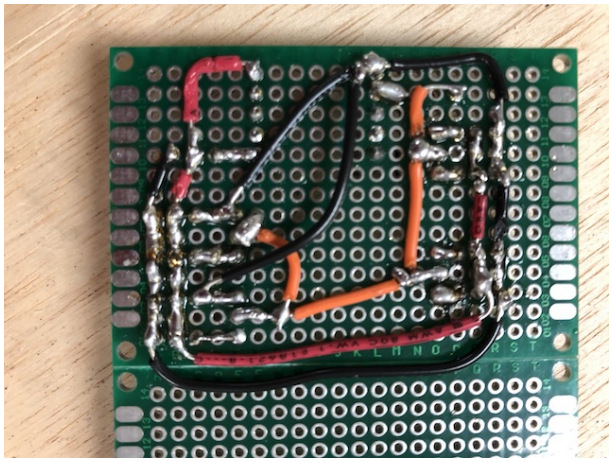


Fig. 32 Back view of the kill switch circuit board.

conductivity test proved that signals do pass through the transistor when the kill switch is not engaged, the quality of the signal was not preserved. This is an issue especially for PWM signals since the weaken signal may not be enough to trigger the ESCs or causes stability issues at the very least. As a result, I had to abandon this circuit and redesign a new version. I decided better idea would be to cut off the power to the ESCs. Two major reasons were that cutting power is objectively safer and it eliminates the signal corruption issue entirely. I used relays instead of transistor since a much higher current must pass through the circuit. Since the redesigned circuit is fundamentally different and involves the entire power system, I had to consider the placement of each components and resource all the components.

I tested each individual component and made sure they worked as expected. The entire layout of the power system was planned and put together. A picture of the system is shown below.

Simulation

In addition, I helped Kevin Sheridan to create a simulation description form so we can create the aerial robot in simulation. This form describes the physical properties of the robot and how it should interact with the environment.

Airframe Manufacturing

Lastly, I worked with both other team members to manufacture and assemble the body frame of our aerial robot.

Links to Source Code

A. Visual Inertial Odometry
`https://github.com/Trexter/invio`

B. Minimum Time Actuator Constrained Trajectory Generation and Control
`https://github.com/purdue-arc/multirotor_trajectory`

C. Target Robot Detection
`https://github.com/purdue-arc/m7_tracking`

D. Obstacle Detection
`https://github.com/purdue-arc/obstacle_detection`

E. Mission 7 Master: High Level Mission Planning and Execution
`https://github.com/purdue-arc/mission7_master`

Funding Sources

References

[1] Robert C. Michelson, e. a., "International Aerial Robotics Competition," , 2018. URL http://www.aerialroboticscompetition.org/.

[2] "Team Website of Ascend NTNU," , 2018. URL https://ascendntnu.no/.

[3] Bloesch, M., Omari, S., Hutter, M., and Siegwart, R., "Robust Visual Inertial Odometry Using a Direct EKF-Based Approach," *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany*, ETH-Zürich, 2015. doi:https://doi.org/10.3929/ethz-a-010566547.

[4] Forster, C., Pizzoli, M., and Scaramuzza, D., "SVO: Fast Semi-Direct Monocular Visual Odometry," *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[5] Itseez, "Open Source Computer Vision Library," https://github.com/itseez/opencv, 2018.

[6] Eade, E., "Lie Groups for 2D and 3D Transformations," , 2017. URL http://ethaneade.com/lie.pdf.

[7] Bourmaud, M. R. A. M. e. a., G., "J Math Imaging Vis," , 2015.

[8] Burri, M., Nikolic, J., Gohl, P., Schneider, T., Rehder, J., Omari, S., Achtelik, M. W., and Siegwart, R., "The EuRoC micro aerial vehicle datasets," *The International Journal of Robotics Research*, 2016. doi:10.1177/0278364915620033, URL http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.abstract.

[9] Rusu, R. B., and Cousins, S., "3d is here: Point cloud library (pcl," *In Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, ????, pp. 1–4.

[10] Zandt, J. R. V., "A more robust unscented transform," 2017. URL https://www.mitre.org/sites/default/files/pdf/vanzandt_unscented.pdf.

[11] *A New Extension of the Kalman Filter to Nonlinear Systems*, Signal Processing, author = Simon J. Julier and Jeffery K. Uhlmann , publisher = AEROSENSE, address = Orlando,Florida, year = 1997, doi = 10.1117/12.280797, URL =https://people.eecs.berkeley.edu/ pabbeel/cs287-fa15/optreadings/JulierUhlmann-UKF.pdf, Sensor Fusion, and Target Recognition VI, Vol. 3068, ????

[12] Fadali, M. S., "Unscented Kalman Filter," , ???? URL https://wolfweb.unr.edu/~fadali/EE782/UnscentedKF.pdf.

[13] Zhu, Y., "Yu's Machine Learning Garden," , 2014. URL http://4.bp.blogspot.com/-zuCQBrN8990/VGt45PZHXhI/AAAAAAAAA1E/jtQQaAj-PMc/s1600/gmm2.png.

[14] Sutton, R., "From Markov Decision Process to Artificial Intelligence," , 2015. URL http://slideplayer.com/slide/3007502/.

[15] M. Cutler, J. H., "Actuator Constrained Trajectory Generation and Control for Variable-Pitch Quadrotors," , 2012.

[16] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., "ROS: an open-source Robot Operating System," *ICRA Workshop on Open Source Software*, 2009.

[17] Koenig, N., and Howard, A., "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," ????

[18] Furrer, F., Burri, M., Achtelik, M., and Siegwart, R., *Robot Operating System (ROS): The Complete Reference (Volume 1)*, Springer International Publishing, Cham, 2016, Chaps. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. doi:10.1007/978-3-319-26054-9_23, URL http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[19] Project, D., "PixHawk SITL Simulator," , 2018. URL https://github.com/PX4/sitl_gazebo.