Spring 5-2020

UAV 6DOF Simulation and Kalman Filter for Localizing Radioactive Sources

John G. Goulet
Univeristy of Maine, john.goulet@maine.edu

UAV 6DOF SIMULATION AND KALMAN FILTER FOR LOCALIZING RADIOACTIVE SOURCES

By

John George Goulet

B.S., University of Maine, 2017

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Engineering

(in Engineering Physics)

The Graduate School

The University of Maine

May 2020

Advisory Committee:

C.T. Hess, Professor of Physics and Astronomy, Advisor

Sam Hess, Professor of Physics and Astronomy

David Rubenstein, President/Research Engineer, Maine Aerospace Consulting, LLC.

UAV 6DOF SIMULATION AND KALMAN FILTER FOR LOCALIZING RADIOACTIVE SOURCES

By John George Goulet

Thesis Advisor: Dr. C.T. Hess

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Engineering
(in Engineering Physics)
May 2020

Unmanned Aerial Vehicles (UAVs) expand the available mission-space for a wide range of budgets. Using MATLAB, this project has developed a six degree of freedom (6DOF) simulation of UAV flight, an Extended Kalman Filter (EKF), and an algorithm for localizing radioactive sources using low-cost hardware. The EKF uses simulated low-cost instruments in an effort to estimate the UAV state throughout simulated flight.

The 6DOF simulates aerodynamics, physics, and controls throughout the flight and provides outputs for each time step. Additionally, the 6DOF simulation offers the ability to control UAV flight via preset waypoints or in realtime via keyboard input.

Using low-cost instruments, the EKF fuses measurements with a nonlinear UAV model to estimate UAV states. The 6DOF simulation was used to compare the true UAV states with the estimated states. EKF results indicate appropriate estimation of states with the exception of UAV yaw. An additional sensor providing yaw information would improve estimation accuracy.

Radioactive sensors which are capable of providing position information are prohibitively expensive. The radioactive source localization algorithm utilizes count-based sensors such as a Geiger counter to estimate the location of a radioactive source. The algorithm constructs a three dimensional gradient using six measurements and attempts to

determine the source position from this gradient. The algorithm was developed such that a wide range of environmental parameters could be localized by swapping the Geiger counter with an alternative count-based instrument.

ACKNOWLEDGEMENTS

I would like to thank Dr. C.T. Hess and my entire committee for their understanding and assistance while working on this thesis remotely. Dr. C.T. Hess's patience, understanding, and input was a critical help throughout the completion of this thesis. I would like to thank Dr. Sam Hess for my first introduction to quadcopters and his responsiveness while working remotely. Additionally, I would like to thank Dr. David Rubenstein for the opportunity to learn from his expertise as well as his participation in hosting three independent studies. I would like to thank Dr. Alex Friess for all aviation related discussions and fueling my interest in aviation. I would like to thank Dr. Richard Eason for his assistance and flight training. I would like to thank my girlfriend, Katie Manzo for her love and continued support in everything I set my mind to. I would like to thank my parents, Ron and Jaye Goulet and my brothers, Raymond and Jason Goulet, for their assistance, dedication, and support over all these years. I would like to thank Will Riihiluoma, Katee Schultz, Mary Jane Yeckley, and Ben Hebert. Finally, I would like to thank anyone not explicitly mentioned that supported me during my time at the University of Maine.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Table 1: Variable definitions

Variable	Definition
${}^{B}\vec{A}$	Sum of aerodynamic forces in the body frame.
A	Activity
A_{thresh}	Activity threshold
A_{+}	Activity measured along positive axis
A_{-}	Activity measured along negative axis
Superscript B	Variable expressed in body frame.
b_x	body-frame x-coordinate
b_y	body-frame y-coordinate
b_z	body-frame z-coordinate
CG	Center of gravity
C_T	Thrust coefficient
C_D	Drag coefficient
${}^{W}\vec{D}$	Drag vector expressed in the wind frame.
${}^{B}\vec{D}$	Drag vector expressed in the body frame.
DCM	Direction cosine matrix
${}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{I}}$	Inertial to body DCM
${}_{\boldsymbol{I}}\boldsymbol{T}_{\boldsymbol{B}}$	Body to inertial DCM
${}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{W}}$	Body to wind DCM

Table 1: Variable definitions (continued)

Variable	Definition
$_{\boldsymbol{W}}\boldsymbol{T}_{\boldsymbol{B}}$	Wind to body DCM
d	Distance to a source
$^{B}\vec{F}$	External forces expressed in the body frame.
$\boldsymbol{F}_{k-1}$	System-update matrix
f_{k-1}	System update function to update state from x_{k-1} to x_k
$^{I}\vec{g}$	Local acceleration due to gravity expressed in the inertial frame.
H	Angular momentum
h_k	Measurement function
$\boldsymbol{H}_k$	measurement matrix
$^{B}\boldsymbol{I}$	Inertia matrix expressed in the body frame.
K_p	Proportional gain
K_d	Derivative gain
K_i	Integral gain
$\boldsymbol{K}_k$	Kalman gain matrix
L	Propeller length
m	UAV mass
$^{B}\vec{M}$	External torques expressed in the body frame.
$^{B}\vec{M}_{T_i}$	Torque due to thrust from ith propeller expressed in the body frame.
$^{B}\vec{M}_{T}$	Torque due to thrust from all propellers in the body frame.
NED	North-East-Down frame.
$\boldsymbol{P}_k^-$	*a priori* Estimation error covariance matrix
$\boldsymbol{P}_k^+$	*a posteriori* Estimation error covariance matrix
$\boldsymbol{Q}_k$	System noise covariance
$\boldsymbol{R}_k$	Measurement noise covariance

Table 1: Variable definitions (continued)

Variable	Definition
${}^{B}\vec{r}_{\mathrm{prop}_i}$	Propeller position relative to the body frame.
${}^{I}\vec{r}_source$	Source position expressed in the inertial frame.
${}^{I}\vec{r}_{UAV}$	UAV inertial position
S_{prop}	Propeller reference area
S_{UAV}	UAV reference area
${}^{B}\vec{T}_i$	Thrust vector from ith propeller expressed in the body frame.
${}^{B}\vec{T}$	Sum of propeller thrust vectors expressed in the body frame.
u	Body-frame x-velocity
u	System input vector
V	Airspeed
${}^{B}\vec{v}$	Translational velocity of the body expressed in the body frame.
v	Body-frame y-velocity
w	Body-frame z-velocity
x_k	True state estimate at time k
$\hat{x}_k^-$	*a priori* state estimate
$\hat{x}_k^+$	*a posteriori* state estimate
w_{k-1}	System noise
y_k	Measurement vector
α	Angle of attack
β	Side-slip angle
Δz	Error from target altitude.
$\Delta \dot{z}$	Error from target vertical velocity.
$\Delta \phi$	Error from target roll
$\Delta \dot{\phi}$	Error from target roll

Table 1: Variable definitions (continued)

Variable	Definition
$\Delta\theta$	Error from target pitch
$\Delta\dot{\theta}$	Error from target pitch rate
θ	Pitch
$\dot{\theta}$	Pitch rate
θ_c	Commanded pitch
λ	Propeller rotation speed squared.
ν_k	Measurement noise
ρ	Density of air
σ_+	Standard deviation of activity measured along positive axis.
σ_-	Standard deviation of activity measured along negative axis.
ϕ	Roll
$\dot{\phi}$	Roll rate
ϕ_c	Commanded roll
ψ	Yaw
$\dot{\psi}$	Yaw rate
χ	Obstacle position
$\vec{\omega}$	Angular velocity of the body.

CHAPTER 1
INTRODUCTION

Unmanned aerial vehicle (UAV) popularity has skyrocketed given the numerous applications and uses they provide. UAVs typically require no fuel (most are powered by electric motors), require little space for takeoff, provide a cost-effective means for payload transportation, and can travel to hazardous locations. Some of the many fields that utilize UAVs include aerospace, forestry, search and rescue, remote sensing, photography, remote imaging, mapping, and exploration.

The ability to quickly navigate throughout hazardous locations allows for the mapping of regions unsuitable for humans. These regions may have high concentrations of radioactivity, harmful gasses, or any number of qualities harmful to humans. This thesis focuses on environments with high concentrations of radioactivity. Measuring the location of radioactive sources typically requires the use of high cost, stationary sensors. Low cost sensors, such as Geiger counters, are count-based and provide no information on the position or direction of a radioactive source. By combining the mobility of a UAV with a Geiger counter (or another count-based instrument), an algorithm has been developed to estimate the position of radioactive sources. The algorithm is not limited to localize radioactive sources. Any count-based instrument could be used for the purpose of measuring a particular environmental parameter of interest.

Estimating the position of radioactive materials using low and high cost sensors has been studied before in [1, 3, 8]. These studies use stationary sensors which may be difficult or impossible to setup in hazardous locations. Additionally, these studies are unable to detect concentrations of radioactivity at a range of altitudes.

The development of an algorithm for localizing a radioactive source was split into three sections. First, a UAV simulation environment was developed in MATLAB for the purpose of testing UAV flight characteristics and source localization. Next, an Extended Kalman

Filter was developed for the purpose of estimating UAV position and attitude. Finally, the localization algorithm was created and verified from the simulation.

CHAPTER 2

METHODS

2.1 UAV Dynamics

The UAV is assumed to be a rigid-body quadcopter whose body frame origin is located at its center of mass (CG). Figure 2.1 shows the body axes from a top-down view of a UAV. The inertial frame of reference is a north-east-down (NED) frame where north, east, and down represent the positive x, y, and z axes respectively. While this is not a true inertial frame, it is sufficient for the purpose of simulating quadcopter flight over a small region.

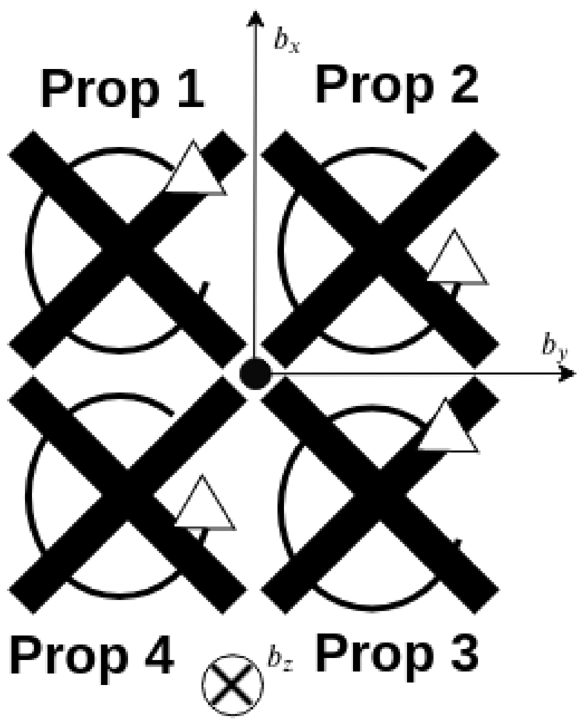


Figure 2.1. UAV body-frame diagram. Diagram indicates body-frame origin, propeller numbering, and direction of propeller rotation. Note the direction of propellers 1 and 3 rotate opposite propellers 2 and 4.

As most UAVs are battery powered, the CG is taken to be time-invariant allowing a simplified version of Euler's rigid body equations of motion to be used as shown in (2.1).

The superscript "B" before a variable indicates that the quantity is formulated in the body-frame.

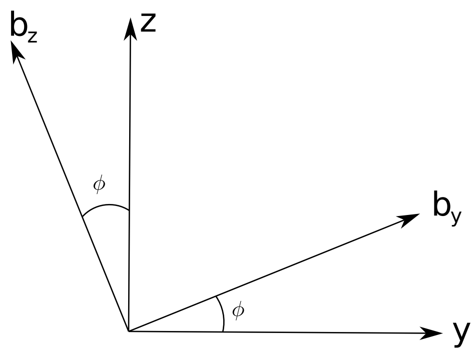
$$
\begin{aligned}
m{}^{B}\dot{\vec{v}} &= {}^{B}\vec{F} - m{}^{B}\vec{\omega} \times {}^{B}\vec{v} \\
{}^{B}\boldsymbol{I}\dot{\vec{\omega}} &= {}^{B}\vec{\tau} - {}^{B}\vec{\omega} \times {}^{B}\boldsymbol{I}\vec{\omega}
\end{aligned}
\tag{2.1}
$$

Quantities ${}^{B}\vec{F}$ and ${}^{B}\vec{\tau}$ represent all inertial forces (non-fictitious) and torques acting on the CG expressed in the body frame. The additional forces acting on the body include gravity, drag, and thrust. Additional torques are nonzero only when UAV propellers have different rotation speeds. ${}^{B}\boldsymbol{I}$ is the inertia tensor of the UAV with respect to the body frame, ${}^{B}\vec{v}$ is the translational velocity, and $\vec{\omega}$ is the angular velocity of the body. Note that for clarity, the body-frame is to be the assumed frame of reference unless indicated otherwise.

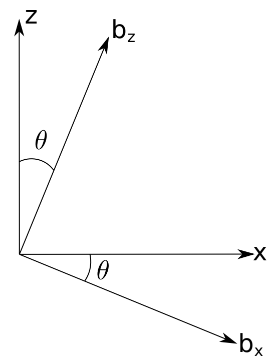
2.1.1 UAV Orientation

To transform between NED and body frames, a 3-2-1 rotation matrix was constructed using Euler angles ϕ, θ, and ψ (roll, pitch, and yaw). Roll is defined as a rotation about the $\hat{b}_x$ axis with respect to the NED x-y plane, pitch is a rotation about the $\hat{b}_y$ axis with respect to the NED x-y plane, and yaw is a rotation about the $\hat{b}_z$ axis with respect to the NED x-z plane. These definitions are shown in Figure 2.2. An illustration of the Euler angles describing UAV attitude is shown in Figure 2.3. The rates of change of the Euler angles are determined using (2.2). The derivation of (2.2) can be found from a variety of sources such as [5] and is not provided.

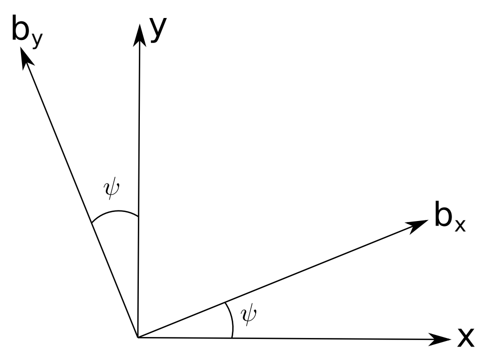
$$
\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}
\tag{2.2}
$$



(a) UAV roll definition.



(b) UAV pitch definition



(c) UAV yaw definition

Figure 2.2. UAV Euler angles

Equation 2.3 shows the complete direction cosine matrix (DCM) for transforming from the body frame to the inertial frame. The opposite transform, from inertial to body, can be obtained by taking the matrix transpose of (2.3).

$$ {}_{\boldsymbol{I}}\boldsymbol{T}_{\boldsymbol{B}} = \begin{bmatrix} \cos\theta\cos\psi & \cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \sin\phi\sin\psi + \cos\phi\cos\psi\sin\theta \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\theta\sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix} \tag{2.3} $$

$$ {}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{I}} = {}_{\boldsymbol{I}}\boldsymbol{T}_{\boldsymbol{B}}^{T} \tag{2.4} $$

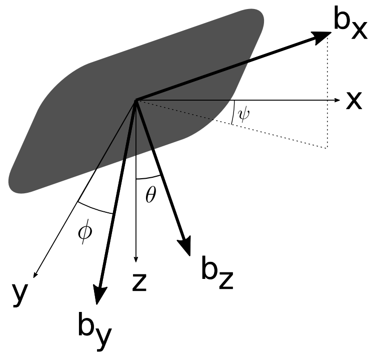


Figure 2.3. All UAV Euler angles

2.1.2 Aerodynamics

Aerodynamic forces acting on a quadcopter UAV include lift, drag, and thrust. In many quadcopter models, lift and drag are neglected due to their small quantities. The aerodynamic model presented includes drag and thrust forces.

Drag is computed in the wind-frame (superscript W) as shown in (2.5). The wind-frame is a frame of reference with the positive x-axis lying along the velocity vector of the aircraft relative to the surround air. The positive z-axis points perpendicular to the x-axis and below the UAV. The positive y-axis completes the right-handed coordinate system.

$$
{}^{W}\vec{D} = 0.5\rho V^2 S_{UAV} C_D \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \tag{2.5}
$$

In (2.5), ρ is the density of air, V is the UAV velocity relative to the surrounding air, S is the reference area of the UAV, and C_D is the coefficient of drag. ${}^{W}\vec{D}$ must be transformed to the body-frame for use in (2.1). The body-to-wind transform matrix is shown in (2.6). The reverse is, again, found by taking its transpose as shown in (2.7). In (2.6), α is the angle of attack and β is the side-slip angle computed using body-frame velocities u, v, and w, as defined in (2.8) and (2.9) respectively.

$$
{}_{\boldsymbol{W}}\boldsymbol{T}_{\boldsymbol{B}} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\beta & -\cos\beta\sin\alpha \\ -\cos\alpha\sin\beta & \cos\beta & \sin\alpha\sin\beta \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{2.6}
$$

$$
{}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{W}} = {}_{\boldsymbol{W}}\boldsymbol{T}_{\boldsymbol{B}}^{T} \tag{2.7}
$$

$$
\alpha = \arctan(\frac{w}{u}) \tag{2.8}
$$

$$
\beta = real(\arcsin(\frac{v}{\sqrt{u^2 + w^2}})) \tag{2.9}
$$

Finally, the body-frame drag is shown in (2.10). There is assumed to be no torque on the body from drag.

$$
{}^{B}\vec{D} = {}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{W}}{}^{W}\vec{D} \tag{2.10}
$$

The thrust force was determined by modeling each propeller blade as a wing slicing the air at a speed equal to the tangential velocity half-way along each propeller. Variable λ is defined as propeller rotation speed squared, allowing propeller thrust to be written as shown in (2.11). Refer to Figure 2.1 to see that the propellers are fixed in the $\hat{b}_x$-$\hat{b}_y$ plane. Assuming the propellers are fixed in the body xy plane (as indicated in Figure 2.1), thrust must act along the $-\hat{b}_z$ direction.

$$
{}^{B}\vec{T}_i = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} 0.5\rho\lambda_i\left(\frac{L}{2}\right)^2 S_{prop}C_T \tag{2.11}
$$

$$
{}^{B}\vec{T} = \sum_{i=1}^{4} {}^{B}\vec{T}_i \tag{2.12}
$$

Thrust forces from each propeller are summed to produce the total thrust force acting on the UAV (2.12). For the ith propeller, the cross product of propeller position and propeller thrust results in a torque as shown in (2.13).

$$^B\vec{M}_{T_i} = {}^B\vec{r}_{\text{prop}_i} \times {}^B\vec{T}_i \tag{2.13}$$

$$^B\vec{M}_T = \sum_{i=1}^{4} {}^B\vec{M}_{T_i} \tag{2.14}$$

Summing the drag force in (2.10) and the thrust force in 2.12 results in the total aerodynamic force acting on the body expressed in the body-frame (2.15).

$$^B\vec{A} = {}^B\vec{D} + {}^B\vec{T} \tag{2.15}$$

2.1.3 Basic Quadcopter Motion

With thrust defined in the previous section, a discussion on how a quadcopter achieves translational motion may be helpful for those less familiar. Refer to Figure 2.1 for body-axes and propeller numbering.

Roll can be achieved by setting the following conditions: $\lambda_1 = \lambda_4$, $\lambda_2 = \lambda_3$, and finally $\lambda_1 > \lambda_2$ (recall that λ_i is the ith propeller rotation speed squared). Under this condition, propellers 1 and 4 will produce a greater thrust than propellers 2 and 3 resulting in a torque about $\hat{b}_x$ (2.13).

The condition for changes in pitch are similar to those for roll. Setting the conditions: $\lambda_1 = \lambda_2$, $\lambda_4 = \lambda_3$, and $\lambda_1 > \lambda_4$ results in a greater thrust force from propellers 1 and 2, thus creating a torque about the $\hat{b}_y$ axis.

If the body x and y axes are aligned with the NED x and y axes, increasing the roll angle will tilt the body z axis in the in the NED $-\hat{y}$ direction, accelerating the UAV east ($+\hat{y}$ direction). Similarly, a positive pitch angle will accelerate the UAV south ($-\hat{x}$).

Yaw motion is less intuitive than roll and pitch. Recall that propellers 1 and 3 rotate opposite the direction of propellers 2 and 4. When all propellers rotate uniformly, the total angular momentum of the system is zero (2.16). Yaw rotation can be induced by changing the propeller rotation speed of diagonal propellers. Figure 2.4 indicates that propellers 1 and 3 rotate faster than propellers 2 and 4. Using (2.17), increasing counter-clockwise propeller speeds will cause the UAV body to rotate clockwise about the $\hat{b}_z$ axis.

$$H_{tot} = 0 = (H_1 + H_3) - (H_2 + H_4) \tag{2.16}$$

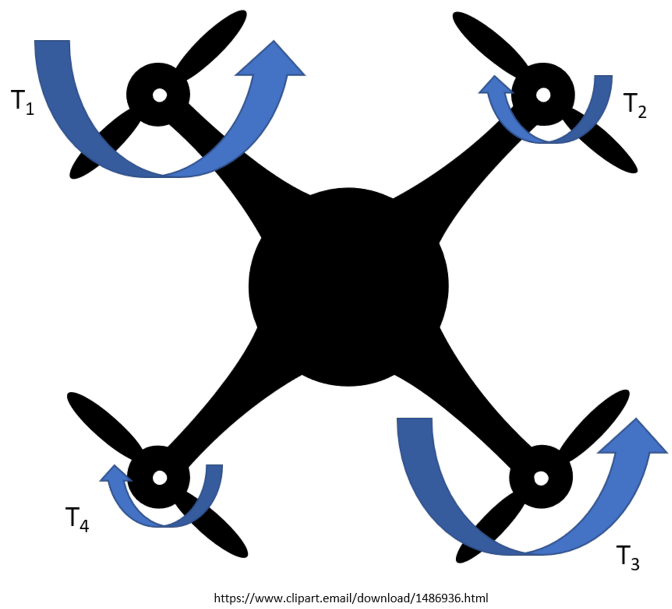


Figure 2.4. UAV Yaw diagram. Propellers 1 and 3 rotate faster than propellers 2 and 4. This will induce a clockwise motion about the UAV.

$$\begin{aligned} 0 &= (H_1 + H_3) - (H_2 + H_4) + H_{UAV} \\ -H_{UAV} &= (H_1 + H_3) - (H_2 + H_4) \end{aligned} \tag{2.17}$$

2.1.4 UAV Equations of Motion

The expanded equations of motion from (2.1) along with all other equations used to model UAV motion are provided below in (2.18) - (2.20) for convenience. Recall that items in bold indicate matrix or vector quantities. All vectors are written in the body frame unless otherwise noted.

$$ {}^B\dot{\vec{v}} = {}_{\boldsymbol{B}}\boldsymbol{T_I}\,{}^I\vec{g} + \frac{1}{m}{}^B\vec{A} - {}^B\vec{\omega} \times {}^B\vec{v} \tag{2.18} $$

$$ {}^B\dot{\vec{\omega}} = {}^B\boldsymbol{I}^{-1}({}^B\vec{M} - {}^B\vec{\omega} \times {}^B\boldsymbol{I}\,{}^B\vec{\omega}) \tag{2.19} $$

$$ \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{2.20} $$

2.2 Controls

2.2.1 UAV Hover and Altitude Control

The UAV defaults to hovering at a constant altitude if no commands are input. Equation 2.21 shows the required λ setting to hover while under straight and level flight conditions.

$$ \lambda_{\text{hover}} = \frac{mg}{2\rho(0.5L_{prop})^2 S_{prop} C_T} \tag{2.21} $$

If the UAV has nonzero roll or pitch, (2.21) will not provide adequate thrust to maintain straight and level flight. Equation 2.22 shows the general form (2.21) that applies for all practical flight conditions.

$$ \lambda_{\text{hover}} = \frac{mg}{2\rho(0.5L_{prop})^2 S_{prop} C_T \cos\phi\cos\theta} \tag{2.22} $$

For changes in altitude, a proportional-derivative (PD) controller was used to command reasonable changes in propeller rotation speed. The general form of the PD controller is shown in (2.23). K_p and K_d are the proportional and derivative gains which were determined experimentally. Δz is the error between the target altitude (z_t) and the current altitude (z). $\Delta\dot{z}$ is the error between the target z velocity and the actual NED z velocity. The target z velocity is always is always set to zero. If both errors are zero, λ is equal to λ_{hover}.

$$\lambda_{alt} = (1 - K_p\Delta z + K_d\Delta\dot{z})\lambda_{hover} \tag{2.23}$$

2.2.2 UAV Roll, Pitch, and Horizontal Motion Control

From Chapter 2.1.3, all horizontal motion is directly related to changes in roll and/or pitch angles. Those angles are controlled using a proportional-integral-derivative (PID) controller. The general form of the PID controller used is shown in (2.24).

$$\begin{aligned} \phi_c &= K_p\Delta\phi + K_d\Delta\dot{\phi} + K_i\int_0^T \Delta\phi dt \\ \theta_c &= K_p\Delta\theta + K_d\Delta\dot{\theta} + K_i\int_0^T \Delta\theta dt \end{aligned} \tag{2.24}$$

The lefthand side of (2.24) (ϕ_c and θ_c) is the roll/pitch angle to be commanded this timestep. The integral term is the sum of all errors leading up to the current simulation time. Note that a maximum rotation angle of five degrees is enforced throughout the simulation. If (2.24) would command a roll or pitch greater than five degrees, the commanded angle is instead set to five degrees. This was done to improve control stability.

With ϕ_c and θ_c known, these commanded angles were used to compute a difference in propeller rotation speeds ($\Delta\lambda$) based on the desired motion. Each propeller is mounted 45 degrees from the CG, allowing (2.11) to be rewritten as shown in (2.25).

$$\begin{aligned} M_i &= \pm T_i L_{prop}\cos(45°) \\ M_i &= \pm\frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \lambda_i \cos(45°) \\ M &= \frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \cos(45°)(\pm\lambda_1 \pm \lambda_2 \pm \lambda_3 \pm \lambda_4) \end{aligned} \tag{2.25}$$

The signs of λ in (2.25) are determined by the desired direction of motion. If the desired direction of travel is along the b_y direction, (2.25) becomes (2.26).

$$
\begin{aligned}
M &= \frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \cos(45^\circ)(\lambda_1 + \lambda_4 - (\lambda_2 + \lambda_3)) \\
M &= \frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \cos(45^\circ)(\lambda_{1,4} - \lambda_{2,3}) \\
M &= \frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \cos(45^\circ)(\Delta\lambda_y)
\end{aligned} \tag{2.26}
$$

Referring back to (2.1), the lefthand side of the torque term is set to the righthand side of (2.27). Equation 2.26 is the τ term.

$$
\begin{aligned}
\theta_c &= \omega_y \Delta t + 0.5\dot{\omega}_y (\Delta t)^2 \\
\dot{\omega}_y &= 2\frac{\theta_c - \omega_y \Delta t}{(\Delta t)^2}
\end{aligned} \tag{2.27}
$$

$$
\begin{aligned}
{}^B\boldsymbol{I}\dot{\vec{\omega}} &= {}^B\vec{\tau} - \vec{\omega} \times {}^B\boldsymbol{I}\vec{\omega} \\
2I_{yy}\frac{\theta_c - \omega_y \Delta t}{(\Delta t)^2} &= \frac{1}{8}\rho L_{prop}^3 S_{prop} C_T \cos(45^\circ)(\Delta\lambda_y) - \left(\vec{\omega} \times {}^B\boldsymbol{I}\vec{\omega}\right)_y
\end{aligned} \tag{2.28}
$$

In (2.27) and (2.28), Δt is the simulation timestep and $(\vec{\omega} \times {}^B\boldsymbol{I}\vec{\omega})_y$ refers to the y-component of the resulting cross product. Solving for $\Delta\lambda_y$ gives (2.29).

$$
\Delta\lambda_y = \frac{8}{\rho L_{prop}^3 S_{prop} C_T \cos(45^\circ)}\left(2I_{yy}\frac{\theta_c - \omega\Delta t}{(\Delta t)^2} - \left(\vec{\omega} \times {}^B\boldsymbol{I}\vec{\omega}\right)_y \tag{2.29}
$$

The result of (2.29) is used to set each propeller λ without changing the result of λ_{alt}.

$$
\begin{aligned}
\lambda_{1,y} &= 0.25\Delta\lambda_y \\
\lambda_{2,y} &= -\lambda_{1,y} \\
\lambda_{3,y} &= \lambda_{2,y} \\
\lambda_{4,y} &= \lambda_{1,y}
\end{aligned} \tag{2.30}
$$

If there are no desired changes in pitch, the vector containing all values of λ are set as follows:

$$\lambda = \lambda_{alt} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} \lambda_{1,y} \\ \lambda_{2,y} \\ \lambda_{3,y} \\ \lambda_{4,y} \end{bmatrix} \tag{2.31}$$

Deriving $\Delta\lambda_x$ follows the same procedure for $\Delta\lambda_y$. Replace each y subscript with x in (2.26) through (2.29) and the steps are identical. Unpacking the result of $\Delta\lambda_x$ follows the following format:

$$\begin{aligned} \lambda_{1,x} &= 0.25\Delta\lambda_x \\ \lambda_{2,x} &= \lambda_{1,x} \\ \lambda_{3,x} &= -\lambda_{1,x} \\ \lambda_{4,x} &= \lambda_{3,x} \end{aligned} \tag{2.32}$$

For any combination of roll and pitch, λ for each propeller is then found by summing $\lambda_{i,x}$ and $\lambda_{i,y}$ as shown in (2.33).

$$\lambda = \lambda_{alt} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} \lambda_{1,y} \\ \lambda_{2,y} \\ \lambda_{3,y} \\ \lambda_{4,y} \end{bmatrix} + \begin{bmatrix} \lambda_{1,x} \\ \lambda_{2,x} \\ \lambda_{3,x} \\ \lambda_{4,x} \end{bmatrix} \tag{2.33}$$

2.3 UAV 6DOF Simulation

Using the dynamics outlined in Chapter 2.1 to simulate UAV flight and the controls outlined in Chapter 2.2, a six degree-of-freedom (6DOF) simulation was constructed in MATLAB. UAV physical parameters are set using "initialize_uav.m" which is provided in Appendix D. The simulation records the following states: UAV NED position ${}^{I}\vec{r}_{UAV}$; body velocity ${}^{B}\vec{v}$; angular velocity $\vec{\omega}$; Euler angles ϕ, θ, and ψ; obstacle position ${}^{I}\vec{\chi}$; and source position relative to the UAV ${}^{I}\vec{r}_{source}$. A basic obstacle position tracking is supported, but there is no UAV logic implemented to avoid obstacles. These simulated values were used as truth to construct the Extended Kalman Filter and radioactive source position estimates.

$$\begin{bmatrix} x = {}^I\vec{r}_{UAV} & {}^B\vec{v} & \vec{\omega} & \phi & \theta & \psi & {}^I\vec{\chi} & {}^I\vec{r}_{source} \end{bmatrix} \tag{2.34}$$

The UAV can receive inputs to command translational motion via two inputs. The first, by presetting waypoints which are passed into the simulation. Waypoints take form of a column vector where there first element contains the time for the command to take place and elements two through four contain the desired NED position. An example waypoint is shown in (2.35).

$$\begin{bmatrix} t_1 \\ r_{North} \\ r_{East} \\ r_{Down} \end{bmatrix} \tag{2.35}$$

Alternatively, the UAV can be controlled using keyboard inputs outlined in Table 2.1. Setting the value of "showRealtimePlot = 1" will provide realtime display of the UAV. Waypoint control and keyboard control can be used simultaneously with keyboard control overriding waypoint control. The MATLAB code which runs the simulation is provided in Appendix A. Simulation results are shown in Chapter 3.1.

Table 2.1. Keyboard control inputs for UAV simulation.

Key	Command
w	Translate along $+b_x$ direction
s	Translate along $-b_x$ direction
d	Translate along $+b_y$ direction
a	Translate along $-b_y$ direction

2.4 UAV Sensors

Consumer-grade quadcopters utilize several instruments for position and attitude information throughout a flight. Following the project goals, four low-cost, well documented sensors were chosen and their specifications used in the creation of sensor models. Table 2.2 provides the model number and quantity measured for each instrument.

Table 2.2. Instrument models and quantities measured.

Instrument	Model	Quantity Measured
GPS	GlobalTop MTK3339	Position and Velocity
Altimeter	Bosch BMP388	Altitude
LIDAR	Garmin Lidar-Lite V3	Body-X Obstacle position
IMU	Bosch BNO055	Acceleration and angular velocity

A brief discussion on the pros and cons of each instrument is presented below.

2.4.1 GPS

Global positioning system (GPS) uses a minimum of five satellites to provide a reliable position, velocity, and time solution. Position measurements triangulate your position using three satellites, while velocity measurements use the Doppler shift of the incoming transmission to determine velocity at a much higher accuracy.

2.4.1.1 Pros

- High accuracy. Position accuracy of ± 3m and velocity accuracy of ± 0.1m.
- Not typically prone to bias or random walk

2.4.1.2 Cons

- Requires satellite signal, severely limiting suitable environments.
- Low frequency. Typical sample rates of 1Hz.

2.4.2 Altimeter

An altimeter compares the pressure difference between a reference pressure and the current pressure to determine altitude. If altitude relative to sea level (true altitude) is known throughout the flightpath, an altimeter provides high accuracy in its altitude measurements.

2.4.2.1 Pros

- High accuracy. Typical accuracy of about $\pm$ 0.5m.
- High frequency of measurements. 50 - 90 Hz can be expected.
- Lightweight

2.4.2.2 Cons

- Requires ground level true altitude to be known throughout flightpath.

2.4.3 LIDAR

LIDAR (LIght Detection And Ranging) sensors operate by transmitting a laser and measuring the duration between laser transmission and detection. With the index of refraction known through the medium at which the sensor is operating within, the distance measured by the LIDAR sensor is given in (2.36) where c is the speed of light in a vacuum, n is the index of refraction of the medium, and Δt is the time between laser transmission and detection.

$$\Delta r = \frac{c}{2n} \Delta t \tag{2.36}$$

LIDAR sensors are used in a variety of applications such as mapping terrain for a particular region. The purpose of a LIDAR sensor in this project was for obstacle detection. While obstacle avoidance has not been implemented in this project, the addition of LIDAR models and outputs allows for future inclusion.

2.4.3.1 Pros

- High frequency
- High accuracy

2.4.3.2 Cons

- Expensive

2.4.4 IMU (Inertial Measurement Unit)

The final instrument used is an inertial measurement unit (IMU). An IMU contains at minimum a three-axis accelerometer and a 3-axis gyroscope while newer models may contain magnetometers. An IMU is capable of providing position and attitude information at high frequencies. This information, however, is prone to 'drift' from the true values. Determining velocity from acceleration requires integration from the accelerometer measurement. Any errors in the accelerometer measurement will accumulate when integrated for the velocity measurement. This results in an error accumulation that is linear for velocity and quadratic for position. The same issue is present when determining Euler angles from gyroscope measurements. Position and velocity drift is solved by fusing the GPS and altimeter measurements with the IMU measurements. Only the problem of attitude drift remains.

When the accelerometer measures the magnitude of acceleration to be +1g, the accelerometer can be used as a drift-free method to compute roll and pitch angles. An accelerometer measures proper acceleration, or more simply, acceleration relative to free-fall. When the magnitude of measured acceleration is equal to the local acceleration due to gravity, the components of acceleration can be used to determine UAV roll and pitch [**pedley_tilt_2013**, 13, 6]. Figure 2.5 illustrates the measured acceleration for a fixed accelerometer which has been tilted with roll angle ϕ and pitch angle θ. Recall that the acceleration is relative to free-fall, which results in a measured acceleration of +1g when all forces are equal.

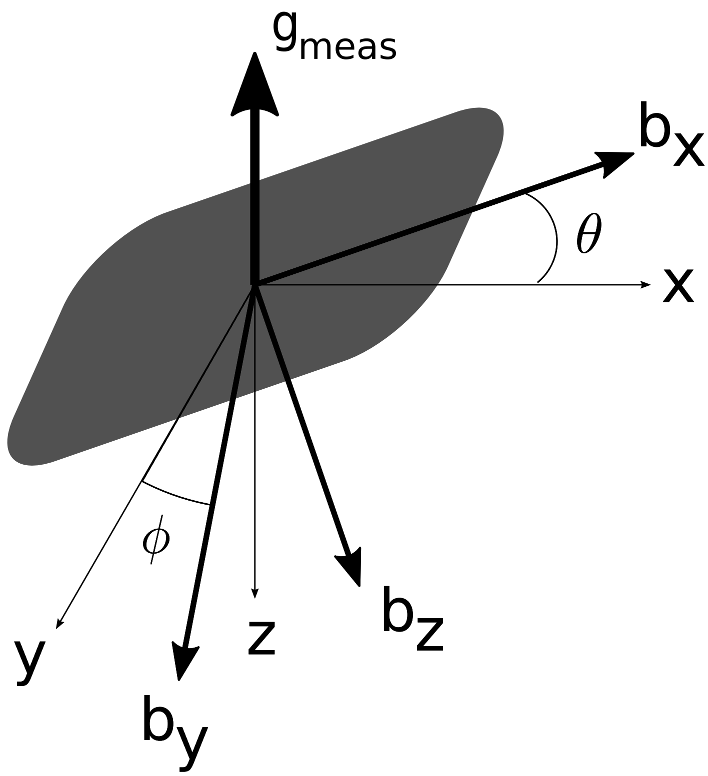


Figure 2.5. Accelerometer measurement with zero net force.

Equation 2.37 provides the trigonometric expressions for computing roll and pitch when the accelerometer is subject to zero net force. While onboard a UAV, this will typically occur during a stable hover.

$$
\begin{aligned}
\phi &= \arctan\left(\frac{a_y}{a_z}\right) \\
\theta &= \arctan\left(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}\right)
\end{aligned}
\tag{2.37}
$$

2.4.4.1 Pros

- High frequency (Ĩ00 Hz)
- Low cost
- Small form factor

- Capable of producing position and attitude measurements.

2.4.4.2 Cons

- Subject to drift

- Typically will have some form of bias that needs to be estimated.

2.5 Extended Kalman Filter

An Extended Kalman filter (EKF) was created to estimate the true UAV states throughout the duration of a flight. The regular Kalman filter produces an optimal estimate of states described by a linear system [10]. Kalman filters follow a two step process, the first, a time-update step which propagates the state from the $k-1$ timestep to timestep k. The second, a measurement-update step which improves the state estimate using available measurements. The EKF extends the Kalman filter for nonlinear systems.

For a UAV, the state vector will typically be composed of attitude and position vectors along with their derivatives. The state vector for this EKF is given in (2.38). All vectors in x are relative to the NED frame. The definitions of all variables are presented in Table 2.3.

$$x = \begin{bmatrix} {}^I\vec{r} & {}^I\vec{v} & {}^I\vec{a} & \phi & \theta & \psi & {}^I\vec{\chi} & {}^I\vec{s} \end{bmatrix} \tag{2.38}$$

Table 2.3. Kalman state vector composition.

Variable	Definition
${}^I\vec{r}$	UAV NED Position Vector
${}^I\vec{v}$	UAV NED Velocity Vector
${}^I\vec{a}$	UAV NED Acceleration Vector
ϕ	UAV Roll
θ	UAV Pitch
ψ	UAV Yaw
${}^I\vec{\chi}$	UAV NED Obstacle Position
${}^I\vec{s}$	NED Estimated Source Position

The EKF receives measurements from the following types of instruments:

- GPS
- Altimeter
- LIDAR
- Inertial measurement unit (IMU)

Using the instruments listed previously in Table 2.2, the measurement vector y_k was then defined as shown in (2.39). If a particular instrument is not available at time k, it is not included in y_k.

$$y_k = \begin{bmatrix} y_{\text{GPS}} & y_{\text{altimeter}} & y_{\text{lidar}} & y_{\text{IMU}} \end{bmatrix} \tag{2.39}$$

Adopting the convention in [17], the EKF system and measurement equations are presented in (2.40) – (2.43). Variable definitions are given in Table 2.4. The notation used in (2.42) defines w_k as a random variable with a mean of 0 and covariance $\boldsymbol{Q}_k$.

$$x_k = f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.40}$$

$$y_k = h_k(x_k, \nu_k) \tag{2.41}$$

$$w_k \sim (0, \boldsymbol{Q}_k) \tag{2.42}$$

$$v_k \sim (0, \boldsymbol{R}_k) \tag{2.43}$$

All EKF code is provided in Appendix C.

2.5.1 Time Update

The time-update step produces an *a priori* state estimate $\hat{x}_k^-$ and error covariance $\boldsymbol{P}_k^-$ for time k from time $k-1$ as shown in (2.44) and (2.45).

$$\hat{x}_k^- = f_{k-1}(\hat{x}_{k-1}^+, u_{k-1}, 0) \tag{2.44}$$

Table 2.4. Variable definitions for the Extended Kalman Filter.

Variable	Definition
x_k	True state estimate at time k
$\hat{x}_k^-$	*a priori* state estimate
$\hat{x}_k^+$	*a posteriori* state estimate
f_{k-1}	System update function to update state from x_{k-1} to x_k
u	System input vector
w_{k-1}	System noise
y_k	Measurement vector
h_k	Measurement function
ν_k	Measurement noise
$\boldsymbol{Q}_k$	System noise covariance
$\boldsymbol{R}_k$	Measurement noise covariance
$\boldsymbol{P}_k^-$	*a priori* Estimation error covariance matrix
$\boldsymbol{P}_k^+$	*a posteriori* Estimation error covariance matrix
$\boldsymbol{F}_{k-1}$	System-update matrix
$\boldsymbol{H}_k$	measurement matrix
$\boldsymbol{K}_k$	Kalman gain matrix

$$\boldsymbol{P}_k^- = \boldsymbol{F}_{k-1}\boldsymbol{P}_{k-1}^+\boldsymbol{F}_{k-1}^T + \boldsymbol{Q}_{k-1} \tag{2.45}$$

The system update function shown in (2.44) takes inputs from the previous *a posteriori* state estimate and the gyroscope (designated as u_{k-1}). The gyroscope angular rates are used to compute Euler rates and Euler angles. The time-update equations performed in f_{k-1} are given in (2.46).

$$\begin{aligned}
\vec{r}_k &= \vec{r}_{k-1} + \vec{v}_{k-1}\Delta t + 0.5\vec{a}_{k-1}(\Delta t)^2 \\
\vec{v}_k &= \vec{v}_{k-1} + \vec{a}_{k-1}\Delta t \\
\vec{a}_k &= \begin{bmatrix} (-{}_{\boldsymbol{I}}\boldsymbol{T}_{\boldsymbol{B}}\frac{g}{\cos\phi\cos\theta})_x & (-{}_{\boldsymbol{I}}\boldsymbol{T}_{\boldsymbol{B}}\frac{g}{\cos\phi\cos\theta})_y & a_{z,k-1} \end{bmatrix} \\
\phi_k &= \phi_{k-1} + \dot{\phi}_{k-1}\Delta t \\
\theta_k &= \theta_{k-1} + \dot{\theta}_{k-1}\Delta t \\
\psi_k &= \psi_{k-1} + \dot{\psi}_{k-1}\Delta t \\
\vec{\chi}_k &= \chi_{k-1} - \vec{v}_{k-1}\Delta t
\end{aligned} \tag{2.46}$$

All terms other than $\vec{a}_k$ follow elementary kinematics. The $\vec{a}_k$ term estimates horizontal acceleration to be a function of the roll and pitch angle with no vertical acceleration. Vertical acceleration remains constant during the time-update step.

The acceleration term does not hold true when the vertical acceleration is nonzero. The horizontal terms will produce adequate estimates which are updated from the accelerometer and GPS. The vertical term, however, relies entirely on the altimeter, accelerometer, and GPS. The high sample rates of the altimeter and accelerometer provide reliable estimates of vertical acceleration.

The second part of the time-update step is to update the error covariance of the state using (2.45). The system-update matrix $\boldsymbol{F}_{k-1}$ is computed by taking the Jacobian of f_{k-1} evaluated at the last state estimate as shown in (2.47).

$$\boldsymbol{F}_{k-1} = \frac{\partial f_{k-1}}{\partial x}\bigg|_{x_{k-1}^{+}} \tag{2.47}$$

Term $\boldsymbol{Q}_{k-1}$ represents process noise in the system. The noise can come from numerous sources, but represents inaccuracies in the system-update function. For example, the model does nothing to predict variations in wind which may occur during a standard flight. A well-tuned process noise matrix helps fuse the time-update step with the measurement-update step. Low elements in $\boldsymbol{Q}$ favor the model while high $\boldsymbol{Q}$ values indicate low reliability in the model and have the filter favor the measurements.

2.5.2 Measurement Update

The measurement-update step in the EKF takes all measurements available at time k and updates the *a priori* state estimate $\hat{x}_k^-$ to the *a posteriori* state estimate $\hat{x}_k^+$. If any measurements are available, the Kalman gain (2.48), *a posteriori* state estimate (2.49), and *a posteriori* error covariance (2.50) are computed.

$$\boldsymbol{K}_k = \boldsymbol{P}_k^- \boldsymbol{H}_K^T (\boldsymbol{H}_k \boldsymbol{P}_k^- \boldsymbol{H}_k^T + \boldsymbol{R}_k)^{-1} \tag{2.48}$$

$$\hat{x}_k^+ = \hat{x}_k^- + \boldsymbol{K}_k (y_k - h_k(\hat{x}_k^-)) \tag{2.49}$$

$$\boldsymbol{P}_k^+ = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_k^- \tag{2.50}$$

To compute the Kalman gain, two matrices must be constructed in advance, the measurement matrix ($\boldsymbol{H}_k$) and the measurement noise covariance ($\boldsymbol{R}_k$). Similar to $\boldsymbol{F}_{k-1}$, the measurement matrix (2.52) is the Jacobian of the measurement function which outputs measurements from the state vector. Equation 2.51 shows each output term in the measurement function $h_k(\hat{x}_k^-, 0)$. Note that in h_{IMU}, roll and pitch are only used if the UAV is in a stable hover (see Chapter 2.4.4).

$$\begin{aligned}
h_{\text{gps}} &= \begin{bmatrix} {}^I\vec{r} & {}^I\vec{v}_{x,y} \end{bmatrix} \\
h_{\text{altimeter}} &= -{}^I\vec{r}_z \\
h_{\text{lidar}} &= ({}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{I}}{}^I\vec{\chi})_x \\
h_{\text{IMU}} &= \begin{bmatrix} {}_{\boldsymbol{B}}\boldsymbol{T}_{\boldsymbol{I}}{}^I\vec{a} & \phi & \theta \end{bmatrix} \\
h_k &= \begin{bmatrix} h_{\text{gps}} & h_{\text{altimeter}} & h_{\text{lidar}} & h_{\text{IMU}} \end{bmatrix}
\end{aligned} \tag{2.51}$$

$$\boldsymbol{H}_k = \frac{\partial h_k}{\partial x}\bigg|_{x_k^-} \tag{2.52}$$

The measurement covariance matrix $\boldsymbol{R}_k$ remains constant between timesteps. This matrix was built using specifications for each instrument listed in Table 2.2.

2.6 Radioactive Source Localization

Following the completion of the 6DOF simulation and the EKF, an algorithm estimating the position of a radioactive source was created. The measurements received were assumed to take the form of counts where a desired count rate could be determined.

This allows for measurements from a traditional (and low cost) Geiger counter as well as any consumer-grade CCD [4].

Measurements from a count-based device can be modelled as a Poisson distribution. One of the challenges in working with such a device is that the standard deviation is equal to the square root of the number of counts recorded (2.53). As such, longer recording times produce favorable measurements. Measurement duration, is limited by UAV maximum flight time.

$$\sigma = \sqrt{A} \tag{2.53}$$

The general idea of the algorithm is to construct a gradient by recording measurements at six different points in space for a uniform duration as shown in Figure 2.6. The UAV then flies to the region with the greatest gradient. Recursively running the algorithm allows the UAV to converge on the radioactive source. The decay rate near the source (A_0) is assumed to be approximately known.

When the UAV is commanded to estimate radioactive source position, the following steps are taken:

1. Record current position as the gradient center.

2. Fly one meter north of the gradient center.

3. Hold position for 30 seconds to record counts.

4. Repeat steps 2 and 3 flying south, east, west, down, and up.

5. Compute estimated source position.

6. Fly to estimated source position and return to step 1.

The distance flown and measurement duration listed may be varied before running the simulation. Note that the UAV will never be commended to fly lower than 10cm above the ground.

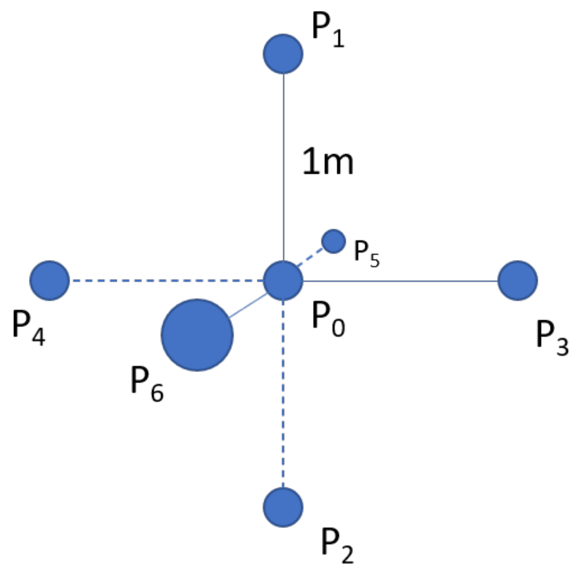


Figure 2.6. Localization algorithm measurement points. The UAV begins at P_0 and flies to P_1 - P_6, stopping at each point to measure radioactivity for 30 seconds. Note that P_5 and P_6 are into the page and out of the page respectively.

To trigger the algorithm, the background activity should be known and an activity threshold must be set. If the last full second of counts is above the threshold, the position estimation algorithm is triggered. To minimize background radioactivity triggering the estimation algorithm, the activity threshold was set such that the probability of background radiation being greater than or equal to the activity threshold was nearly zero. In all test cases presented in Chapter 3.3, activity threshold is set to five counts per second and background is set to one count per second. The probability of background exceeding the five counts per second was determined using (2.54) to be 0.37%.

$$P(k > \lambda) = \sum_{k}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \tag{2.54}$$

The source localization code is presented in Appendix E.

2.6.1 Estimation Logic

For some distance d away from a source, (2.55) can be used to compute the distance from a radioactive source. A_0 is the known activity at distance d_0 and A is the source activity recorded within the timespan (30 seconds by default).

$$d = d_0\sqrt{\frac{A_0}{A}} \tag{2.55}$$

Each axis is observed separately. Recall that two sets of data are taken along each axis. Thus we have two values for d: d_- and d_+. Additionally, the activity at both locations can be used to determine which direction the source lies. If $A_+ > A_-$, then the source must lie in the positive direction. The estimated distance along this axis is determined by the difference between d_- and d_+ which is defined as Δd. There are two scenarios to examine. The first, and most common, occurs when A_+ and A_- are less than A_0. The second, when a measured activity is greater than A_0. All possible outcomes are presented below. Note r_i represents the ith axis where i is either the north axis, east axis, or down axis. All outcomes listed below also require (2.56) to be satisfied. If this condition is not met, both measured activities fall within the same standard deviation. This would permit only small or zero UAV motion.

$$\begin{aligned} |\Delta A_{+,-}| &> \sqrt{\min(A_+, A_-)} \\ |\Delta A_{+,-}| &> \min(\sigma_+, \sigma_-) \end{aligned} \tag{2.56}$$

$\boldsymbol{A_0 \geq (A_-, A_+)}$ **&** $\boldsymbol{\Delta d > 0.9m}$**:** Under these conditions, the source is significantly closer to one measurement location than the other. As such, the measurement with more counts is used to determine source distance. Figure 2.7 illustrates an example of when this may occur. Equation 2.57 shows the equation used if $A_+ > A_-$. If $A_- > A_+$, then replace d_+ with d_-.

$$r_i = \text{sign}(\Delta A)d_+ \tag{2.57}$$

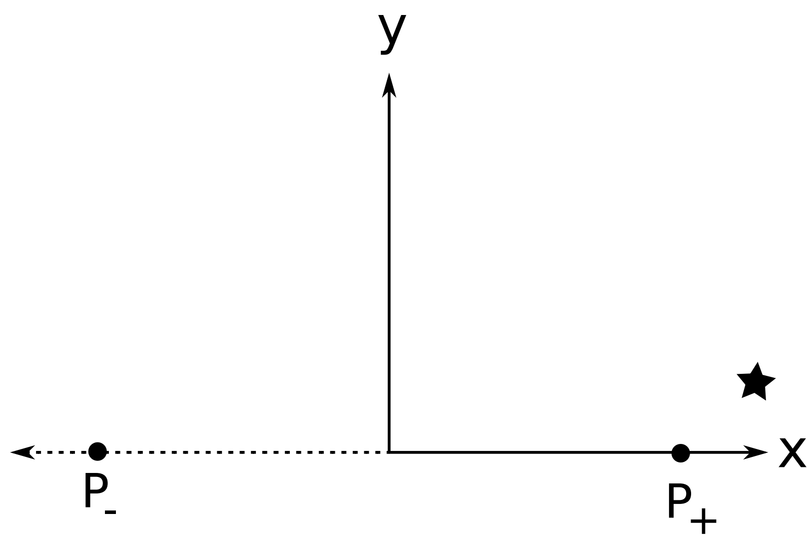


Figure 2.7. Localization algorithm logic for $A_0 \geq (A_-, A_+)$ & $\Delta d > 0.9m$. Measurement points represented by P_+ and P_-. The star shows the relative source location.

$\boldsymbol{A_0 \geq (A_-, A_+)}$ **&** $\boldsymbol{0.5m < \Delta d \leq 0.9m}$**:** This condition is likely to occur when the source does not lie close to a single measurement point, but lies near the line segment between the two points as shown in Figure 2.8. Equation 2.58 shows the equation used if $A_+ > A_-$.

$$r_i = \text{sign}(\Delta d)|d_+ - 1\text{m}| \tag{2.58}$$

$\boldsymbol{A_0 \geq (A_-, A_+)}$ **&** $\boldsymbol{\Delta d \leq 0.5m}$**:** This condition is likely to occur when the source lies near the gradient center. See Figure 2.9 for an illustration.

$$r_i = \text{sign}(\Delta d)\frac{|d_+ - d_-|}{2} \tag{2.59}$$

$\boldsymbol{(A_0 - \sqrt{A_0}) < \max(A_+, A_-)}$ **&** $\boldsymbol{\Delta d \geq 1.5}$**:** If this condition is met, the result is that one of the measurements is greater than A_0 meaning it is actually closer to the source than r_0. The UAV disregards the lower activity measurement and flies to the point where the measurement was taken. Figure 2.10 provides an example of this scenario.

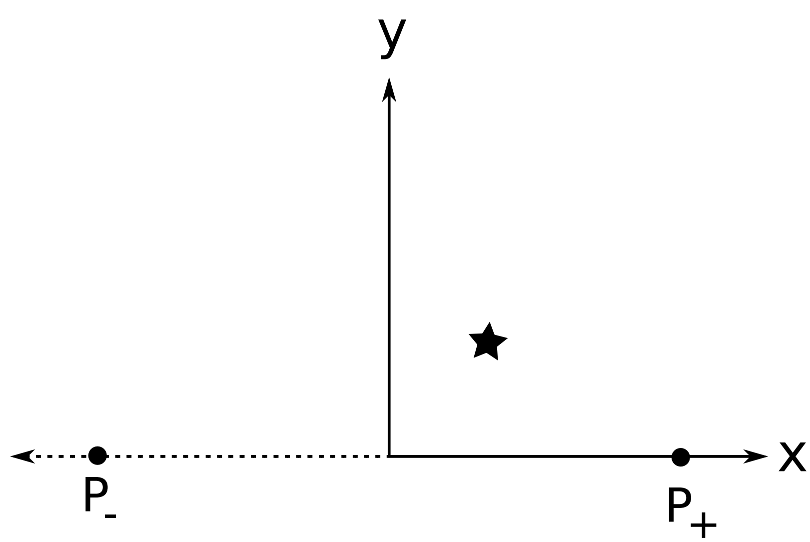


Figure 2.8. Localization algorithm logic for $A_0 \geq (A_-, A_+)$ & $0.5m < \Delta d \leq 0.9m$. Measurement points represented by P_+ and P_-. The star shows the relative source location.

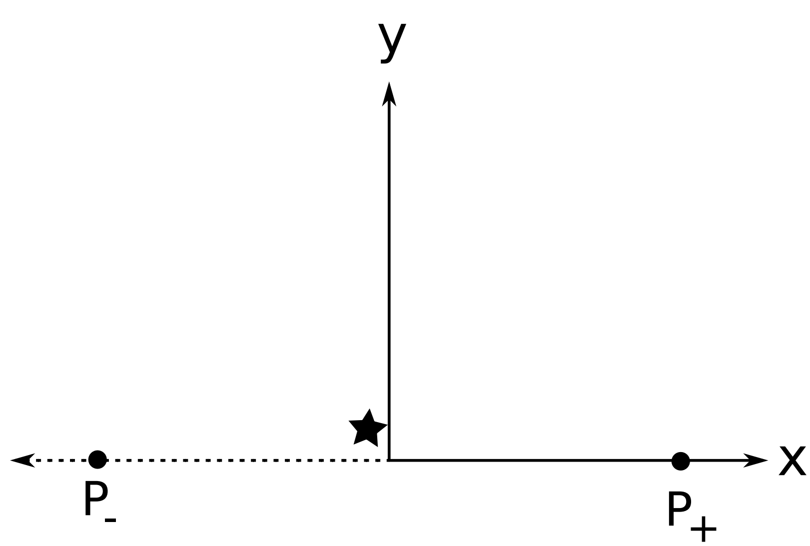


Figure 2.9. Localization algorithm logic for $A_0 \geq (A_-, A_+)$ & $\Delta d \leq 0.5m$. Measurement points represented by P_+ and P_-. The star shows the relative source location.

$$r_i = r_{max,i} - r_{center,i} \tag{2.60}$$

Once r_i has been determined from the scenarios listed above, there are checks to prevent poor measurements from compromising the algorithm. For example, (2.61) shows the maximum possible source distance, along any axis, within 1σ of A_0. If r_i is greater

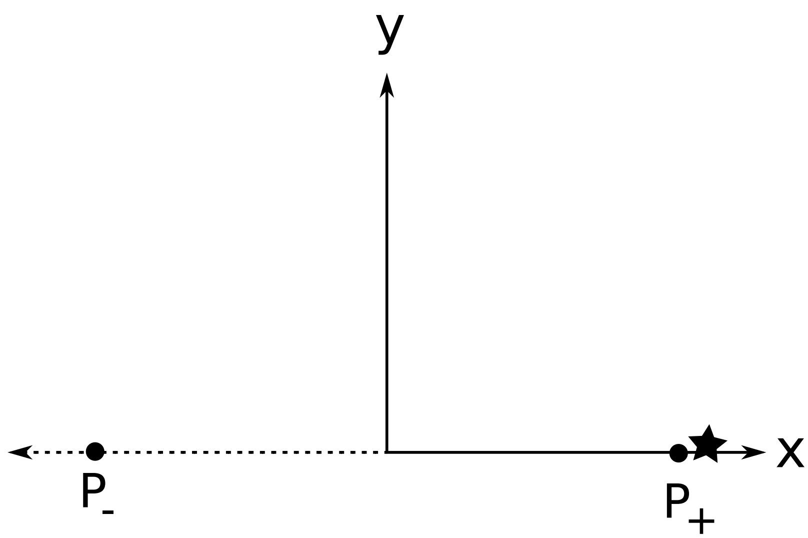


Figure 2.10. Localization algorithm logic for $(A_0 - \sigma_0) < \max(A_+, A_-)$ & $\Delta d \geq 1.5$. Measurement points represented by P_+ and P_-. The star shows the relative source location.

than r_{max}, r_i is set to zero. This is likely to occur if the source is located more than one meter along a plane orthogonal to the current axis.

$$r_{max} = r_0 \frac{\sqrt{A_0 + \sqrt{A_0}}}{A_{thresh}} \tag{2.61}$$

An additional check requires that the difference in measured activity must be greater than the minimum standard deviation as shown in (2.62). If this condition is violated, the UAV is either too far from the source to produce unique measurements or the UAV is nearly centered on the source.

$$\Delta A > \sqrt{A_{min}} \tag{2.62}$$

When r_i is set for the three axes, they are added to the center gradient position to create the new source position estimate (2.63).

$${}^{I}\vec{r}_{source} = \begin{bmatrix} r_{north} \\ r_{east} \\ r_{down} \end{bmatrix} + {}^{I}\vec{r}_{center} \tag{2.63}$$

Figure 2.11 provides a block diagram of the algorithm logic.

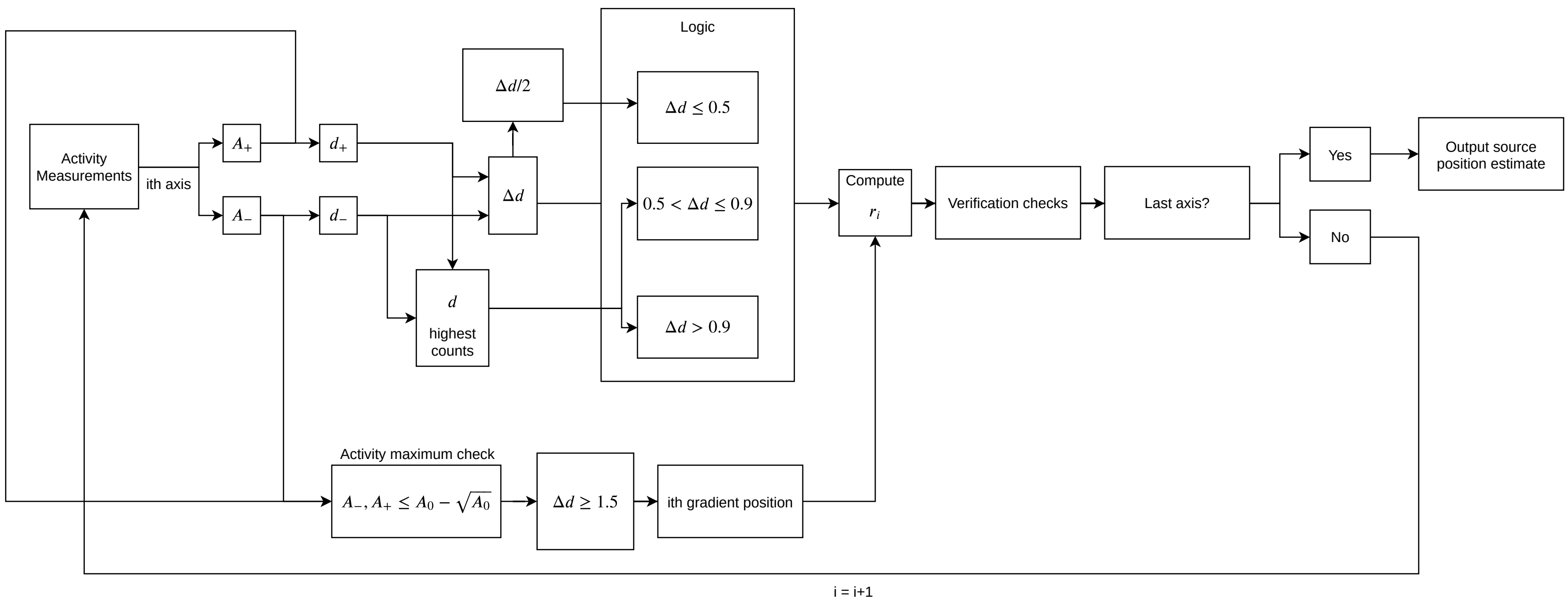


Figure 2.11. Block diagram illustrating radioactive source localization logic.

CHAPTER 3

RESULTS

3.1 Simulation

The results listed below include simulation outputs only. The first set of plots use waypoints to command the UAV. The second set comes from a simulation run where keyboard inputs were used to command the UAV arbitrarily.

3.1.1 Waypoint Flight

The UAV was commanded to fly to the waypoints listed in Table 3.1. Figures 3.1 - 3.8 illustrate the UAV flight dynamics.

Table 3.1. UAV commanded waypoints

Time (s)	North (m)	East (m)	Down (m)
0	0	0	-1
10	5	5	-5
30	-10	10	-0.2
50	0	0	-10
100	20	-20	-1

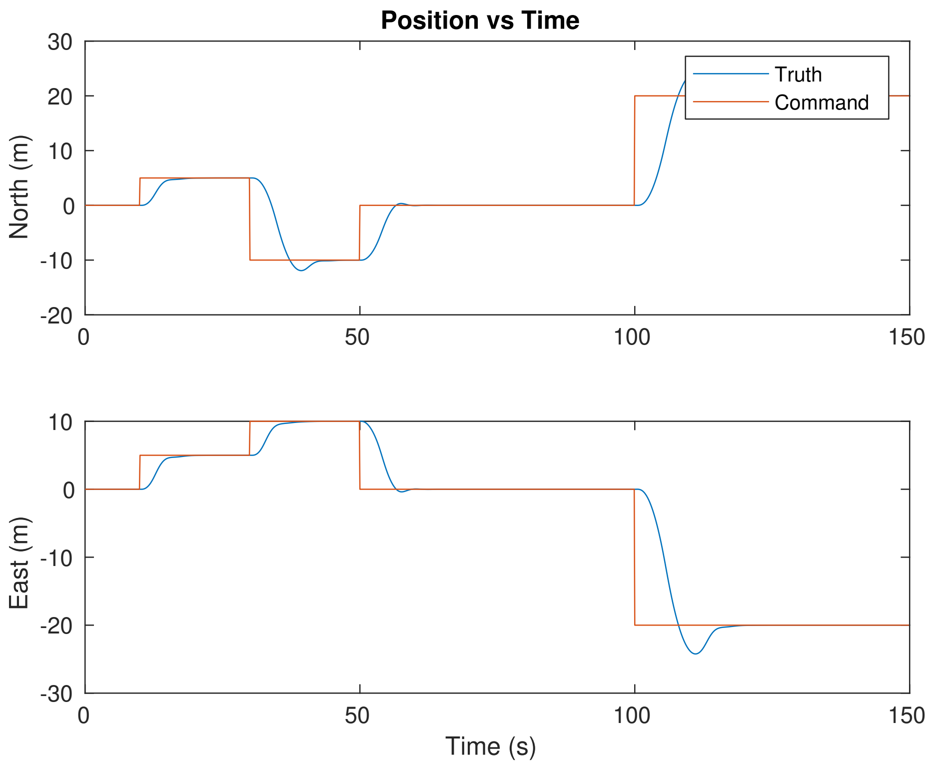


Figure 3.1. Waypoint flight - UAV position vs time

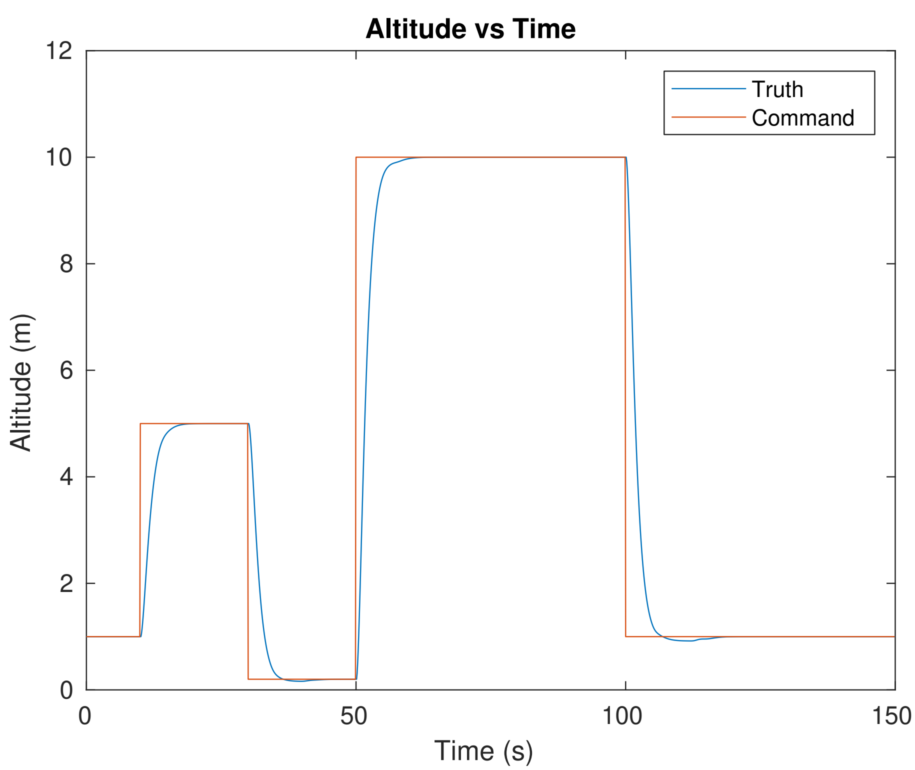


Figure 3.2. Waypoint flight - UAV altitude vs time

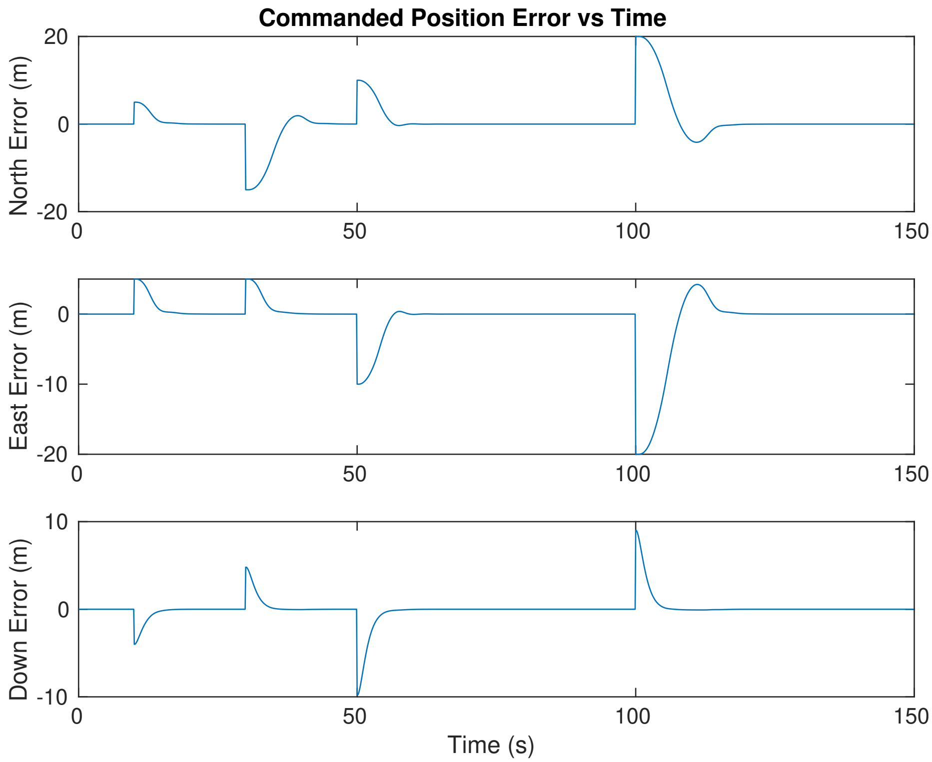


Figure 3.3. Waypoint flight - UAV commanded position error vs time

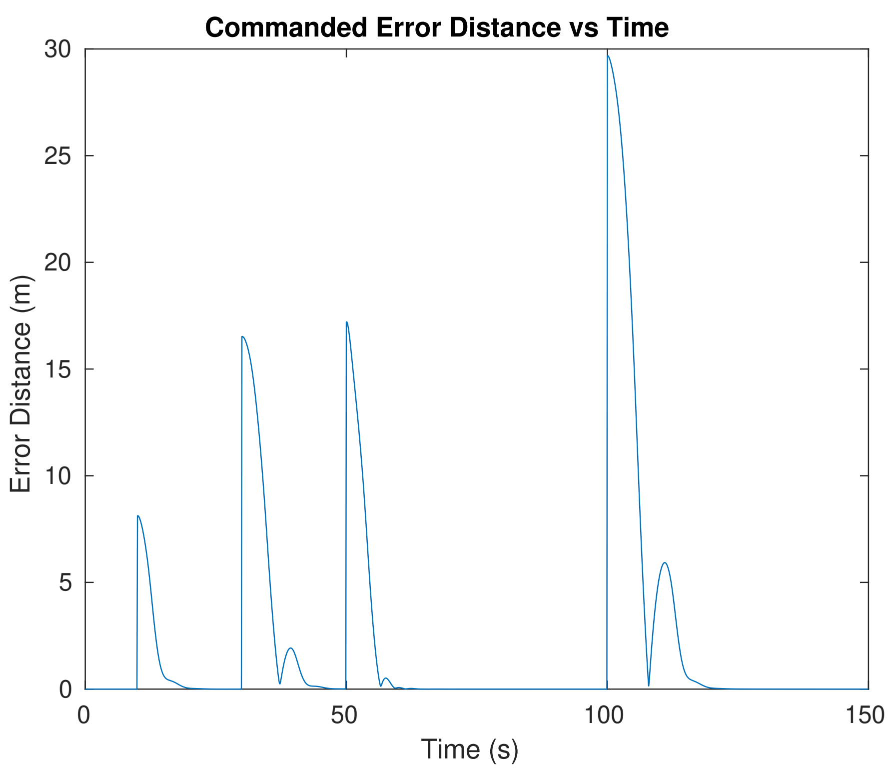


Figure 3.4. Waypoint flight - UAV commanded distance error vs time

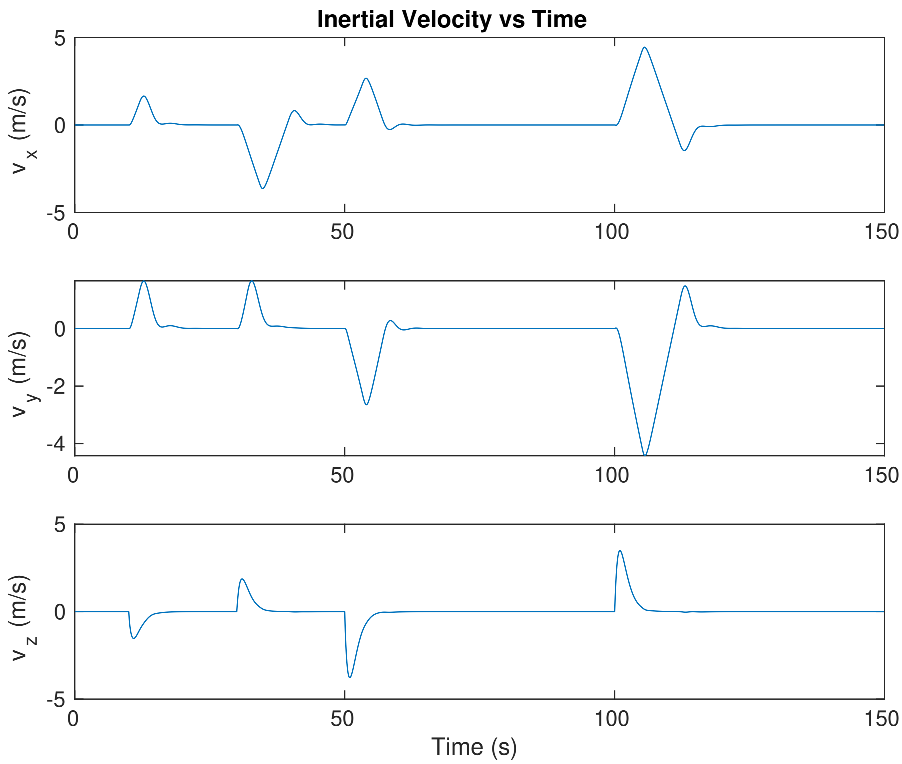


Figure 3.5. Waypoint flight - velocity vs time

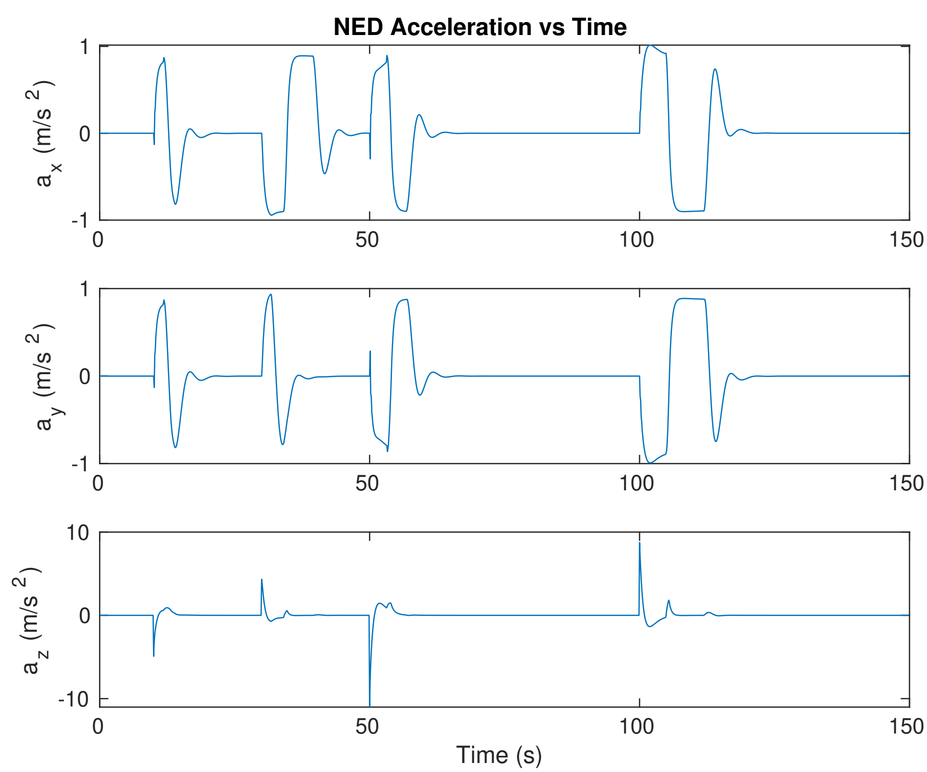


Figure 3.6. Waypoint flight - acceleration vs time

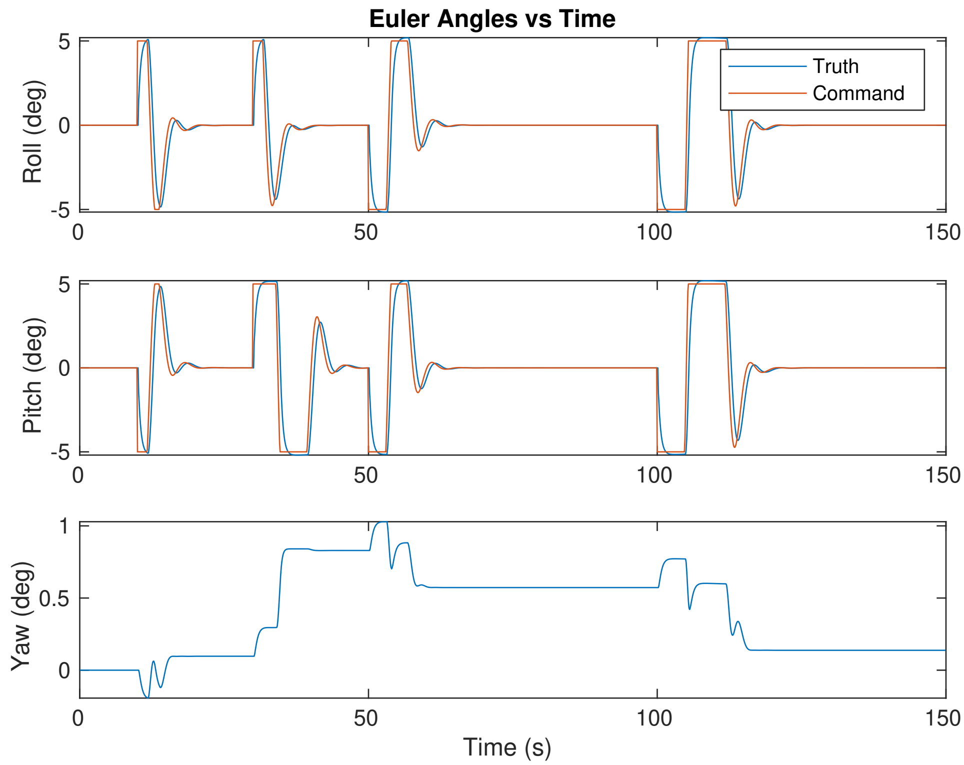


Figure 3.7. Waypoint flight - Euler angles vs time

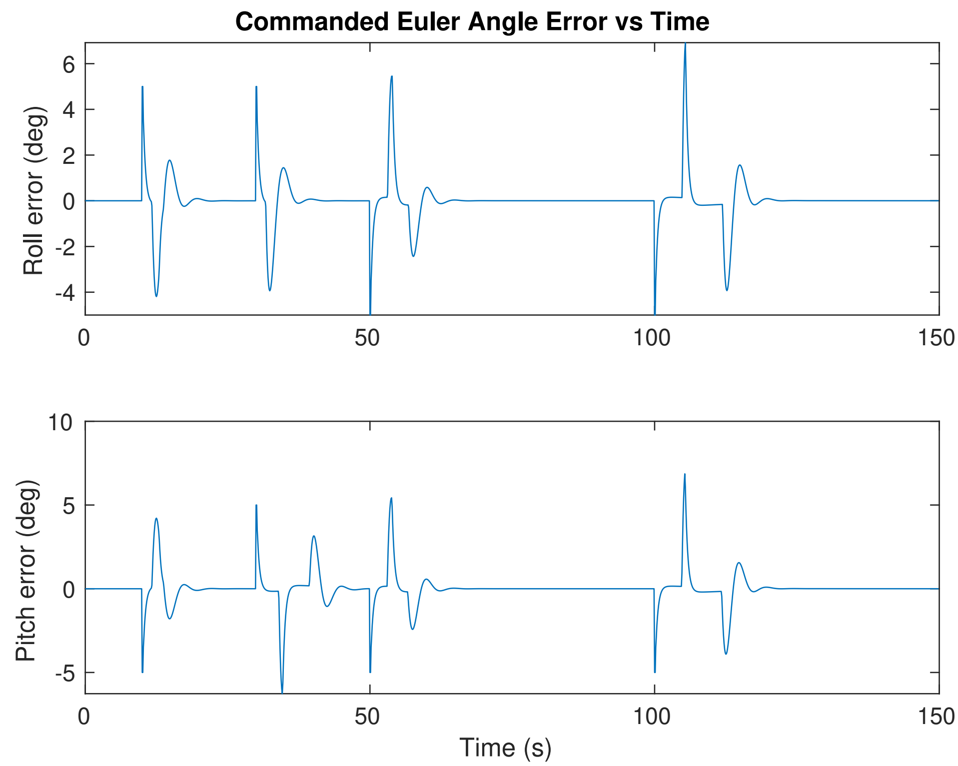


Figure 3.8. Waypoint flight - commanded Euler angle error vs time

3.1.2 Manual Flight

Figures 3.9 - 3.14 illustrate the simulated flight dynamics when a user is controlling the UAV via realtime keyboard inputs.

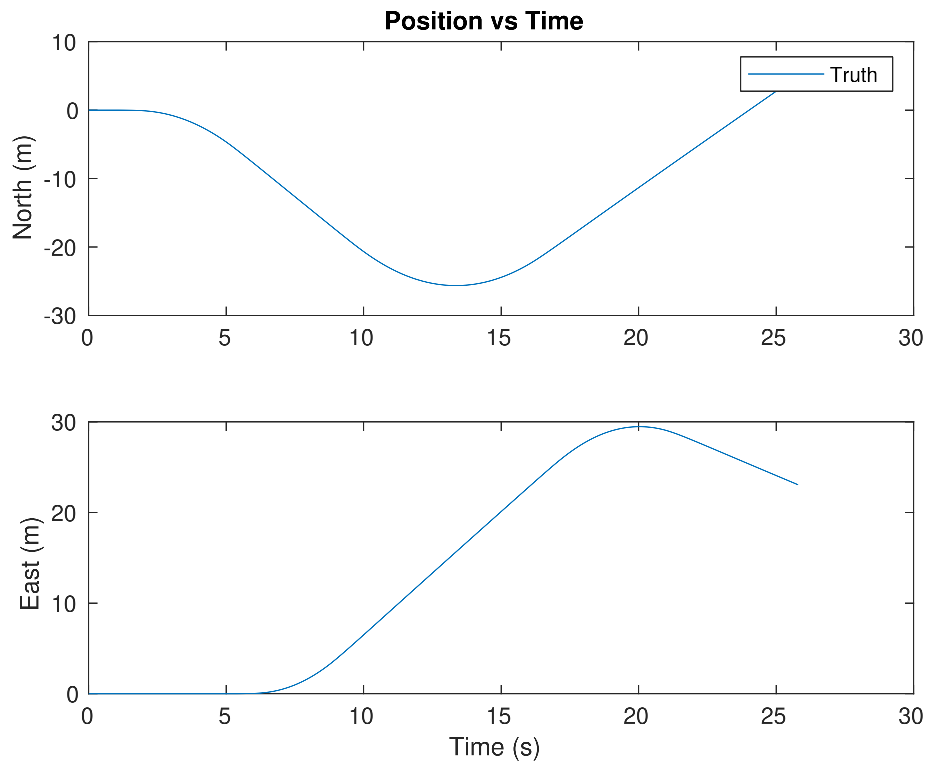


Figure 3.9. Manual flight - UAV position vs time

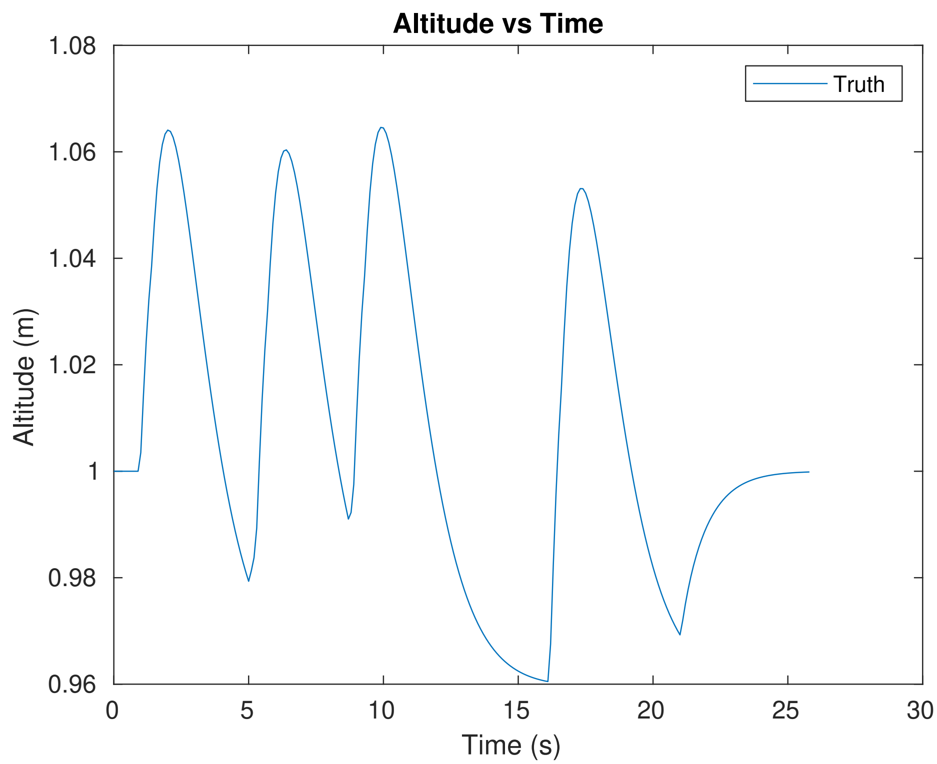


Figure 3.10. Manual flight - UAV altitude vs time

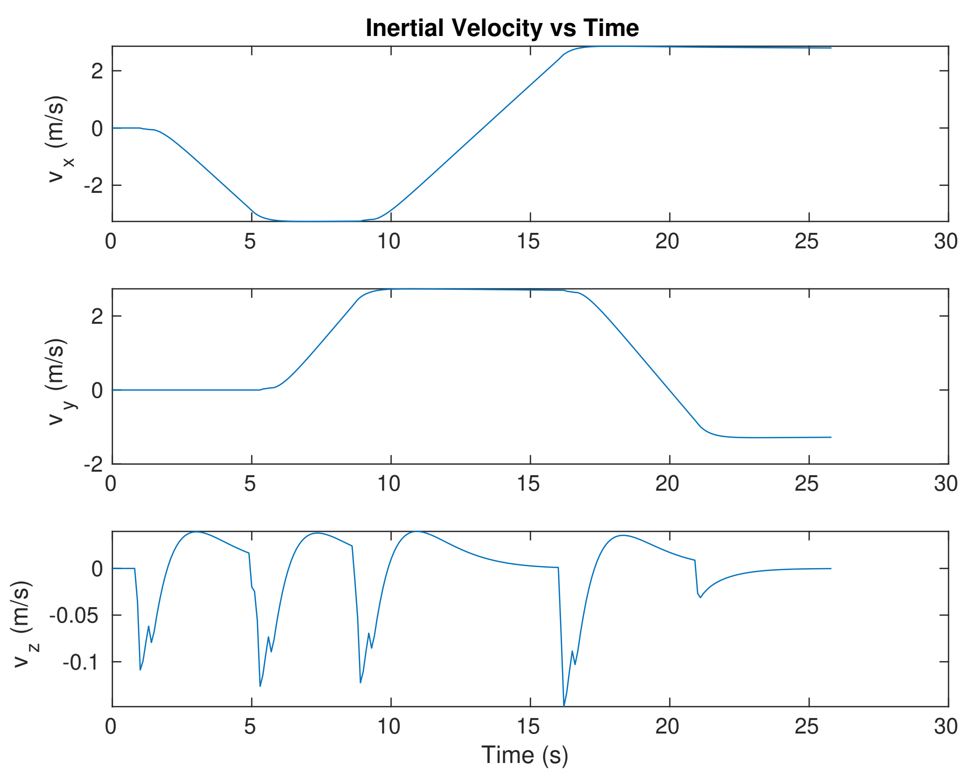


Figure 3.11. Manual flight - velocity vs time

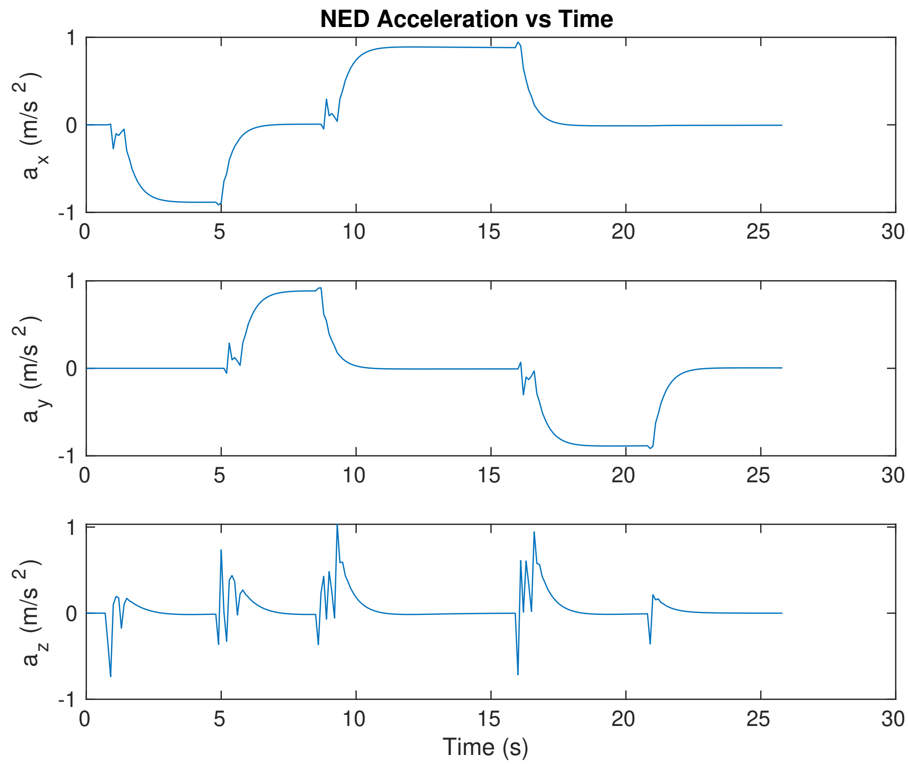


Figure 3.12. Manual flight - acceleration vs time

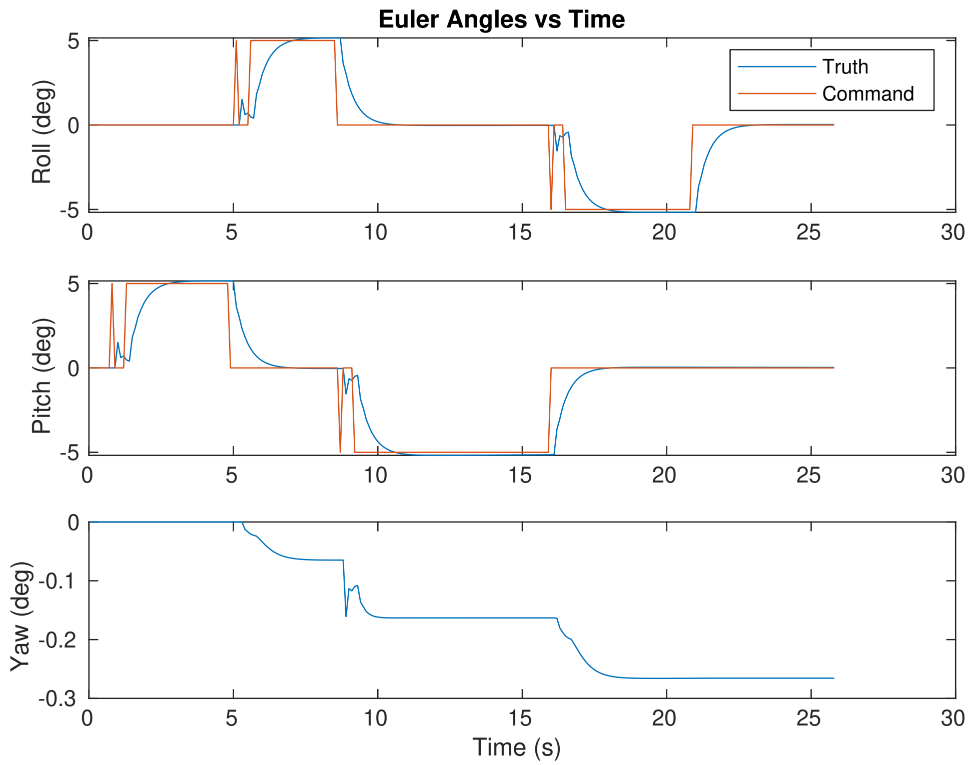


Figure 3.13. Manual flight - Euler angles vs time

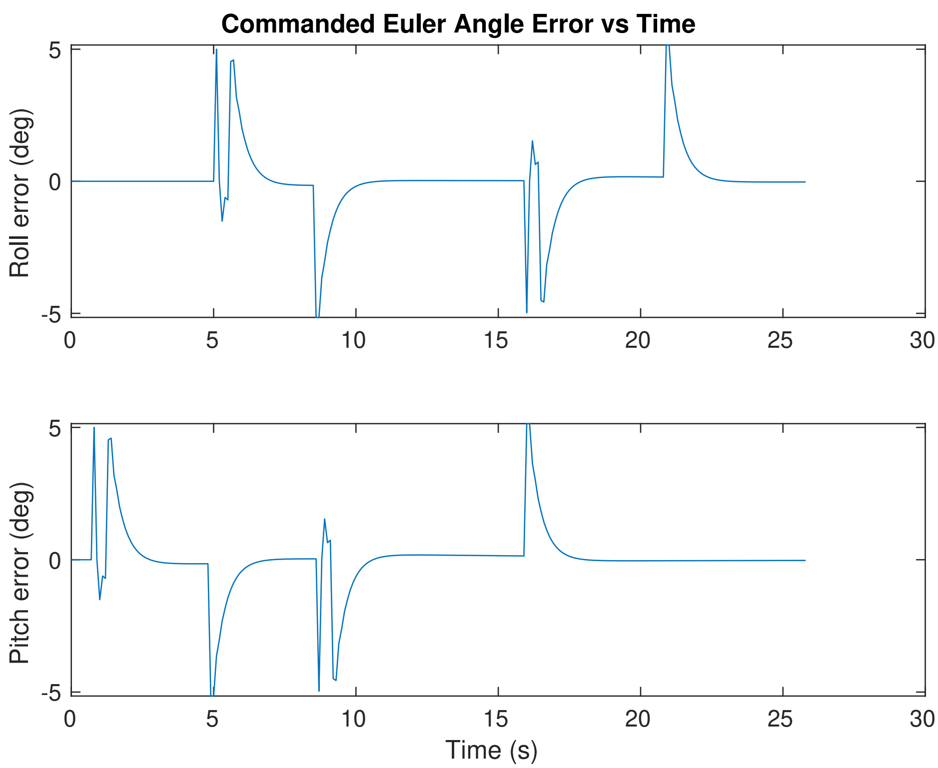


Figure 3.14. Manual flight - commanded euler angle error vs time

3.2 Kalman Filter

3.2.1 Waypoint Flight

Plots from the Kalman filter output are presented in Figures 3.15 - 3.19.

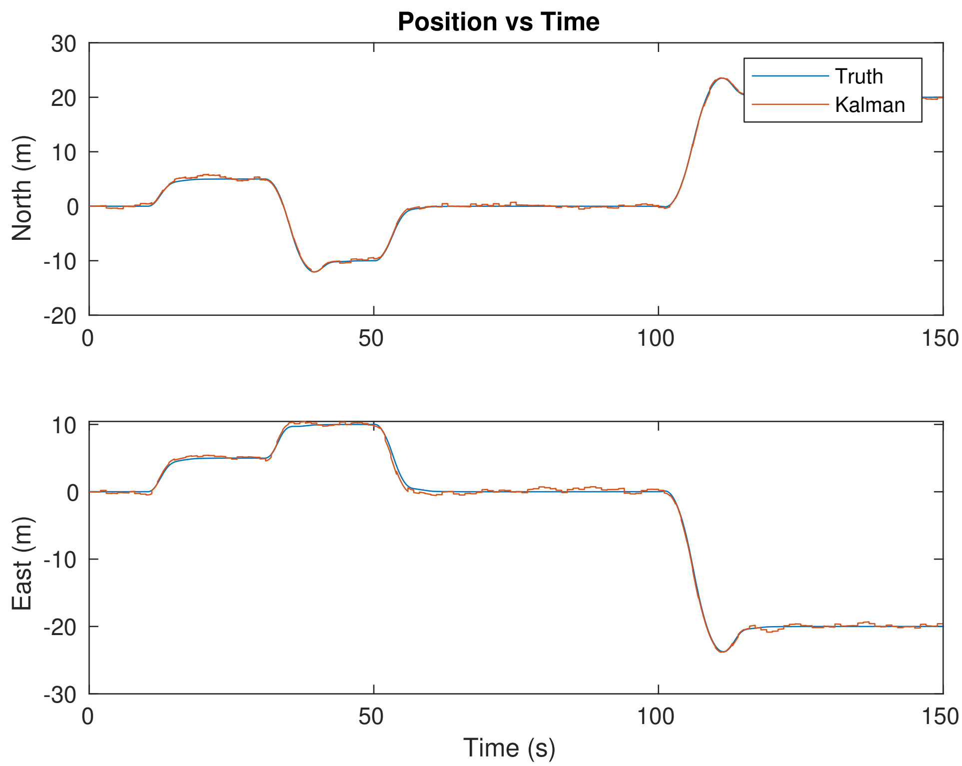


Figure 3.15. Kalman and truth - position vs time

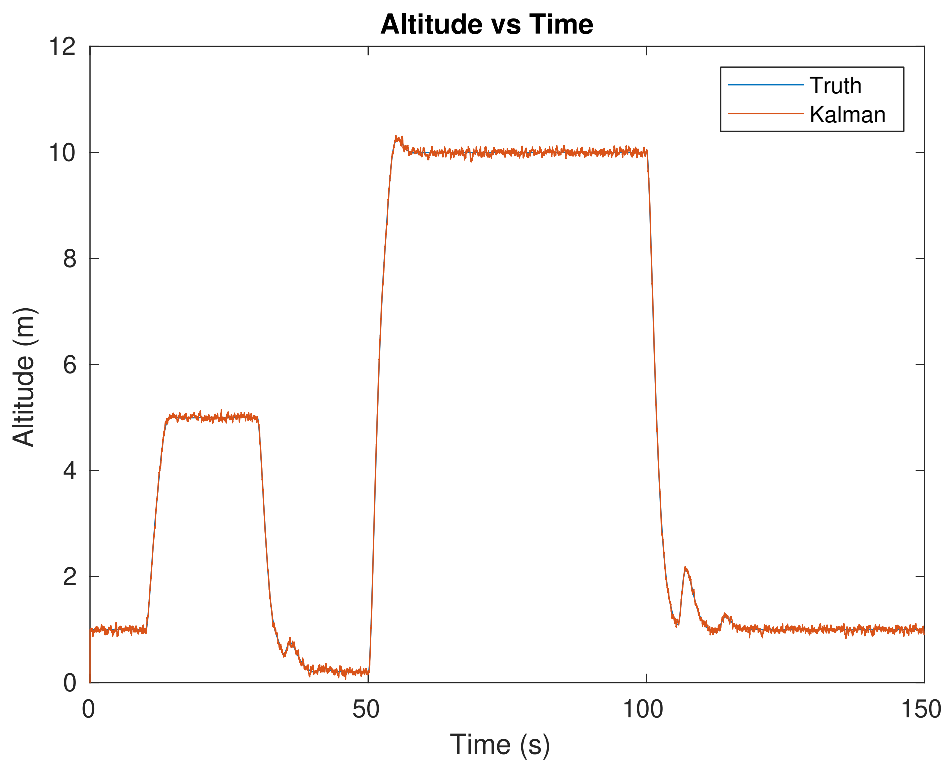


Figure 3.16. Kalman and truth - altitude vs time

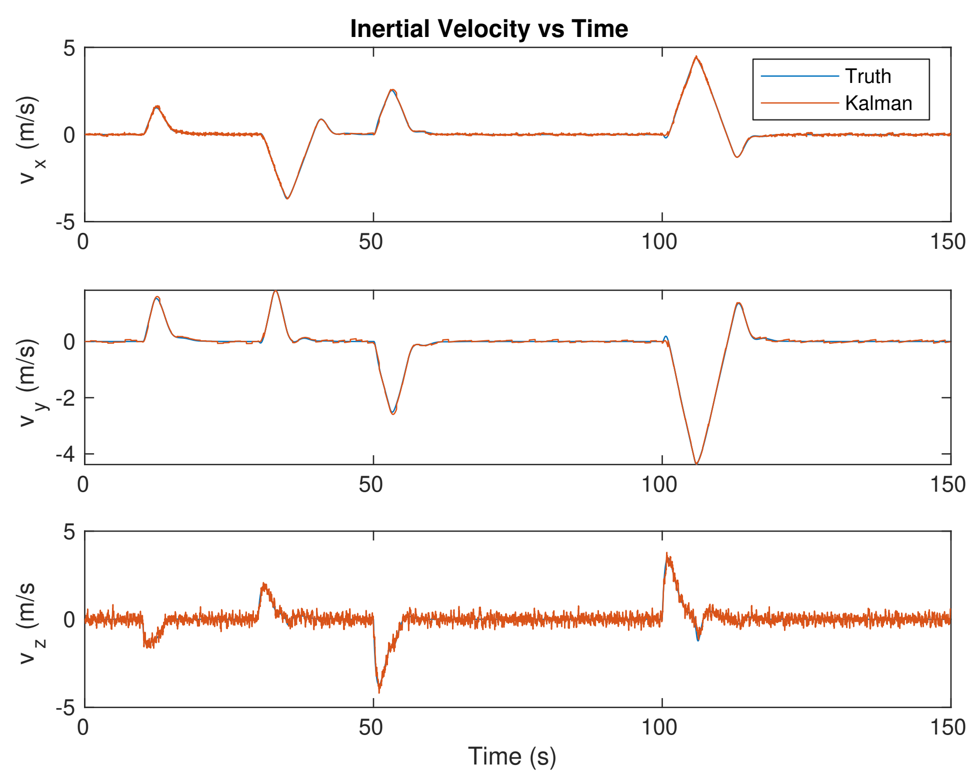


Figure 3.17. Kalman and truth - velocity vs time

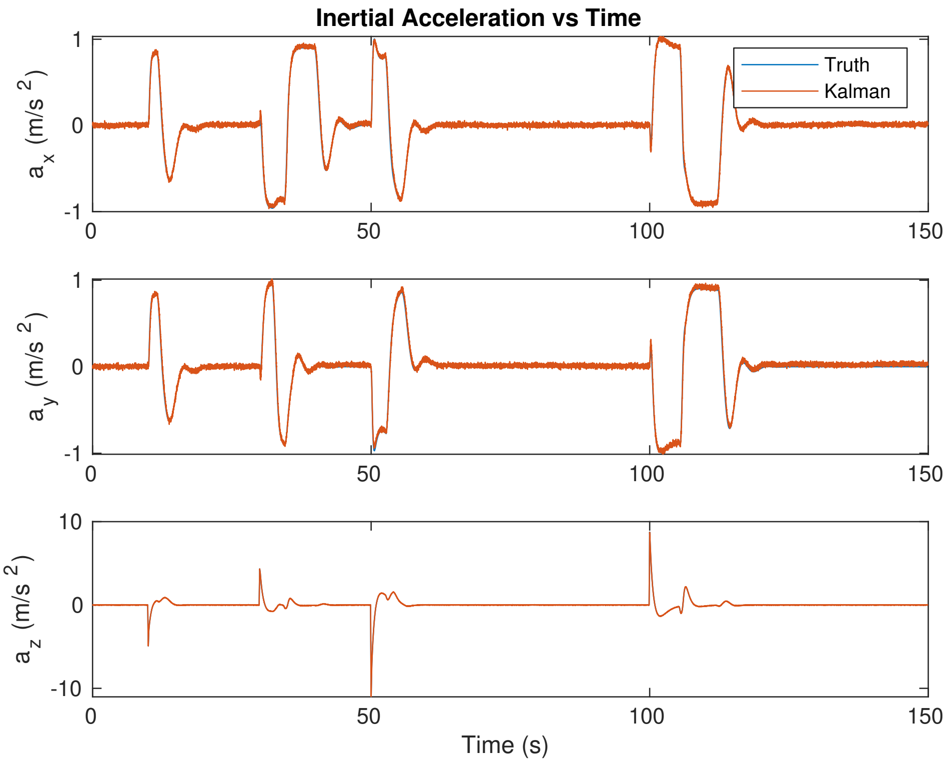


Figure 3.18. Kalman and truth - acceleration vs time

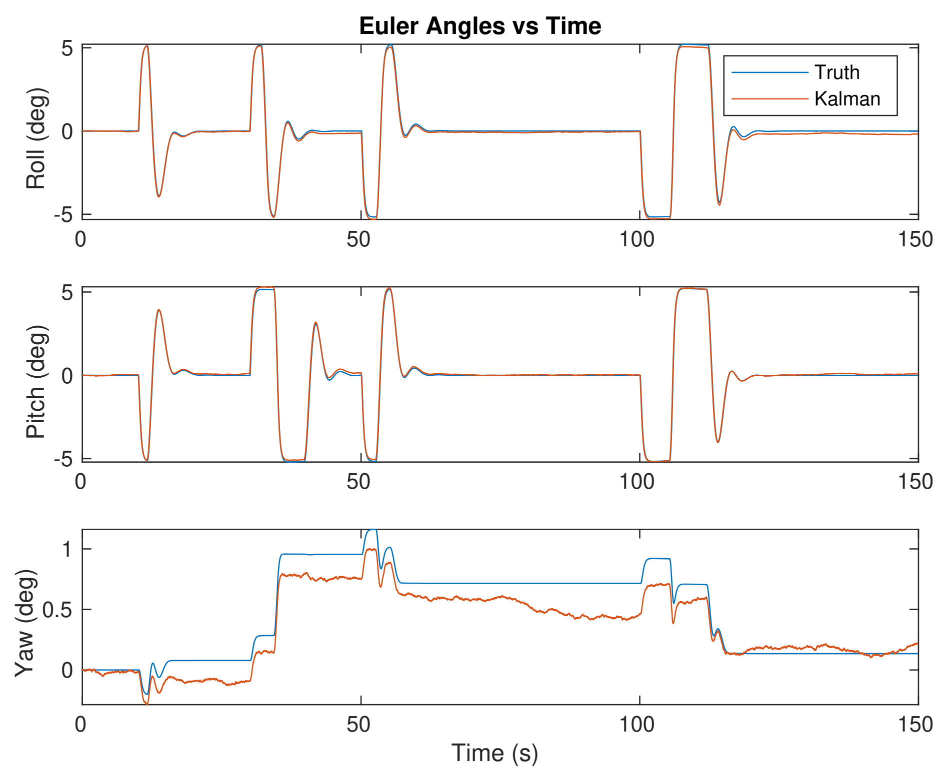


Figure 3.19. Kalman and truth - Euler angles vs time

3.3 Radioactive Source Localization

The final set of results enables radioactive source detection. Two cases are presented below. The first case assumes a 1mCi source is located at a distance ranging from 1m to 25m. 100 runs were performed holding the initial distance from the source constant while generating a random starting location for the source. These runs simulated 1800 seconds of flight time, assumed an average background activity of 1 count/sec, and used an activity threshold of 5 counts/sec.

Case 2 uses a source ten times weaker (0.1mCi) representing capability for the algorithm to localize a weak source. All other conditions were unchanged from the first case.

Figure 3.20 reveals the number of runs which converge withim 1m. Figure 3.21 indicates the average number of iterations required for a run to converge within 1m. The upper plot uses error bars to show minimum and maximum number of iterations while the lower plot shows standard deviations. Only runs which were able to localize the source within 1m are shown. Figure 3.22 shows the average duration of time spent idle before the localization algorithm is triggered.

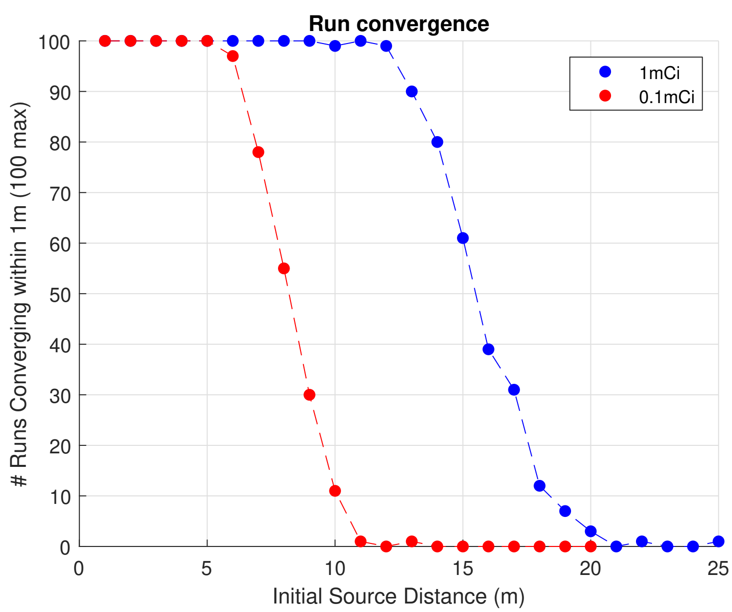


Figure 3.20. Run convergence for the 1mCi and 0.1mCi cases.

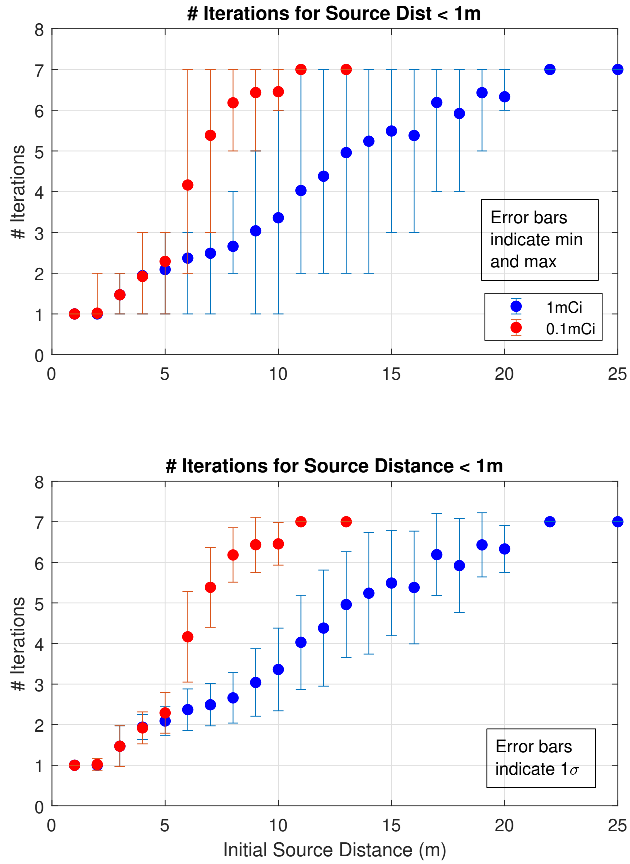


Figure 3.21. Number of iterations required for the UAV to localize the source within 1m. Only runs that were able to localize the source are plotted. Seven iterations is the maximum number of iterations possible within the constrained simulation time.

3.3.1 Case 1: 1mCi Source

For the 1mCi source, 90% of the runs converged within 1m of the radioactive source when initialized within 13m. The 1mCi source was assumed to record 1000 counts/sec at a distance of 1m.

Simulation and Kalman plots for a single run where the source was localized 10m away are presented in Figures 3.23 - 3.28.

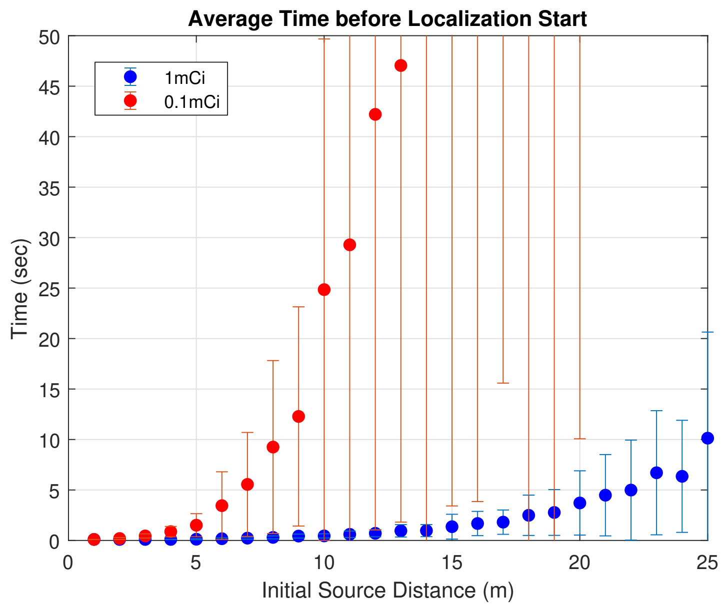


Figure 3.22. Average length of time before the UAV will start it's first localization run. The y-axis is cut off at 50 seconds.

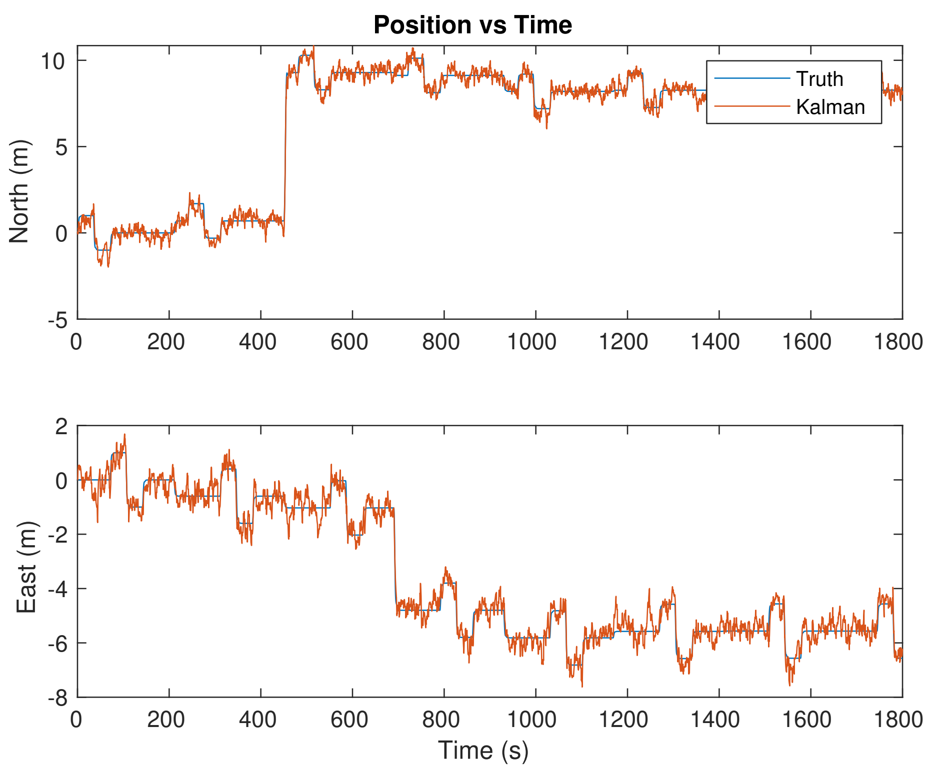


Figure 3.23. 1mCi Source initialized 10m away - Kalman UAV position vs time

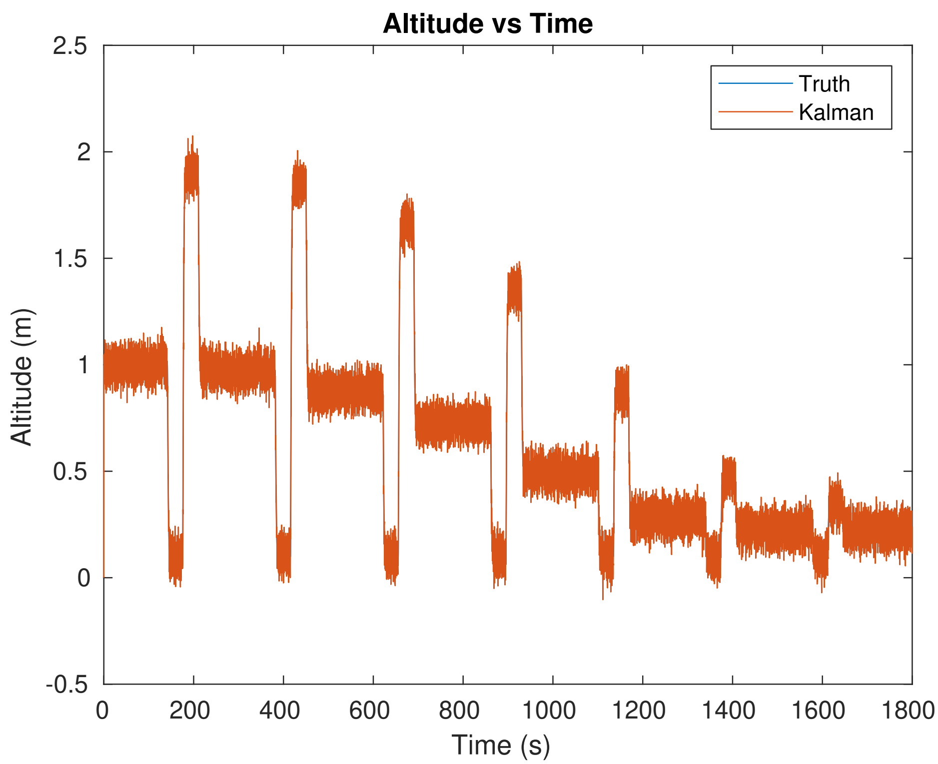


Figure 3.24. 1mCi Source initialized 10m away - Kalman UAV altitude vs time

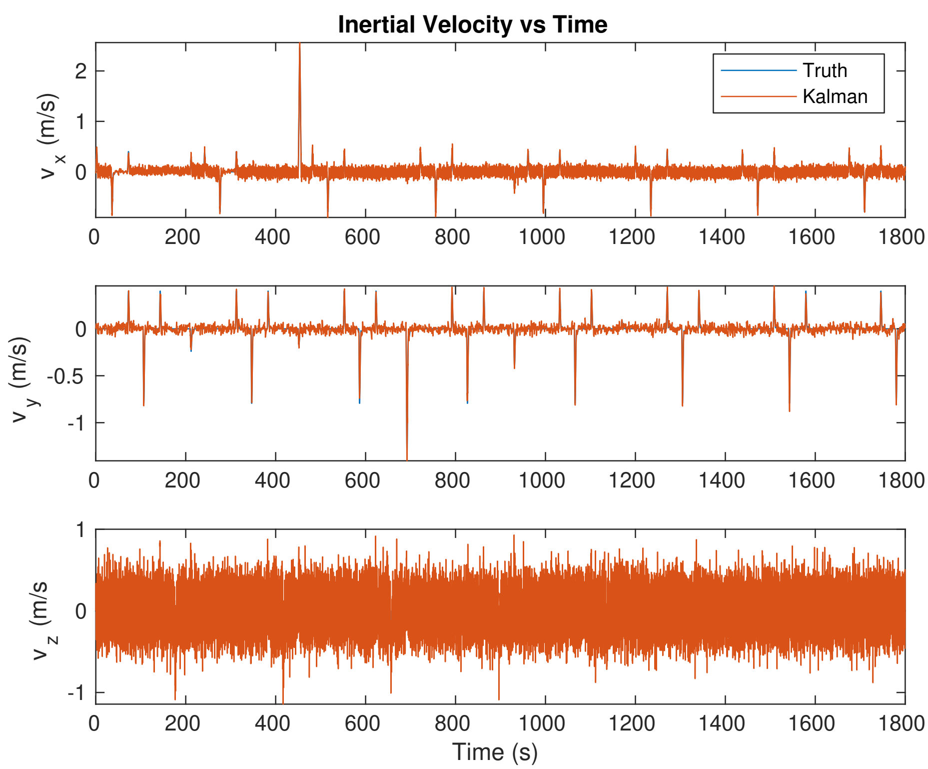


Figure 3.25. 1mCi Source initialized 10m away - Kalman velocity vs time

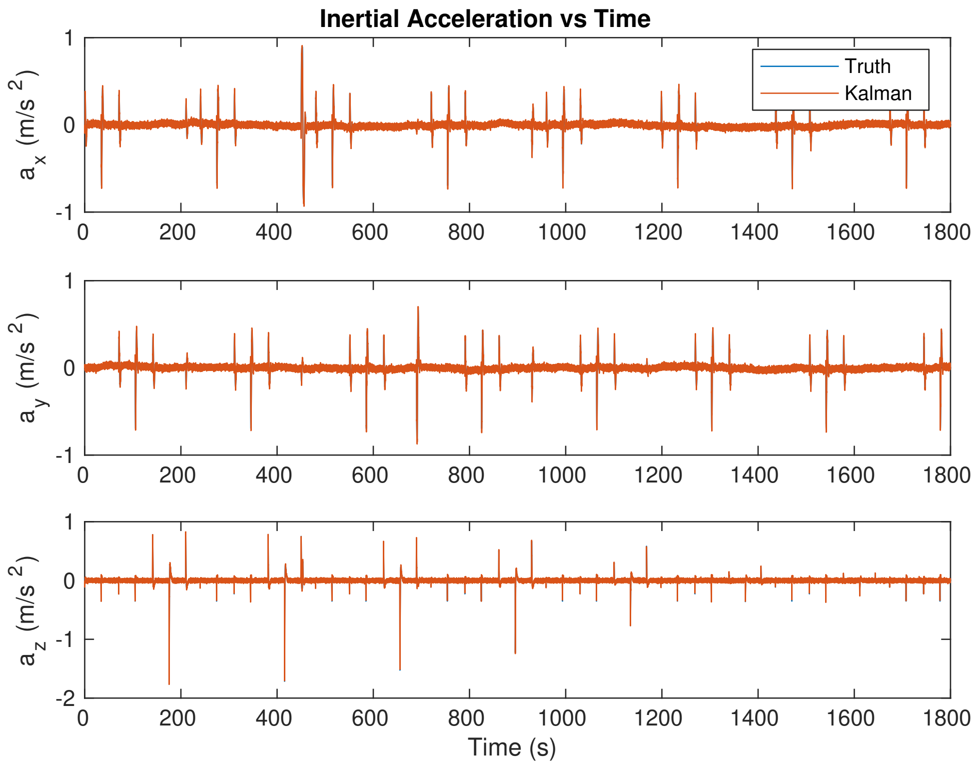


Figure 3.26. 1mCi Source initialized 10m away - Kalman acceleration vs time

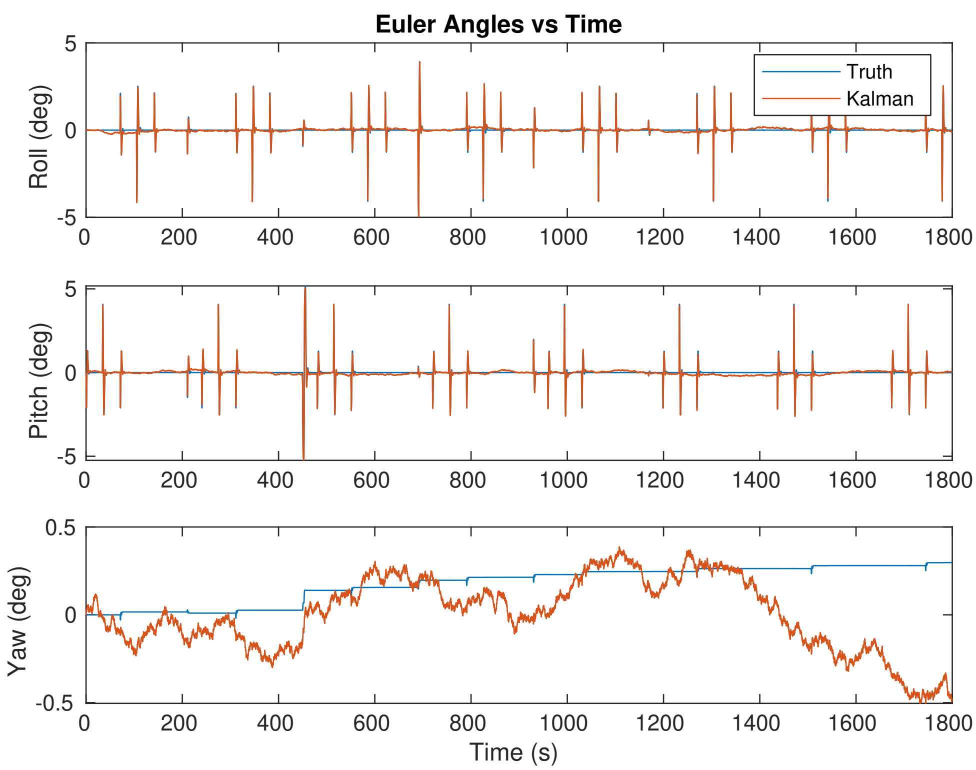


Figure 3.27. 1mCi Source initialized 10m away - Kalman Euler angles vs time

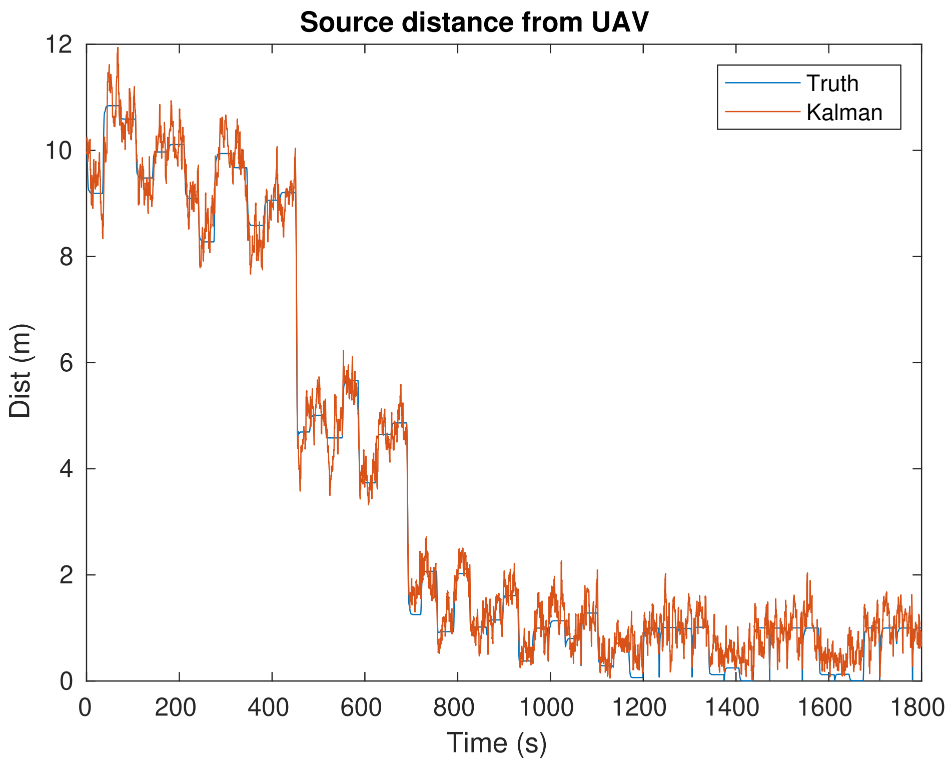


Figure 3.28. 1mCi Source initialized 10m away - Kalman source distance from UAV vs time

3.3.2 Case 2: 0.1mCi Source

Figure 3.20 shows that greater than 90% of runs will converge if the source is initialized within 6m. The 0.1mCi source was assumed to record 100 counts/sec at a distance of 1m.

CHAPTER 4

DISCUSSION

4.1 Simulation

The figures presented in Chapter 3.1 suggest the 6DOF UAV simulation is stable and provide a reasonable model of UAV motion in the absence of wind. The position vs time plots for waypoint commands in Figures 3.1 and 3.2 show that the UAV reaches each location without significant overshoot. Table 4.1 shows the UAV time of flight until it is within 10cm of its destination moving less than 5cm/sec.

Table 4.1. Time until UAV reaches target waypoint. This is defined when the UAV is within 10cm of the location and has a velocity magnitude of less than 5cm/sec.

Waypoint	Time of flight (sec)
1	9.4
2	15.2
3	9.9
4	19.5

For all simulations, the maximum allowable roll or pitch angle was set to five degrees. The PID controller clearly follows this as shown in Figures 3.7 and 3.13. In Figure 3.13 a MATLAB limitation results in the short spikes preceding the extended command lines. When a key is held, inputs are not fired repeatedly until after a short duration. At t = 5sec, the roll key was pressed and held, but did not begin repeatedly firing until shortly after as is clearly shown in Figure 3.13.

A simulation limitation can also be seen in Figures 3.7 and 3.13. In Chapter 2.2, there is no control for yaw motion. Why are yaw changes present? Referring back to (2.20), a nonzero roll, pitch, and ω_y will result in an nonzero yaw rate of change ($\dot{\psi}$).

4.2 Kalman Filter

As seen in the figures presented in Chapter 3.2, the Kalman Filter adequately estimates all state vectors with the most noticeable error occuring when estimating yaw. Improving the yaw estimate would require the addition of an additional sensor, such as a magnetometer.

Estimating roll and pitch angles from accelerometer measurements effectively mitigates the integration error accumulated from the gyroscope measurements. Figures 3.19 and 3.27 provide no indication error accumulation within 180 or 1800 seconds respectively. This method for mitigating gyroscope error should work well for a wide variety of quadcopter applications. The condition required for a valid accelerometer angle measurement is often satisfied throughout quadcopter flight.

4.3 Radioactive Source Localization

4.3.1 1mCi Case

For a source with radioactivity equal to 1mCi, Figure 3.20 shows that 90% of runs converge when the initial source distance ranges from 1m to 13m (inclusive). Between 13m and 20m, the number of runs that converge within the simulation time rapidly diminishes. Beyond 20m, one run was able to converge at 21m and 25m. This should be regarded as two fortunate runs.

The number of iterations until convergence within 1m (shown in Figure 3.21) gradually increases from 1 iteration to 7 iterations beyond 20m. From Figure 4.1, the decline of radioactivity with distance is much more gradual compared to the 0.1mCi case.

The gradual climb results from increasing initial distance which in turn decreases the accuracy of measurements as shown in Figure 4.1 Additionally, only runs that converge within 1m of the source are shown in Figure 3.21. It should be expected that if a run will converge, it will require more iterations to converge as distance increases.

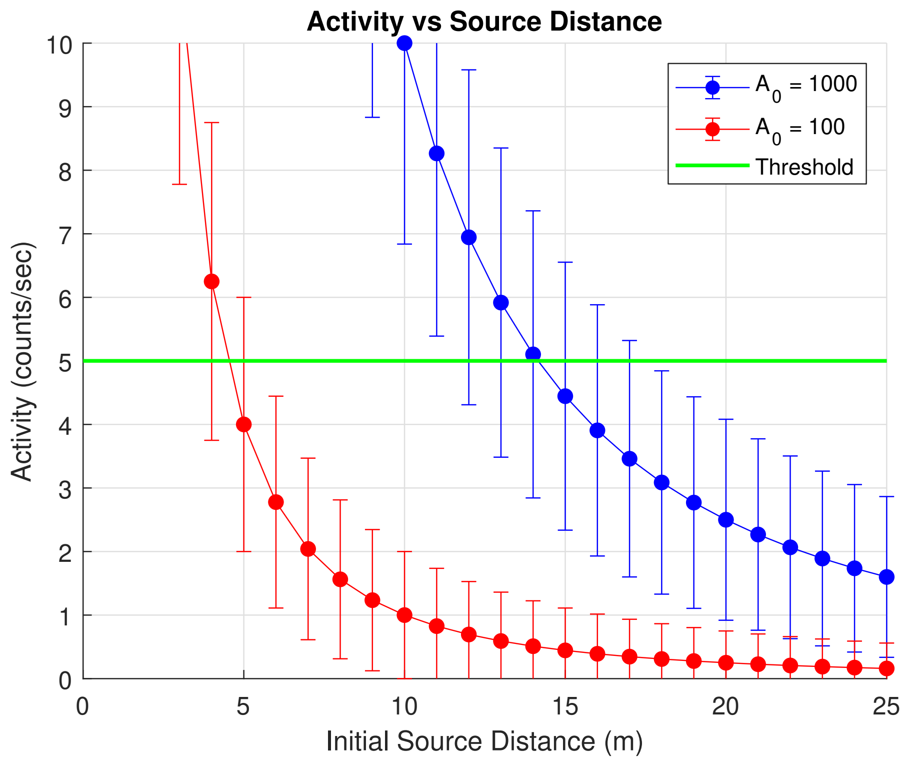


Figure 4.1. Expected activity measurements as distance increases. An activity threshold of five counts per second was used in both cases. The activity 1m from the source is assumed to be 1000 counts/sec and 100 counts/sec for the 1mCi and 0.1mCi source respectively.

Comparing Figures 3.20 and 4.1 reveals the cause of the rapid decline in run convergence begining beyond 12m (13m had 90% convergence, but prior to 12m, all runs had 99%–100% convergence). Assume the UAV is located 12m from the source and that the source is located along a measurement axis (which gives the best algorithm performance). Two measurements will be taken along this axis, one at 11m, and one at 13m. From Figure 4.1, the activity measured (within one standard deviation) for the 11m and 13m measurement is $A_{11m} = 8.26 \pm 2.87$counts/sec and $A_{13m} = 5.91 \pm 2.43$counts/sec respectively. The upper end of A_{13m} is greater than the mean of A_{11m}. As the source distance continues to increase, the difference in average activity continues to decrease. Recall that for a set of measurements to be used, the difference between activities at two measurement points must be greater than the standard deviation of the lower activity measurement. Beyond this 12m mark, the probability that a set of measurements will produce any useful localization result rapidly diminishes.

As the probability of obtaining a usable result diminishes, the probability that a detrimental set of measurements is obtained will increase. Figure 4.2 shows the probability that the farther measurement will record a **greater** activity than the average activity of the closer measurement if the source lies along a measurement axis. As such, this plot represents the most favorable scenario for the algorithm.

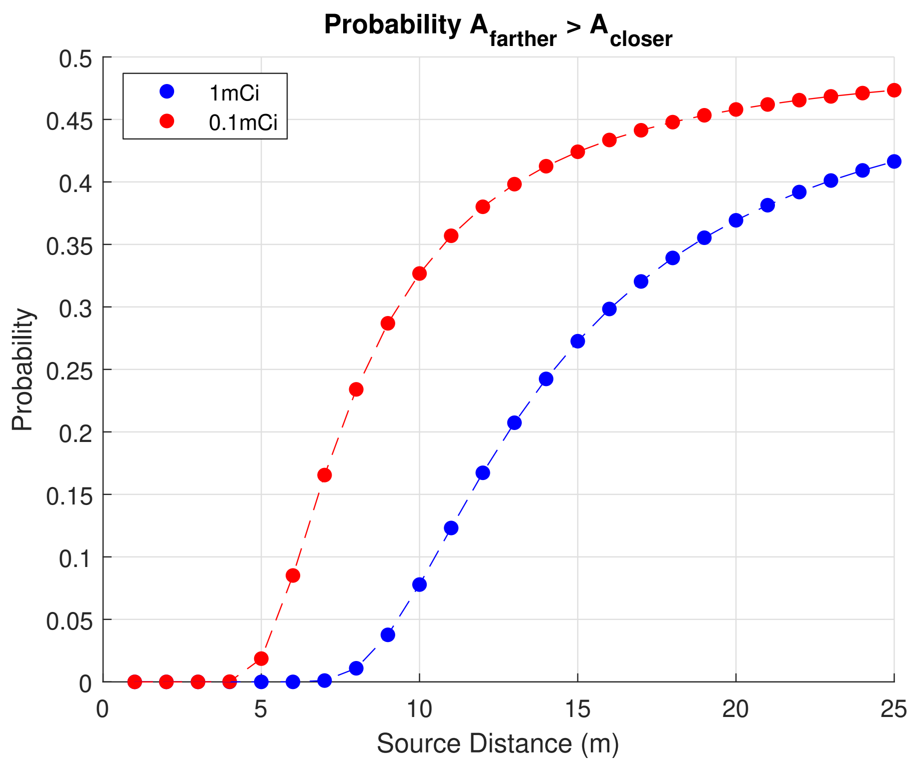


Figure 4.2. Probability that the farther activity measurement will be greater than the closer activity measurement's mean. For example: For the 1mCi source, when the localization algorithm is run at a distance of 15m (and lies along a measurement axis), the probability that the 16m measurement produces a greater result than the average activity at 14m is 27%.

A UAV searching for radioactive sources would need to have the activity threshold met while in flight. Figure 3.22 shows the average time until activity threshold is exceeded. On average, the activity threshold is exceeded within one second when initialized within 14m and 4m for the 1mCi and 0.1mCi case respectively. For a particular mission, this can be used to determine a maximum UAV travel speed.

4.3.2 0.1mCi Case

The 0.1mCi case was used to illustrate algorithm effectiveness for measuring algorithm effectiveness for a weak radioactive source. Figure 3.20 shows that 90% of runs converge when the source is initialized within 6m. Beyond 6m, run convergence decreases rapidly until reaching a distance of 10m. Beyond 10m, one run converged at 11m and 13m, while all other runs were unable to converge within 1m.

In Figure 3.21, there is a noticeable jump between 5m and 6m in the number of iterations required for a run to converge. Again, consider the case where the radioactive source lies along a measurement axis. If the source is 5m away, we will record measurements at distances of 4m and 6m. From Figure 4.1, the error bars between 4m and 6m have minimal overlap and the average activity to be measured is expected to be above the activity threshold. When then source is 6m away, the 5m and 7m activity measurements reveal that the 7m measurement is within one standard deviation of the 5m measurement (the opposite is not true). Additionally, the mean of both measurements falls below the activity threshold. The combination of these two factors leads to iterations with unusable measurements, which results in the significant jump from in iterations required between 5m and 6m.

Beyond 6m, the activity threshold is outside one standard deviation of any possible measurement. Additionally, the standard deviation at each measurement location will enclose the other measurement's mean. For example, if the source is initialized at 7m, the standard deviation of activity at 8m encloses the average activity at 6m and vice-versa. As the source distance increases, the difference between average activities will continue to decrease.

CHAPTER 5
CONCLUSION

UAVs are capable of expanding the available mission-space to environments unsuitable for humans. Regions with high concentrations of radioactive materials may produce lethal doses rendering them inaccessible. An algorithm has been developed to localize radioactive sources in an effort to improve the ability to safely perform experiments and missions in these areas.

Testing the algorithm required a model capable of simulating UAV flight characteristics. The simulated UAV can be controlled via keyboard input or by predetermined waypoints. Simulation results are presented in Chapter 3.1.

In addition to the simulation, an Extended Kalman Filter (EKF) was developed for use in estimating UAV states. The EKF has been shown to adequately estimate all states presented in Table 2.4 with the exception of the yaw Euler angle. This particular state is limited by the instruments assumed to be onboard a typical UAV.

Finally, localizing a radioactive source has been shown for a 1mCi source and a 0.1mCi source. Both sources were simulated for 1800 seconds. For the 1mCi source, the UAV was shown to converge within 1m of the radioactive source in $\geq 90\%$ until initialized 13m away. For the 0.1mCi source, the UAV converged within 1m in $\geq 90\%$ of runs until initialized 6m away from the source.

5.1 Future Work

This project leaves numerous avenues available for improvement. Listed below are the possible improvements which could be made to the project.

5.1.1 Simulation Verification

The simulation was developed and tested without true flight data for verification. Using the physical properties of a UAV, the simulation could be verified or improved via flight test.

5.1.2 Simulation Improvements

The fidelity of the simulation could be improved by modeling additional factors in flight. These factors include wind, electrical / mechanical interations, and improvements to the PID controller. A model for the effects of an electric motor on propeller dynamics, along with an alternative dynamics formulation, can be found in [11].

5.1.3 Extended Kalman Filter

As previously discussed, an additional sensor added to the EKF which provides yaw information would vastly reduce the total state estimation error. A magnetometer is such a sensor which could be used to determine the angle between magnetic north and the body-x axis.

An additional EKF improvement would be to expand the state vector to estimate any gyroscope or accelerometer bias. Such an improvement would lead to a more robust filter with improved hardware compatibility. In addition, this would help mitigate the yaw drift visible in Figure 3.27.

5.1.4 Source Localization

Additional sensors capable of measuring counts from radioactive sources would improve the probability of measuring activity near the true mean of the source. As a result, this would improve the speed at which the UAV can converge on a source and improve the maximum distance that the UAV can detect a source.

A method for detecting radioactivity as outlined in [4] would provide a light, cost-effective, option as any consumer-grade CCD (such as a cheap web camera) would produce measurements of radioactivity. Using a CCD for the source localization algorithm would require a wrapper to simulate counts from CCD outputs.

5.1.5 Source Localization In the EKF

Moving the source localization algorithm into the EKF may provide numerous benefits to the algorithm. If the localization algorithm were part of the EKF, the inclusion of additional sensors would improve measurement reliability and thus improve algorithm effectiveness.

The initial state estimate would prove useful if any prior information about the source is known. For example, if the source is known to be located north of the UAV initial position, this information can be used to produce a better estimate of the source position.

Finally, radioactivity is known to have detrimental effects on electronics. Improving radiation hardening technology is a current area of research [18, 12, 15]. It may be possible to use these detrimental effects to improve localization accuracy in the EKF.

BIBLIOGRAPHY

[1] Henry Ernest Baidoo-Williams. “Novel techniques for estimation and tracking of radioactive sources”. PhD thesis. Iowa City, Iowa, USA: University of Iowa, Jan. 2014. DOI: 10.17077/etd.comm5gze. URL: https://ir.uiowa.edu/etd/1539.

[2] Shawn S Brackett. “Integrated Environment and Proximity Sensing for UAV Applications”. PhD thesis. University of Maine, 2017. 263 pp. URL: https://digitalcommons.library.umaine.edu/etd/2796/.

[3] Jren-Chit Chin et al. “Accurate localization of low-level radioactive source under noise and measurement errors”. In: *Proceedings of the 6th ACM conference on Embedded network sensor systems - SenSys ’08*. the 6th ACM conference. Raleigh, NC, USA: ACM Press, 2008, p. 183. ISBN: 978-1-59593-990-6. DOI: 10.1145/1460412.1460431. URL: http://portal.acm.org/citation.cfm?doid=1460412.1460431.

[4] John A. Cummings et al. “Detection and Analysis of Uncharged Particles Using Consumer-grade CCDs:” in: *Health Physics* 118.6 (June 2020), pp. 583–592. ISSN: 0017-9078. DOI: 10.1097/HP.0000000000001211. URL: http://journals.lww.com/10.1097/HP.0000000000001211 (visited on 05/05/2020).

[5] Bernard Etkin and Lloyd D. Reid. *Dynamics of flight: stability and control*. 3rd ed. New York: Wiley, 1996. 382 pp. ISBN: 978-0-471-03418-6.

[6] Chirstopher J. Fisher. *Using An Accelerometer for Inclination Sensing | DigiKey - www.digikey.com/*. Digi-Key. Contributed By Convergence Promotions LLC. May 6, 2011. URL: https://www.digikey.com/en/articles/techzone/2011/may/using-an-accelerometer-for-inclination-sensing.

[7] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control: Fossen/Handbook of Marine Craft Hydrodynamics and Motion Control*. Chichester, UK: John Wiley & Sons, Ltd, Apr. 8, 2011. ISBN: 978-1-119-99413-8 978-1-119-99149-6. DOI: 10.1002/9781119994138. URL: http://doi.wiley.com/10.1002/9781119994138 (visited on 03/17/2018).

[8] Kristofer Henderson et al. “Tracking Radioactive Sources through Sensor Fusion of Omnidirectional LIDAR and Isotropic Rad-Detectors”. In: *2017 International Conference on 3D Vision (3DV)*. 2017 International Conference on 3D Vision (3DV). Qingdao: IEEE, Oct. 2017, pp. 97–106. ISBN: 978-1-5386-2610-8. DOI: 10.1109/3DV.2017.00021. URL: https://ieeexplore.ieee.org/document/8374562/.

[9] Hui Liu and Yingwei Yao. “A moving radioactive source tracking and detection system”. In: *2009 43rd Annual Conference on Information Sciences and Systems*. 2009 43rd Annual Conference on Information Sciences and Systems (CISS). Baltimore, MD, USA: IEEE, Mar. 2009, pp. 50–54. ISBN: 978-1-4244-2733-8. DOI: 10.1109/CISS.2009.5054689. URL: http://ieeexplore.ieee.org/document/5054689/.

[10] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82 (Series D 1960), pp. 35–45.

[11] Bouzgou Kamel et al. “Dynamic modeling, simulation and PID controller of unmanned aerial vehicle UAV”. In: *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*. 2017 Seventh International Conference on Innovative Computing Technology (INTECH). Luton: IEEE, Aug. 2017, pp. 64–69. ISBN: 978-1-5090-3989-0. DOI: 10.1109/INTECH.2017.8102445. URL: http://ieeexplore.ieee.org/document/8102445/.

[12] Andrew S. Keys and Michael D. Watson. “Radiation Hardened Electronics for Extreme Environments”. In: (Jan. 1, 2007), p. 4.

[13] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. “Using Inertial Sensors for Position and Orientation Estimation”. In: *Foundations and Trendső in Signal Processing* 11.1 (2017), pp. 1–153. ISSN: 1932-8346, 1932-8354. DOI: 10.1561/2000000094. arXiv: 1704.06053. URL: http://arxiv.org/abs/1704.06053.

[14] Mathworks. *MATLAB*. 2019.

[15] George C. Messenger. *Radiation hardening*. type: dataset. McGraw-Hill Education, 2014. DOI: 10.1036/1097-8542.566850. URL: http://accessscience.com/content/566850.

[16] Mark Pedley. *Tilt Sensing Using a Three-Axis Accelerometer*. 2014. URL: https://cache.freescale.com/files/sensors/doc/app_note/AN3461.pdf.

[17] Dan Simon. *Optimal state estimation: Kalman, H [infinity] and nonlinear approaches*. OCLC: ocm64084871. Hoboken, N.J: Wiley-Interscience, 2006. 526 pp. ISBN: 978-0-471-70858-2.

[18] R. Sorge et al. “JICG MOS transistors for reduction of radiation effects in CMOS electronics”. In: *2018 IEEE Topical Workshop on Internet of Space (TWIOS)*. 2018 IEEE Topical Workshop on Internet of Space (TWIOS). Anaheim, CA: IEEE, Jan. 2018, pp. 17–19. ISBN: 978-1-5386-1294-1. DOI: 10.1109/TWIOS.2018.8311401. URL: http://ieeexplore.ieee.org/document/8311401/.

[19] Chih-Hung Wu, Wei-Zhou Hong, and Shing-Tai Pan. “Performance Evaluation of Extended Kalman Filtering for Obstacle Avoidance of Mobile Robots”. In: *Hong Kong* (2015), p. 5.

APPENDIX A

UAV SIMULATION

```
function A = aerodynamic_forces_moments(x, windVel, u, UAV)
% aerodynamic_forces_moments - Computs aerodynamics forces and
    moments
% acting on the UAV
%
% USAGE:
%   aerodynamic_forces_moments(x, windVel, u, UAV)
%
% INPUTS
%   x:          state vector
%   windVel: velocity of the wind in the wind frame (m/s)
%   UAV:        structure containing UAV constants
% OUTPUTS
%   A:          [6 x 1] vector where the first three elements contain
%               aero forces (N) and the last three contain aero
    moments (N*m^2)
%   aeroForce:         Aero forces in body frame (N)
%   aeroMom:           Aero Moments (N*m)

[~, ~, ~, density] = atmosisa(-x(3)); % density (kg/m^3)

% unpack state
uavVelInBody = [x(4), x(5), x(6)]';      % NED velocity

```

```
% initialize variables
propLength = norm(UAV.propVects(:, 1)); % propeller length (m)

propVelInBodySquared = u*(propLength/2)^2; % body frame propeller
    speed

% wind frame conversions
[aoa, ssa] = get_aoa_ssa(uavVelInBody);
uavVelInWind = DCM_body2wind(uavVelInBody, aoa, ssa);
velWithWind = uavVelInWind + windVel; % total relative air
    velocity (wind frame)

% Drag force and moment (wind frame)
dragInWind = 1/4 * [-1/2 * density * velWithWind(1)^2 * UAV.
    propArea * UAV.CD; 0; 0]; % drag (N)
dragInBody = DCM_wind2body(dragInWind, aoa, ssa);
% Thrust force and moment (body frame)
thrustInBody = zeros(3, 4);
thrustMomInBody = zeros(3, 4);
dragMomInBody = zeros(3, 4);
for i = 1:4 % four props
    velSquared = propVelInBodySquared(i);
    thrustInBody(:, i) = [0; 0; -1/2 * density * velSquared * UAV.
        propArea * UAV.CT]; % Thrust (N)
    thrustMomInBody(:, i) = cross(UAV.propVects(:, i)/2,
        thrustInBody(:, i)); % Thrust moment (N*m)
```

```
    dragMomInBody(:, i) = cross(UAV.propVects(:, i)/2, dragInBody)
        ;
end
dragMomInBody = zeros(3, 4); % drag moment (N*m)
% Aero force and moment in body frame
aeroForce = sum(dragInBody, 2) + sum(thrustInBody, 2);
aeroMom = sum(dragMomInBody, 2) + sum(thrustMomInBody, 2);
A = [aeroForce;aeroMom];
end
```

```
function [uDelta, angleTarget, angleErrorSum, rollCommand,
    pitchCommand] = compute_uDelta(controllerInput, x, Kp, Td,Ti,
    angleDeadband, UAV, dt, angleErrorSum, target_pos_ned)
% compute_uDelta - Computes change in u to achieve desired input
%
% INPUTS
%   controllerInput:  [2 x 1] +/- 1 from keyboard or joystick
    input
%         [+/- bx, +/- by]. Note bx stands for body-x direction.
%         Example: [1, -1] would mean roll for +bx and -by
%   x: [18 x 1] state vector
%   angleGain: controller gain
%   angleDeadband: minimum angle error to ignore commands
%   UAV: UAV parameters
%   dt: time step
%
% OUTPUTS
```

```
%   uDelta: [4 x 1] control input delta to be added to uHover to
    achieve
%   commanded input.
%
% NOTE
%   propeller arrangement is as follows [1  2]  with [1  2] in +bx
%                                       [4  3]  and  2, 3 in +by
%

%% unpack input
bxInput = controllerInput(1);
byInput = controllerInput(2);

% pos
r_uav_ned = [x(1), x(2), x(3)]';

% vel
v_uav_body = [x(4), x(5), x(6)]';

% body angular velocity
rollRate = x(7);
pitchRate = x(8);
yawRate = x(9);
omega = [rollRate, pitchRate, yawRate]';
% euler angles (rad)zVelInertial
roll = x(10);
pitch = x(11);
```

```
yaw = x(12);

% angle error sums
rollErrorSum = angleErrorSum(1);
pitchErrorSum = angleErrorSum(2);

%%
[~, ~, ~, density] = atmosisa(-x(3)); % density (kg/m^3)

propLength = norm(UAV.propVects(1:3, 1)); % meters

gain_pos2angle = 0.05;
gain_vel2angle = 0.13;
% if target_pos_ned is set, override controller commands
if ~isempty(target_pos_ned)
    [target_pos_body] = DCM_ned2body(target_pos_ned, [roll; pitch;
        yaw]);
    r_uav_body = DCM_ned2body(r_uav_ned, [roll; pitch; yaw]);
    bx_error = target_pos_body(1) - r_uav_body(1);
    by_error = target_pos_body(2) - r_uav_body(2);
    bz_error = target_pos_body(3) - r_uav_body(3);
    vel_error = 0 - v_uav_body;

    % roll and pitch targets are set by position errors
    pitchTarget = -(gain_pos2angle * bx_error + gain_vel2angle *
        vel_error(1));
```

```
    rollTarget = gain_pos2angle * by_error + gain_vel2angle *
        vel_error(2);
    if abs(pitchTarget) > abs(UAV.maxRotation)
        pitchTarget = sign(pitchTarget) * UAV.maxRotation;
    end
    if abs(rollTarget) > abs(UAV.maxRotation)
        rollTarget = sign(rollTarget) * UAV.maxRotation;
    end
else
    % input gives all or nothing +/- input (rad)
    pitchTarget = -bxInput * UAV.maxRotation; % negative pitch
        angle results in +bx motion
    rollTarget = byInput * UAV.maxRotation; % positive roll angle
        results in +by motion
end
% angle errors (radians)
rollError = rollTarget - roll;
rollErrorSum = rollErrorSum + rollError;
pitchError = pitchTarget - pitch;
pitchErrorSum = pitchErrorSum + pitchError;

% angle rate errors (rad/s)
rollRateError = 0 - rollRate;
pitchRateError = 0 - pitchRate;

% commanded pitch and roll based on gain K.
```

```
pitchCommand = Kp * ((pitchError) + Td * (pitchRateError) + 1/Ti *
    (pitchErrorSum * dt));
rollCommand = Kp * ((rollError) + Td * (rollRateError) + 1/Ti * (
    rollErrorSum * dt));

% uDeltaFun terms
inertialTerm = cross(omega, UAV.inertiaMatrix*omega);
thrustTerm = 8/(propLength^3 * cosd(45) * density * UAV.propArea *
    UAV.CT);
uDeltaFun = @(angleError, angleRateCurrent, I) ...
    2*I * (angleError - angleRateCurrent * dt)/(dt^2);

% If difference between target angle and current angle is less
    than the set
% deadband, do not issue a command.
if abs(pitchTarget - pitch) < angleDeadband
    bxDeltaU = 0;
else
    % rotation about y-axis (pitch) results in bx motion
    bxDeltaU = thrustTerm * ...
        (uDeltaFun(pitchCommand, pitchRate, UAV.inertiaMatrix(2,
            2)) - inertialTerm(2));
end
if abs(rollTarget - roll) < angleDeadband
    byDeltaU = 0;
else
    % rotation about x-axis (roll) results in by motion
```

```
    byDeltaU = thrustTerm * ...
        (uDeltaFun(rollCommand, rollRate, UAV.inertiaMatrix(1, 1))
            - inertialTerm(1));
end

% decode uDelta to correct propellers.
% bxDeltaU = u12 - u34
u1x = 1/4 * bxDeltaU;
u2x = u1x;
u3x = -u1x;
u4x = u3x;

% byDeltaY = u14 - u23
u1y = 1/4 * byDeltaU;
u2y = -u1y;
u3y = u2y;
u4y = u1y;

% sum each u value
u1 = u1x + u1y;
u2 = u2x + u2y;
u3 = u3x + u3y;
u4 = u4x + u4y;

% construct output
uDelta = [u1, u2, u3, u4]';
angleTarget = [rollTarget, pitchTarget]';
```

```
angleErrorSum = [rollErrorSum, pitchErrorSum]';

end
```

```
function get_keypress(~, keyData)
% get_keypress - Determines key that was pressed and outputs
    appropriate
% commands for the simulation

% Key that was pressed. Always will be a lowercase string such as
    'space'
key = keyData.Key;

% initialize outputs
bxInput = 0;
byInput = 0;

if strcmp(key, 'w')
    bxInput = 1;
    disp('w');
elseif strcmp(key, 's')
    bxInput = -1;
    disp('s');
elseif strcmp(key, 'd')
    byInput = 1;
    disp('d');
elseif strcmp(key, 'a')
```

```
    byInput = -1;
    disp('a');
elseif strcmp(key, 'q')
    setappdata(gcf, 'stopsim', 1);
    disp('q');
end
controllerInput = [bxInput, byInput]';
setappdata(gcf, 'controllerInput', controllerInput); % saves
    controllerInput
end
```

```
function [yout, zout] = initialize_sim(plots_flag,
    r_source_inertial, A0)
% initialize_sim - Initialize and run UAV simulation
%
% Syntax: [yout, zout] = initialize_sim(plots_flag,
    r_source_inertial, A0)
%
% INPUTS
%   plots_flag:  ([0] or 1) raise flag for plots. Default is 0
%   r_source_inertial (optional):   [3 x 1] vector containing
    inertial position of
%                                   a radioactive source.
%   A0 (optional):  Sets the activity of the radioactive source
    measured 1cm
%   away from the source. Default is 1000
%
```

```
% OUTPUTS
%   yout:   Structure containing all saved simulation outputs
%   zout:   Structure containing simulated measurements.

if nargin < 1
    plots_flag = 0;
end

% make sure Utilities folder is in path
addpath(genpath(fullfile('..', 'Utilities'))); % add Utilities to
    path
addpath(genpath('./Measurements')); % add measurements to path

UAV = initialize_uav; % initialize uav parameters

% initialize state vector
r_uav_ned = [0, 0, -1]'; % inertial position (m)

% source location relative to inertial frame
if nargin < 2
    randVec = random_vector(10);
    if randVec(3) > -r_uav_ned(3)
        randVec(3) = -randVec(3);
    end
    r_source_inertial = r_uav_ned + randVec;
else
    r_source_inertial = r_uav_ned + r_source_inertial;
```

```
end

vel_uav_body = [0, 0, 0, 0, 0, 0]'; % velocities and angular velocities (m/s and rad/s)
eulerAngles = [deg2rad(0), deg2rad(0), deg2rad(0)]'; % [roll, pitch, yaw] Euler Angles in radians
r_obstacle_ned = [20, 0, -2]'; % Obstacle position. Initialize such that obstacle is not a factor.
obstacle_pos_from_body_ned = r_obstacle_ned - r_uav_ned; % position from body to obstacle in ned
r_uav2source_inertial = (r_source_inertial - r_uav_ned);  % Uav to source vector in inertial frame
x0 = [r_uav_ned; vel_uav_body; eulerAngles; obstacle_pos_from_body_ned; r_uav2source_inertial]; % initial state

% wind velocity
windVel = [0, 0, 0]';

% time settings
t_max = 100;
dt = 0.1; % note that setting this lower than 0.1 will result in a slower-than-realtime simulation
tspan = [0 t_max];

% control settings
% angle gains
```

```
Kp_angle = 3; % proportional gain
Td_angle = 0.5; % derivative sample time
Ti_angle = 15; % time for sum of all past errors to be eliminated
% thrust gain
Kp_thrust = 1 * dt;
% hover control gain
Kp_hover = 1e-1;

deadband_angle = deg2rad(0);    % radians

waypoint1 = [10, 5, 5, -5];
waypoint2 = [30, -10, 10, -0.2];
waypoint3 = [50, 0, 0, -10];
waypoint4 = [70, 20, -20, -1];

fly2source_flag = 0;
waypoints = [waypoint1; waypoint2; waypoint3; waypoint4]';
% waypoints = [];

showRealtimePlot = 0;

A_thresh = 5;
A_background = 1;
if nargin < 3
    A0 = 1000;
end

```

```
% run simulation
yout = run_sim(tspan, dt, x0, windVel, UAV, showRealtimePlot,
    Kp_angle, Td_angle, Ti_angle, Kp_thrust, Kp_hover,
    deadband_angle, r_source_inertial, waypoints, A_thresh,
    A_background, A0, fly2source_flag);
zout = simulate_measurements(yout);
%save([datestr(now, 'yyyy-mm-ddTHHMMSS'), '.mat'])
if plots_flag == 1
%      simulation_plots(yout);
     %measurement_plots(yout, zout);
     measurement_plots(yout);
end
```

```
function many_runs(num_runs, dist_from_source, output_path, A0)

    if nargin < 3
        current_datetime = datestr(now, 'yyyy-mm-ddTHHMMSS');
        output_path = fullfile('.', current_datetime);
    end
    mkdir(output_path);
    for i = 1:num_runs
        disp(['A0 = ', num2str(A0), ' Run number: ', num2str(i)]);
        r_source_inertial = random_vector(dist_from_source);
        r_uav_ned = [0, 0, -1]'; % inertial position (m)
        if r_source_inertial(3) > -r_uav_ned(3)
            r_source_inertial(3) = -r_source_inertial(3);
        end
```

```
        [yout, zout] = initialize_sim(0, r_source_inertial, A0);
        save(fullfile(output_path, ['run', num2str(i), '.mat']), '
            yout', 'zout');
    end


end
```

```
function realtime_plot(x, figureNum, source_pos_ned)
% realtime_plot plot position and orientation of UAV in real time
%
% USAGE: realtime_plot(x)
%
% INPUT
%    x:   state vector

arrowLen = 5; % length modifier for plots

% unpack current state
rNED = [x(1), x(2), x(3)]';
eulerAngles = [x(10), x(11), x(12)]';

% obtain body frame coordinates in NED frame
bodyXInNED = DCM_body2ned([1, 0, 0]', eulerAngles) * arrowLen;
bodyYInNED = DCM_body2ned([0, 1, 0]', eulerAngles) * arrowLen;
bodyZInNED = DCM_body2ned([0, 0, 1]', eulerAngles) * arrowLen;

```

```
figure(figureNum);
plot3(rNED(1), rNED(2), rNED(3));
xlabel('North');
ylabel('East');
zlabel('Down');
title('UAV position and orientation');
hold on;
quiver3(rNED(1), rNED(2), rNED(3), bodyXInNED(1), bodyXInNED(2),
    bodyXInNED(3)); % body x vector in NED
quiver3(rNED(1), rNED(2), rNED(3), bodyYInNED(1), bodyYInNED(2),
    bodyYInNED(3)); % body y vector in NED
quiver3(rNED(1), rNED(2), rNED(3), bodyZInNED(1), bodyZInNED(2),
    bodyZInNED(3)); % body z vector in NED

% plot source position
plot3(source_pos_ned(1), source_pos_ned(2), source_pos_ned(3), 'o'
    );

axis([-30, 30, -30, 30, -30, 0])
hold off
drawnow;
end
```

```
function yout = run_sim(tSpan, dt, x, windVel, UAV,
    showRealtimePlot, KpAngle, KdAngle, KiAngle, KpThrust, KpHover,
     angleDeadband, true_source_pos, waypoints, A_thresh,
    A_background, A0, fly2source_flag)
```

```
% run_sim - runs simulation from given initial conditions
%
% USAGE
%   run_sim(tSpan, dt, xInit, windVel, UAV, rSourceInertial,
    showRealtimePlot)
% INPUTS
%   tSpan:       [1 x 2] or [2 x 1] array containing start and end
    time(seconds)
%   dt:          time step (seconds)
%   x:           [18 x 1] state vector
%   windVel:     [3 x 1] wind velocity vector
%   UAV:         Structure containing UAV constants
%   rSourceInertial:    [3 x 1] inertial position of radiation
    source
%   showRealtimePlot:   1 to show controllable realtime plot, 0 to
      distable
%   KpAngle:     proportional gain for angle control
%   KdAngle:     derivative gain for angle control
%   KiAngle:     integral gain for angle control
%   KpThrust:    Proportional gain for thrust
%   KpHover:     Proportional gain for hover
%   angleDeadband:  angle error were command is not required.
%   true_source_pos: [3 x 1] True source position in NED frame
%   waypoints:   [4 x N] matrix containing waypoints
%   A_thresh:    Activity threshold used to trigger a source
    gradient estimate
%   A_background:   Average background counts per second
```

```
%   A0:           Activity of radioactive source located 1cm away
%   fly2source_flag:     Flag used to command the UAV to disregard other controls
%                          and fly to the estimated source position.
%
% OUTPUTS
%   yout: structure containing all simulated states and additional analysis information

maxLoops = ceil((tSpan(2) - tSpan(1))/dt); % maximum loops in sim
% initialize outputs
xout = zeros(18, maxLoops);
dxdt_out = zeros(18, maxLoops);
tout = zeros(1, maxLoops);
uout = zeros(4, maxLoops);
angle_target_out = zeros(2, maxLoops);
yout = struct();
yout.true_source_pos = true_source_pos;
yout.source.A0 = A0;
yout.source.A_thresh = A_thresh;
yout.source.A_background = A_background;
source.counts = zeros(1, maxLoops);
source_pos_ned_store = [];

% Initialize times
time = tSpan(1);
```

```
time_no_measurements = 0; % time must be greater than
    time_no_measurements for a measurement to be performed

hover_alt = x(3);

% Initialize radiation measurement parameters
% radiation measurement parameters
r0 = 1;      % distance from A0 during A0 measurements (m)
measurement_deadtime = 30;  % minimum time between gradient
    measurements
meas_duration = 30; % duration of measurements at each point
measurement_num = 0;
gradient_store_counts = zeros(6, meas_duration * 10);
gradient_center_ned = [];
time_stop_meas = 0;

measuring_gradient_flag = 0;
r_measurements_ned = zeros(3, 6);
at_target_flag = 0;
counts_1sec = zeros(1, 10);
estimation_start_times = [];

% initialize figure
figureNum = 99; % figure number
figure(figureNum);

angleErrorSum = [0, 0]'; % initialize angle error sum
```

```

j = 1; % loop index
while time < tSpan(2)
    time = round(time, 6);
    %~~~~~~~~~~~~~~~~~~~~~~~~~Source Position Logic ____________________________________
    tempstruct.x = x;
    tempstruct.t = time;
    % Determine counts for this timestep and counts for the last second
    counts_dt = counts_sim(tempstruct, dt, A0, A_background);
    counts_1sec(1:9) = counts_1sec(2:10);
    counts_1sec(10) = counts_dt;

    % detection_flag is raised if the last second of counts are greater
    % than activity threshold A_thresh.
    if fly2source_flag == 1
        detection_flag = check_near_source_counts(counts_1sec, A_thresh, A_background);
    else
        detection_flag = 0;
    end
    %detection_flag = 0;
    if (detection_flag == 1 || measuring_gradient_flag == 1) && (time > time_no_measurements )
        measuring_gradient_flag = 1;
```

```
if measurement_num == 0     % if we are just starting our measurements
    estimation_start_times = [estimation_start_times, time];
    [r_measurements_ned, gradient_center_ned] = get_gradient_waypoints(x);
    measurement_num = 1;
    at_target_flag = 0;
    count_num = 0;
end
% have uav fly to current measurement position.
target_pos_ned = r_measurements_ned(:, measurement_num);
% check if uav is at our target measurement spot
if at_target_flag == 0
    at_target_flag = check_uav_at_target(x, r_measurements_ned(:, measurement_num));
    if at_target_flag == 1
        time_stop_meas = time + meas_duration;
    end
end
% if time_stop_meas is empty and we are at our target, set time_stop_meas
if at_target_flag == 1
    if time >= time_stop_meas
        measurement_num = measurement_num + 1;
        at_target_flag = 0;
        count_num = 0;
```

```
            else     % time <= time_stop_meas
                % if we are at the target, record counts
                count_num = count_num + 1;
                gradient_store_counts(measurement_num, count_num)
                    = counts_dt;
            end
        end
        % if measurement_num == 7, we are done saving measurements
            . Estimate
        % source position from gradient_store_counts
        if measurement_num == 7
            measuring_gradient_flag = 0;
            source_pos_ned = estimate_source_pos(
                gradient_store_counts, gradient_center_ned,
                r_measurements_ned, meas_duration, A0, A_background
                , r0, A_thresh, true_source_pos);
            source_pos_ned_store = [source_pos_ned_store, [time;
                source_pos_ned]];
            time_no_measurements = time + measurement_deadtime; %
                do perform another gradient measurement for this
                time
            measurement_num = 0;
            target_pos_ned = source_pos_ned;
            waypoints = [waypoints, [time; target_pos_ned]];
            % sort waypoints by time
            [~, waypoint_ind] = sort(waypoints(1, :), 2);
            waypoints = waypoints(:, waypoint_ind);
```

```
        end

    end
    % -------------------------------------------------------------------------

    % if we are measuring a gradient, always use those waypoints.
    if measuring_gradient_flag == 0
        target_pos_ned = [];
        % check if we are flying to a waypoint
        if ~isempty(waypoints)
            waypoint_ind = find(time >= waypoints(1, :), 1, 'last'
                );
            if ~isempty(waypoint_ind)
                target_pos_ned = waypoints(2:4, waypoint_ind);
                hover_alt = target_pos_ned(3);
            end
        end
    end
    % if key was pressed, update controllerInput accordingly
    set(figureNum, 'KeyPressFcn', @get_keypress);
    controllerInput = getappdata(figureNum, 'controllerInput');
    if isempty(controllerInput)
        controllerInput = [0, 0]';
    end

```

```
    % controls
    [uDelta, angTarget, angleErrorSum, rollCommand, pitchCommand]
        = ...
        compute_uDelta(controllerInput, x, KpAngle, KdAngle,
            KiAngle, angleDeadband, UAV, dt, angleErrorSum,
            target_pos_ned);
    if ~isempty(target_pos_ned)
        uHover = u_for_hover(x, UAV, KpHover, target_pos_ned,
            rollCommand, pitchCommand);
    else
        uHover = u_for_hover(x, UAV, KpHover, [0, 0, hover_alt]',
            rollCommand, pitchCommand);
    end
    u = uHover + KpThrust .* uDelta;

    setappdata(gcf, 'controllerInput', [0, 0]'); % reset
        controllerInput for future commands

    % get aerodynamic forces and moments
    aeroForcesMomentsInBody = aerodynamic_forces_moments(x,
        windVel, u, UAV);

    % compute state rates
    dxdt = uav_eom(x, aeroForcesMomentsInBody, UAV);

    % store variables
    % variables to store
```

```
    xout(:, j) = x;
    dxdt_out(:, j) = dxdt;
    tout(:, j) = time;
    uout(:, j) = u;
    angle_target_out(:, j) = angTarget;
    source.counts(:, j) = counts_dt;

    % update state
    x = x + dxdt.*dt;
    time = time + dt;
    j = j + 1;

    % terminate loop if UAV has hits ground
    if x(3) >= 0
        break
    end

    % plot output every tenth of a second
    if showRealtimePlot == 1
        realtime_plot(x, figureNum, true_source_pos);
    end

    if getappdata(gcf, 'stopsim') == 1
        break
    end
%     pause(0.01);  I don't think this is needed anymore
end
```

```
% Cut unused array slots if simulation stops early
timeStopped = time;
numIterations = floor((timeStopped - tSpan(1))/dt);
xout = xout(:, 1:numIterations);
dxdt_out = dxdt_out(:, 1:numIterations);
tout = tout(:, 1:numIterations);
uout = uout(:, 1:numIterations);
angle_target_out = angle_target_out(:, 1:numIterations);
source.counts = source.counts(:, 1:numIterations);

% convert all body frame elements to inertial frame
% velocity
vel_ned = DCM_body2ned(xout(4:6, :), xout(10:12, :));
xout(4:6, :) = vel_ned;
% acceleration
accel_ned = DCM_body2ned(dxdt_out(4:6, :), xout(10:12, :));
dxdt_out(4:6, :) = accel_ned;

% Change variables in xout to match the kalman state vector
angvel_body = xout(7:9, :);
xout(7:9, :) = accel_ned;

% add structure with outputs to store
yout.t = round(tout, 6); % to fix overflow
yout.x = xout;
yout.source.position_estimate = source_pos_ned_store;
yout.source.counts = source.counts;
```

```
yout.source.estimation_start_times = estimation_start_times;
yout.angvel_body = angvel_body;
yout.dxdt = dxdt_out;
yout.controls.u = uout;
yout.controls.angle_target = angle_target_out;
yout.controls.waypoints = waypoints;

delete(figure(figureNum));
end
```

```
% Run simulation multiple times and save the results

for j = 1:2
    if j == 1
        source_distances = 1:1:25;
        A0 = 1000;
    elseif j == 2
        A0 = 100;
        source_distances = 1:1:20;
    end
    output_dir = fullfile('D:\Shared\SavedRuns', ['A0_', num2str(
        A0)]);
    num_runs = 100;
    for i = source_distances
        dist_from_source = source_distances(i);
        disp(['Source distance: ', num2str(i)]);
```

```
        output_path = fullfile(output_dir, ['distance_', sprintf('%02i', dist_from_source), 'm_', 'A0_', num2str(A0)]);
        many_runs(num_runs, dist_from_source, output_path, A0);
    end
end
```

```
function uHover = u_for_hover(x, UAV, KpHover, target_pos_ned, rollCommand, pitchCommand)
% u_for_hover - Determine motor speed squared required to maintain constant
% altitude.
%
% INPUTS
%   x: [18 x 1] state vector
%   UAV:         Structure containing UAV physical parameters
%
% OUTPUTS
%   uHover: [4 x 1] array of propeller angular speeds squared

g = 9.8;

% unpack inputs

% body velocities
uBody = x(4);
vBody = x(5);
wBody = x(6);
```

```
% euler angles
phi = x(10);
theta = x(11);
psi = x(12);

if ~isempty(target_pos_ned)
    target_down = target_pos_ned(3);
else
    target_down = x(3);
end

if target_down > -0.1
    target_down = -0.1;
end

% get propeller length
propLength = norm(UAV.propVects(:, 1)); % prop length (m)

[~, ~, ~, density] = atmosisa(-x(3));

uHover = (UAV.mass * g) / (2 * density * (propLength/2)^2 * UAV.
    propArea * UAV.CT * cos(rollCommand) * cos(pitchCommand));

% altitude control
velInertial = DCM_body2ned([uBody, vBody, wBody]', [phi, theta,
    psi]');
```

```
wInertial = velInertial(3);
wError = 0 - wInertial;
zError = target_down - x(3);

uControl = (1 - KpHover*(zError) - 2*KpHover*wError) * uHover;
% output
uHover = uControl .* [1, 1, 1, 1]';

end
```

```
function dxdt = uav_eom(x, A_body, UAV)
% uav_eom - Equations of motion for UAV
%
% USAGE
%   uav_eom(x, A_body, mass, inertiaMatrix)
%
% INPUTS
%   x:       [18 x 1] state vector
%   A_body: [6 x 1] aerodynamic forces and moments in body frame.
    First
%        three terms are forces, last three are moments.
%   UAV:     Structure containing UAV constants
%
% OUTPUTS
%   dxdt:    [18 x 1] state vector derivative with respect to time

```

```
g = 9.8; % gravity (m/s^2)
% unpack inputs
vel_body = [x(4), x(5), x(6)]';
angvel = [x(7), x(8), x(9)]';
eulerangles = [x(10), x(11), x(12)]';

% convert aero forces and moments to inertial frame
F_aero_body = A_body(1:3);
M_aero_body = A_body(4:6);
% outputs
r_dot = DCM_body2ned(vel_body, eulerangles); % inertial frame
v_dot = (DCM_ned2body([0, 0, UAV.mass*g]', eulerangles) + F_aero_body - UAV.mass * cross(angvel, vel_body)) / UAV.mass; % body frame
angvel_dot = ...
    UAV.inertiaMatrix^(-1) * (M_aero_body - cross(angvel, UAV.inertiaMatrix*angvel)); % body frame
if angvel_dot(3) ~= 0
    angvel_dot(3);
end
angvel_to_euler_rates_matrix = angVel_to_eulerRates(eulerangles);
eulerRates = angvel_to_euler_rates_matrix*angvel;
r_obstacle_dot = -DCM_body2ned(vel_body, eulerangles); % inertial frame. Assumes obstacle is stationary
r_uav_to_source_dot = -DCM_body2ned(vel_body, eulerangles);

```

```
dxdt = [r_dot; v_dot; angvel_dot; eulerRates; r_obstacle_dot;
    r_uav_to_source_dot];

end
```

APPENDIX B

MEASUREMENT MODELS

```
function [altitude_measured, t_altimeter] = altimeter_model(yout)
% altimeter_model - Simulate altimeter reading

    % altimeter specs
    [~, ~, ~, altimeter, ~, ~] = get_sensor_specs();

    % unpack yout
    t = yout.t;
    altTruth = -yout.x(3, :);

    t_altimeter = round(t(1): 1/altimeter.sampleRate: t(end), 6);
    altAtTimeStep = interp1(t', altTruth', t_altimeter')';

    altitude_measured = altAtTimeStep + altimeter.sigma .* randn(1, length(altAtTimeStep));

end
```

```
function [counts_recorded, time] = counts_sim(yout, dt, A0, A_background)
%counts_sim - Simulates counts received as a function of source distance
%
% Syntax: counts_recorded = counts_sim(yout)
%
```

```
% INPUTS:
%   yout [struct]
% OUTPUTS:
%   counts_recorded - counts output during time span dt.

    % unpack state
    x = yout.x;
    time = yout.t;
    if length(yout.t) > 1
        dt = yout.t(2) - yout.t(1);
    end
    [~, ~, ~, ~, ~, source_ned] = unpack_state_vector(x);

    source_distance = sqrt(source_ned(1, :).^2 + source_ned(2, :)
       .^2 + source_ned(3, :).^2);

    activity = A0 .* 1 ./ (source_distance.^2); % activity falls
       off as 1/r^2

    % Probability for a count in a given timestep is represented
       using
    % intensity divided by timestep.  Counts must be integers.
    % Example: if intensity / dt = 0.1, then give a 10 %
       probability for a
    % count to be measured.
    countAvePerTimestep = activity .* dt; % average counts in one
       timestep
```

```

    % determine random value for counts
    counts_from_source = poissrnd(countAvePerTimestep);
    counts_from_backround = poissrnd(A_background .* dt);
    counts_recorded = counts_from_source + counts_from_backround;

end
```

```
function [gps_position, gps_velocity, t_gps, gps_course,
    ned_origin_in_lla] = gps_model(yout)
% gps_model - model readings from GPS receiver (MTK3339)
% Syntax: [position, velocity, t_gps, ned_origin_in_lla] =
    gps_model(yout)
%
% INPUTS
%   yout: Simulation output structure
%
% OUTPUTS
%   position: [3 x N] Geodetic position [Lat, Long, Alt] (deg, deg
    , m)
%   groundspeed: [1 x N] Magnitude of horizontal vel (m/s)
%   course: [1 x N] Direction of horizontal vel in NED (degrees)
%   t_gps: [1 x N] time output of GPS measurements (sec)
%   ned_origin_in_lla: [latitude, longitude, altitude]
%       containing origin of local NED frame (degrees, degrees,
    meters)

```

```
% MTK3999 GPS Module
[~, ~, gps, ~, ~, ~] = get_sensor_specs();

ned_origin_in_lla = [44.902236, -68.669829, 0]'; % starting GPS
    position (UMaine mall)

% unpack inputs
t = round(yout.t, 6); % simulation times. Rounded to sixth decimal
    place
gpsTruthIndex = find(mod(t, gps.sampleRate) == 0); % determine
    indices to put in GPS model from truth index
gps_pos_ned_truth = yout.x(1:3, gpsTruthIndex); % [3 x N] array of
    positions in local NED frame (meters)
gps_vel_ne_truth = yout.x(4:5, gpsTruthIndex); % [3 x N] array of
    velocities in local NED frame (meters)
t_gps = round(t(gpsTruthIndex), 6); % measurement times

% gps noise
gps_pos_noise = gps.sigmaPos .* randn(3, length(t_gps));
gps_vel_noise = gps.sigmaVel .* randn(2, length(t_gps));

% gps measurements
gps_position = gps_pos_ned_truth + gps_pos_noise;
gps_velocity = gps_vel_ne_truth + gps_vel_noise;
gps_course = atan2d(gps_velocity(2, :), gps_velocity(1, :));
end
```

```
function [accelerometer_measurements_body, gyro_measurements,
    t_imu, eulerangle_gyro, roll_pitch_accelerometer] = imu_model(
    yout)
% imu_model - Models IMU and outputs accelerometer and gyroscope
    values
%
% INPUTS
%   yout: structure containing simulation outputs
%
% OUTPUTS
%   accelerometer_measurements_body: [3 x N] accelerations in body
      frame (m/s^2)
%   gyro_measurements: [3 x N] angular velocities (rad/s)
%   t_imu: [1 x n] IMU measurement times (s)
%   sampleRate: sample rate of IMU (Hz)

    random_walk_flag = 0; % enable or disable random walk for gyro
         / accelerometer

    % Simulate Measurements
    [accelerometer_measurements_body, t_imu,
        roll_pitch_accelerometer] = accelerometer_model(yout,
        random_walk_flag);
    [gyro_measurements, eulerangle_gyro] = gyro_model(yout,
        random_walk_flag);

end
```

```
%% IMU Composition

function [accelerometer_measured, t_accelerometer,
    roll_pitch_accelerometer] = accelerometer_model(yout,
    random_walk_flag)
% accelerometer_model - outputs measured accelerometer values and
    times
%
% INPUTS
%   yout [struct]: Simulation outputs
%   random_walk_flag [1, 0]: Flag to enable random walk output
% OUTPUTS
%   accelerometer_measured [3 x N]: Simulated accelerometer
    measurements
%   t_accelerometer [1 x N]: Simulated accelerometer measurement
    times

    % Accelerometer Specs
    [accelerometer, ~, ~, ~, ~, ~] = get_sensor_specs();

    % unpack inputs
    t = yout.t;
    t_accelerometer = round(t(1):1/accelerometer.sampleRate:t(end)
        , 6); % measurement times
    eulerangles = yout.x(10:12, :);
    accelerations_from_sim_ned = yout.dxdt(4:6, :);
```

```
accelerations_from_sim_body = DCM_ned2body(accelerations_from_sim_ned + [0; 0; 9.8], eulerangles);

% generate true accelerometer values by interpolating simulation results based
% on accelerometer sample time
accelerometer_truth = interp1(t', accelerations_from_sim_body', t_accelerometer')';

% accelerometer is modeled as a random walk process.
if random_walk_flag == 1
    random_walk_noise = cumsum(randn(3, length(accelerometer_truth)), 2);
else
    random_walk_noise = zeros(3, length(accelerometer_truth));
end
accelerometer_bias = accelerometer.initBias + accelerometer.biasStability .* random_walk_noise;
accelerometer_noise = accelerometer.sigma .* randn(3, length(accelerometer_truth));
accelerometer_measured = accelerometer_truth + accelerometer_noise + accelerometer_bias;

% roll and pitch estimate from accelerometer. These are only accurate when the
% only force present is gravity.
ax = accelerometer_measured(1, :);
```

```
    ay = accelerometer_measured(2, :);
    az = accelerometer_measured(3, :);
    roll = atan2(ay, az);
    pitch = atan2(-ax, sqrt(ay.^2 + az.^2));
    [roll_pitch_accelerometer] = [roll; pitch];
end


function [gyro_measured, eulerAngles_measured] = gyro_model(yout,
    random_walk_flag)
% gyro_model - simulates measured values for gyroscope
%
% INPUTS
%   yout [struct]: Simulation outputs
%   random_walk_flag [1, 0]: Flag to enable random walk output
% OUTPUTS
%   gyro_measured [3 x N]: Simulated gyroscope measurements
%   eulerAngles_measured [3 X N]: Simulated euler angle
    measurements

    % Gyroscope specs
    [~, gyro, ~, ~, ~, ~] = get_sensor_specs();

    % unpack inputs
    t = yout.t;
    t_gyro = round(yout.t(1):1/gyro.sampleRate:yout.t(end), 6); %
        measurement times
```

```
ang_vel_from_sim = yout.angvel_body;

% generate true gyro values by interpolating simulation
    results based
% on gyro sample time
gyro_truth = interp1(t', ang_vel_from_sim', t_gyro')';

% gyro is modeled as a random walk process.
if random_walk_flag == 1
    random_walk_noise = cumsum(randn(3, length(gyro_truth)),
        2);
else
    random_walk_noise = zeros(3, length(gyro_truth));
end
gyro_bias = gyro.initBias + gyro.biasStability .*
    random_walk_noise;
gyro_noise = gyro.sigma .* randn(3, length(gyro_truth));
gyro_measured = gyro_truth + gyro_noise + gyro_bias;

% Get euler angles from measurements
eulerAngles_measured = zeros(3, length(gyro_measured)); %
    initialize array
eulerAngles_measured(:, 1) = yout.x(10:12, 1); % initial Euler
     Angles
for i = 2:length(gyro_measured)
    angVel2EulerMatrix = angVel_to_eulerRates(
        eulerAngles_measured(:, i-1));
```

```
        eulerRatesMeasured = angVel2EulerMatrix * gyro_measured(:, i-1);
        eulerAngles_measured(:, i) = eulerAngles_measured(:, i-1) + eulerRatesMeasured * 1/gyro.sampleRate;
    end

end
```

```
function [lidar_distance_body, t_lidar] = lidar_model(yout)
%lidar_model - Simulate lidar measurement in body frame
%
% Syntax: [lidar_distance_body, t_lidar] = lidar_model(yout)
%
%

% lidar specs
[~, ~, ~, ~, lidar, ~] = get_sensor_specs();

% Lidar measurements available
t_lidar = round(yout.t(1):1/lidar.sampleRate:yout.t(end), 6);
obstacle_distances_from_sim_ned = yout.x(13:15, :);
eulerangles = yout.x(10:12, :);
[obstacle_distances_from_sim_body, ~] = DCM_ned2body(obstacle_distances_from_sim_ned, eulerangles);

% generate uncorrupted lidar values by interpolating simulation results
```

```
lidar_truth = interp1(yout.t', obstacle_distances_from_sim_body',
    t_lidar')';

% lidar measurement
lidar_noise = lidar.sigma .* randn(3, length(lidar_truth));
lidar_distance_body = lidar_truth + lidar_noise;

% currently, lidar only measures in +body x direction
lidar_distance_body = [lidar_distance_body(1, :); zeros(1, length(
    lidar_distance_body)); zeros(1, length(lidar_distance_body))];

% if distance would be negative, set measured distance equal to
    zero
lidar_distance_body(:, lidar_distance_body(1, :) < 0) = 0;
end
```

```
function measurement_plots(yout_truth, zout)
%measurement_plots - plot true state against measurements
%
% Syntax: measurement_plots(yout_truth, zout)

% unpack inputs
t = yout_truth.t;
x = yout_truth.x;
controls = yout_truth.controls;

if nargin == 2
```

```
    imu = zout.imu;
    gps = zout.gps;
    altimeter = zout.altimeter;
    lidar = zout.lidar;
    source = zout.source;
end

% unpack state
[pos_ned_truth, vel_ned_truth, accel_ned_truth, eulerangles_truth, obstacle_ned_truth, uav2source_ned_truth] = unpack_state_vector(x);
angVelTruthInBody = yout_truth.angvel_body;

if nargin == 1

    if ~isempty(yout_truth.source.position_estimate)
        % Source distance estimate error
        source_pos_estimate = [[0;pos_ned_truth(:, 1)], yout_truth.source.position_estimate];
        source_pos_truth_ned = uav2source_ned_truth + pos_ned_truth;
        source_pos_estimate = interp1(source_pos_estimate(1, :)', source_pos_estimate(2:4, :)', t, 'previous', 'extrap')';

        source_pos_error = source_pos_truth_ned - source_pos_estimate;
```

```
        source_dist_error = zeros(1, length(source_pos_error));
        for i = 1:length(source_pos_error)
            source_dist_error(i) = norm(source_pos_error(:, i));
        end
        figure(1)
        plot(t, source_dist_error);
        xlabel('Time (s)')
        ylabel('Distance error (m)');
        title('Source Distance error ')
    end
    if isfield(controls, 'waypoints')
        controls.waypoints = [[t(1); x(1:3, 1)], controls.
           waypoints];
        uav_commanded_pos = interp1(controls.waypoints(1, :)',
           controls.waypoints(2:4, :)', t, 'previous', 'extrap')';
        figure(2);
        subplot(2, 1, 1);
        plot(t, pos_ned_truth(1, :), t, uav_commanded_pos(1, :));
        ylabel('North (m)');
        title('Position vs Time')
        legend('Truth', 'Command')


        subplot(2, 1, 2)
        plot(t, pos_ned_truth(2, :), t, uav_commanded_pos(2, :))
        ylabel('East (m)');
        xlabel('Time (s)');
```

```
        figure(213);
        ned2enu = [0, 1, 0; 1, 0, 0; 0, 0, -1];
        pos_enu_truth = zeros(size(pos_ned_truth));
        uav_commands_enu = zeros(size(pos_ned_truth));
        for i = 1:length(pos_ned_truth)
            pos_enu_truth(:, i) = ned2enu * pos_ned_truth(:, i);
            uav_commands_enu(:, i) = ned2enu * uav_commanded_pos
                (:, i);
        end
        subplot(3, 1, 1);
        plot(t, pos_enu_truth(1, :), t, uav_commands_enu(1, :));
        legend('Truth', 'Command');
        ylabel('East (m)');
        title('Position vs Time');
        subplot(3, 1, 2);
        plot(t, pos_enu_truth(2, :), t, uav_commands_enu(2, :));
        ylabel('North (m)');
        subplot(3, 1, 3);
        plot(t, pos_enu_truth(3, :), t, uav_commands_enu(3, :));
        ylabel('Up (m)')
        xlabel('Time (sec)');



        % Altitude
        figure(3)
```

```
        plot(t, -pos_ned_truth(3, :), t, -uav_commanded_pos(3, :))
            ;
        ylabel('Altitude (m)');
        xlabel('Time (s)');
        title('Altitude vs Time')
        legend('Truth', 'Command')

        % Position error
        pos_error = uav_commanded_pos - pos_ned_truth;
        dist_error = zeros(1, length(pos_error));
        for i = 1:length(pos_error);
            dist_error(i) = norm(pos_error(:, i));
        end

        figure(4)
        subplot(3, 1, 1);
        plot(t, pos_error(1, :));
        ylabel('North Error (m)');
        title('Commanded Position Error vs Time');

        subplot(3, 1, 2);
        plot(t, pos_error(2, :));
        ylabel('East Error (m)');

        subplot(3, 1, 3);
        plot(t, pos_error(3, :));
        ylabel('Down Error (m)');
```

```
        xlabel('Time (s)')

        % Distance error
        figure(5);
        plot(t, dist_error);
        title('Commanded Error Distance vs Time')
        ylabel('Error Distance (m)');
        xlabel('Time (s)')
    else
        figure(2);
        subplot(2, 1, 1);
        plot(t, pos_ned_truth(1, :));
        ylabel('North (m)');
        title('Position vs Time')
        legend('Truth')


        subplot(2, 1, 2)
        plot(t, pos_ned_truth(2, :))
        ylabel('East (m)');
        xlabel('Time (s)');

        figure(3)
        plot(t, -pos_ned_truth(3, :));
        ylabel('Altitude (m)');
        xlabel('Time (s)');
        title('Altitude vs Time')
```

```
        legend('Truth')
    end

    % Velocity

    figure(6);
    subplot(3, 1, 1);
    plot(t, vel_ned_truth(1, :));
    ylabel('v_x (m/s)');
    title('Inertial Velocity vs Time')

    subplot(3, 1, 2)
    plot(t, vel_ned_truth(2, :));
    ylabel('v_y (m/s)');

    subplot(3, 1, 3);
    plot(t, vel_ned_truth(3, :))
    ylabel('v_z (m/s)');
    xlabel('Time (s)');

    % Acceleration
    figure(7);
    subplot(3, 1, 1);
    plot(t, accel_ned_truth(1, :))
    ylabel('a_x (m/s^2)');
    title('NED Acceleration vs Time')

```

```
subplot(3, 1, 2);
plot(t, accel_ned_truth(2, :))
ylabel('a_y (m/s^2)');

subplot(3, 1, 3);
plot(t, accel_ned_truth(3, :))
ylabel('a_z (m/s^2)');
xlabel('Time (s)');

% Angular Velocity
figure(8);
subplot(3, 1, 1);
plot(t, rad2deg(angVelTruthInBody(1, :)))
ylabel('\omega_x (deg/s)');
title('Angular Velocity vs Time');

subplot(3, 1, 2);
plot(t, rad2deg(angVelTruthInBody(2, :)))
ylabel('\omega_y (deg/s)');

subplot(3, 1, 3);
plot(t, rad2deg(angVelTruthInBody(3, :)))
ylabel('\omega_z (deg/s)');

% Euler Angles
figure(9)

```

```
    subplot(2, 1, 1);
    plot(t, rad2deg(eulerangles_truth(1, :)), t, rad2deg(controls.angle_target(1, :)));
    title('Euler Angles vs Time')
    ylabel('Roll (deg)')
    legend('Truth', 'Command');

    subplot(2, 1, 2);
    plot(t, rad2deg(eulerangles_truth(2, :)), t, rad2deg(controls.angle_target(2, :)));
    ylabel('Pitch (deg)')
    xlabel('Time (sec)');

%     subplot(3, 1, 3);
%     plot(t, rad2deg(eulerangles_truth(3, :)))
%     ylabel('Yaw (deg)');
%     xlabel('Time (s)');

    % Euler angle error
    eulerangle_error = controls.angle_target - eulerangles_truth(1:2, :);
    figure(10);
    subplot(2, 1, 1);
    plot(t, rad2deg(eulerangle_error(1, :)));
    ylabel('Roll error (deg)');
    title('Commanded Euler Angle Error vs Time')

```

```
subplot(2, 1, 2);
plot(t, rad2deg(eulerangle_error(2, :)))
ylabel('Pitch error (deg)')
xlabel('Time (s)')

% source position relative to UAV
figure(11);
subplot(3, 1, 1);
plot(t, uav2source_ned_truth(1, :))
ylabel('s_x');
title('NED Source position from UAV vs Time');

subplot(3, 1, 2);
plot(t, uav2source_ned_truth(2, :));
ylabel('s_y');

subplot(3, 1, 3);
plot(t, uav2source_ned_truth(3, :));
xlabel('Time (s)');
ylabel('s_z');

% source distance from UAV
source_dist = zeros(1, length(uav2source_ned_truth));
for i = 1:length(uav2source_ned_truth)
    source_dist(i) = norm(uav2source_ned_truth(:, i));
end
figure(12)
```

```
    plot(t, source_dist)
    xlabel('Time (s)')
    ylabel('Dist (m)');
    title('Source distance from UAV');


end
if nargin == 2
    % euler angles interpolated for same imu timestep
    eulerangles_truth_interpolated = interp1(t', eulerangles_truth
        ', imu.t')';
    % Position
    gpsPosNED = gps.position;

    figure();
    subplot(2, 1, 1);
    plot(t, pos_ned_truth(1, :), gps.t, gpsPosNED(1, :));
    ylabel('North (m)');
    title('Position vs Time')
    legend('Truth', 'gps');

    subplot(2, 1, 2)
    plot(t, pos_ned_truth(2, :), gps.t, gpsPosNED(2, :));
    ylabel('East (m)');
    xlabel('Time (s)');

    % Altitude
```

```
    figure()
    plot(t, -pos_ned_truth(3, :), gps.t, -gpsPosNED(3, :),
        altimeter.t, altimeter.altitude);
    ylabel('Altitude (m)');
    xlabel('Time (s)');
    title('Altitude vs Time')
    legend('Truth', 'gps', 'Altimeter');

    % Velocity
    gpsVelocityInNE = gps.velocity;
    figure();
    subplot(3, 1, 1);
    plot(t, vel_ned_truth(1, :), gps.t, gpsVelocityInNE(1, :));
    ylabel('v_x (m/s)');
    title('Inertial Velocity vs Time')
    legend('Truth', 'gps');

    subplot(3, 1, 2)
    plot(t, vel_ned_truth(2, :), gps.t, gpsVelocityInNE(2, :));
    ylabel('v_y (m/s)');

    subplot(3, 1, 3);
    plot(t, vel_ned_truth(3, :))
    ylabel('v_z (m/s)');
    xlabel('Time (s)');

    % Acceleration
```

```
accelerometer_ned = DCM_body2ned(imu.accelerometer ,
    eulerangles_truth_interpolated);
figure();
subplot(3, 1, 1);
plot(t, accel_ned_truth(1, :), imu.t, accelerometer_ned(1, :))
    ;
ylabel('a_x (m/s^2)');
title('NED Acceleration vs Time')
legend('Truth', 'imu');

subplot(3, 1, 2);
plot(t, accel_ned_truth(2, :), imu.t, accelerometer_ned(2, :))
    ;
ylabel('a_y (m/s^2)');

subplot(3, 1, 3);
plot(t, accel_ned_truth(3, :), imu.t, accelerometer_ned(3, :))
    ;
ylabel('a_z (m/s^2)');
xlabel('Time (s)');

% Angular Velocity
figure();
subplot(3, 1, 1);
plot(t, rad2deg(angVelTruthInBody(1, :)), imu.t, rad2deg(imu.
    gyroscope(1, :)));
ylabel('\omega_x (deg/s)');
```

```
    title('Gyroscope Measurements vs Time');
    legend('Truth', 'imu');

    subplot(3, 1, 2);
    plot(t, rad2deg(angVelTruthInBody(2, :)), imu.t, rad2deg(imu.
        gyroscope(2, :)));
    ylabel('\omega_y (deg/s)');

    subplot(3, 1, 3);
    plot(t, rad2deg(angVelTruthInBody(3, :)), imu.t, rad2deg(imu.
        gyroscope(3, :)));
    ylabel('\omega_z (deg/s)');

    % Euler Angles
    figure()

    subplot(3, 1, 1);
    plot(t, rad2deg(eulerangles_truth(1, :)), imu.t, rad2deg(imu.
        eulerangles(1, :)));
    title('Euler Angles vs Time')
    ylabel('Roll (deg)')
    legend('Actual', 'imu');

    subplot(3, 1, 2);
    plot(t, rad2deg(eulerangles_truth(2, :)), imu.t, rad2deg(imu.
        eulerangles(2, :)));
    ylabel('Pitch (deg)')
```

```
    subplot(3, 1, 3);
    plot(t, rad2deg(eulerangles_truth(3, :)), imu.t, rad2deg(imu.
        eulerangles(3, :)));
    ylabel('Yaw (deg)');
    xlabel('Time (s)');

    % Distance to obstacle
    lidar_dist_ned = DCM_body2ned(lidar.distance,
        eulerangles_truth_interpolated);
    figure();
    subplot(3, 1, 1)
    plot(t, obstacle_ned_truth(1, :), lidar.t, lidar_dist_ned(1,
        :));
    ylabel('\chi_N (m)');
    title('Obstacle position relative to UAV vs Time')

    subplot(3, 1, 2)
    plot(t, obstacle_ned_truth(2, :), lidar.t, lidar_dist_ned(2,
        :));
    ylabel('\chi_E (m)')

    subplot(3, 1, 3)
    plot(t, obstacle_ned_truth(3, :), lidar.t, lidar_dist_ned(3,
        :));
    ylabel('\chi_D (m)');
    xlabel('Time (s)');
```

```
    legend('Truth', 'Lidar');

    % counts plot
    figure();
    plot(t, source.counts);
    xlabel('Time (s)');
    ylabel('Counts');
    title('Counts vs Time');

    % source position relative to UAV
    figure();
    subplot(3, 1, 1);
    plot(t, uav2source_ned_truth(1, :))
    ylabel('s_x');
    title('NED Source position from UAV vs Time');

    subplot(3, 1, 2);
    plot(t, uav2source_ned_truth(2, :));
    ylabel('s_y');

    subplot(3, 1, 3);
    plot(t, uav2source_ned_truth(3, :));
    xlabel('Time (s)');
    ylabel('s_z');
end

end
```

```
function [zout] = simulate_measurements(yout)
%simulate_measurements - simulate measurements from truth
%
% Syntax: zout = simulate_measurements(yout)
%
% zout contents:
%   zout.imu.[accelerometer, gyroscope, t, eulerangles]
%   zout.gps.[position, velocity, t, ned_origin_in_lla]
%   zout.altimeter.[altitude, t]
%   zout.lidar.[measurement, t]

% initialize structures
zout = struct('imu', [], 'gps', []);
imu = struct();
gps = struct();
altimeter = struct();
lidar = struct();
source = struct();

% simulate measurements
[imu.accelerometer, imu.gyroscope, imu.t, imu.eulerangles, imu.
    accelerometer_roll_pitch] = imu_model(yout); % simulate
    accelerometer measurements

[gps.position, gps.velocity, gps.t, gps.course, gps.
    ned_origin_in_lla] ...
```

```
    = gps_model(yout); % simulate gps measurements

[altimeter.altitude, altimeter.t] = altimeter_model(yout); % simulate altimeter measurements (m)

[lidar.distance, lidar.t] = lidar_model(yout); % simulate lidar distances

%[source.counts, source.t] = counts_sim(yout); % simulate counts
source.counts = yout.source.counts;
source.t = yout.t;
source.position_estimate = yout.source.position_estimate;

% create structures
zout.imu = imu;
zout.gps = gps;
zout.altimeter = altimeter;
zout.lidar = lidar;
zout.source = source;
end
```

APPENDIX C

KALMAN FILTER

```
function P0 = initial_covariance_matrix()
% initial_covariance_matrix - outputs initial covariance matrix
    based on
% IMU variance

% populate sensor specs used in covariance matrix
[~, gyro, ~, ~, ~, ~] = get_sensor_specs();

P0_pos = zeros(3, 18);
P0_vel = zeros(3, 18);
P0_accel = zeros(3, 18);
P0_euler = zeros(3, 18);
P0_chi = zeros(3, 18);
P0_source = zeros(3, 18);

P0 = vertcat(P0_pos, P0_vel, P0_accel, P0_euler, P0_chi, P0_source
    );

end
```

```
function H = linearized_measurement_matrix(x1_minus, hx)
% linearized_measurement_matrix - linearize measurement function
    about state
% estimate
%
```

```
% Syntax: H = linearized_measurement_matrix(x1_minus, hx)
%
% INPUTS
%   x1_minus [18 x 1]: state vector for kth state without
    measurement update
%

    % unpack state
    [~, ~, accel_ned, eulerangles, obstacle_ned, ~] =
        unpack_state_vector(x1_minus);

    % GPS included
    if hx(1) ~= 0
        H_gps = linearized_h_gps();
    else
        H_gps = zeros(5, 18);
    end

    % altimeter included
    if hx(6) ~= 0
        H_altimeter = linearized_h_altimeter();
    else
        H_altimeter = zeros(1, 18);
    end

    % lidar
    if hx(7) ~= 0
```

```
        H_lidar = linearized_h_lidar(eulerangles, obstacle_ned);
    else
        H_lidar = zeros(1, 18);
    end

    % imu
    if hx(8) ~= 0
        H_imu = linearized_h_imu(accel_ned, eulerangles);
    else
        H_imu = zeros(5, 18);
    end

    % if we are accelerating, we can't use gravity for attitude
        measurements.
    if hx(11) == 0
        H_imu(4:5, :) = zeros(2, 18);
    end

    % source of interest
    H_source = linearized_h_source(x1_minus);

    % create H
    H = vertcat(H_gps, H_altimeter, H_lidar, H_imu, H_source);

end

function H_gps = linearized_h_gps()
```

```
    % linearize gps terms in measurement function about x1_minus

    dpos_dx = [eye(3), zeros(3, 15)];
    dvel_dx = [zeros(2, 3), eye(2, 3), zeros(2, 12)];

    %% Linearized GPS Measurement Matrix

    H_gps = vertcat(dpos_dx, dvel_dx);
end

function H_altimeter = linearized_h_altimeter()
% linearize altimeter term in measurement function about x1_minus

    H_altimeter = [0, 0, -1, zeros(1, 15)];

end

function H_lidar = linearized_h_lidar(eulerangles, chi)
%linearized_h_lidar - linearize lidar terms in measurement
    function about x1_minus
%
% Syntax: H_lidar = linearized_h_lidar(x1_minus)
%
% h_{LIDAR} = chi(1)*cos(theta)*cos(psi) + chi(2)*cos(theta)*sin(
    psi) - chi(3)*sin(theta)

    % unpack euler angles
```

```
    theta = eulerangles(2);
    psi = eulerangles(3);
    % chi terms
    dLidar_dChi = [cos(theta)*cos(psi), cos(theta)*sin(psi), -sin(
        theta)];

    % euler angle terms
    dLidar_dphi = 0;
    dLidar_dtheta = -chi(1)*sin(theta)*cos(psi) - chi(2)*sin(theta
        )*sin(psi) - chi(3)*cos(theta);
    dLidar_dpsi = -chi(1)*cos(theta)*sin(psi) + chi(2)*cos(theta)*
        cos(psi);
    dLidar_deuler = [dLidar_dphi, dLidar_dtheta, dLidar_dpsi];

    H_lidar = [zeros(1, 9), dLidar_deuler, dLidar_dChi, zeros(1,
        3)];
end

function H_imu = linearized_h_imu(accel_ned, eulerangles)

    % body acceleration
    daccel_dpos = zeros(3);
    daccel_dvel = zeros(3);
    [~, daccel_daccel] = DCM_ned2body([1, 1, 1]', eulerangles);
    [d_bTi_d_phi, d_bTi_d_theta, d_bTi_d_psi] =
        jacobian_DCM_ned2body(eulerangles);
```

```
    daccel_deuler = [d_bTi_d_phi * accel_ned, d_bTi_d_theta * accel_ned, d_bTi_d_psi * accel_ned];
    daccel_dobstacle = zeros(3);
    daccel_dsource = zeros(3);

    H_imu_accelerometer = ...
        [daccel_dpos, daccel_dvel, daccel_daccel, daccel_deuler, daccel_dobstacle, daccel_dsource];

    % accelerometer roll and pitch
    H_accelerometer_roll = [zeros(1, 9), 1, zeros(1, 8)];
    H_accelerometer_pitch = [zeros(1, 10), 1, zeros(1, 7)];

    H_imu = ...
        [H_imu_accelerometer; H_accelerometer_roll; H_accelerometer_pitch];

end

function H_source = linearized_h_source(~)
%linearized_h_source - linearize source of interest term in measurement function
%
% Syntax: H_source = linearized_h_source(x1_minus)
%
% not used yet
```

```
    H_source = [zeros(4, 15), [0, 0, 0; 1, 0, 0; 0, 1, 0; 0, 0,
        1]];
end
```

```
function F = linearized_state_update_matrix(x0, dt)
% linearized_state_update_matrix - state update matrix used to
    propagate
%   covariance matrix (P0).

    % Make sure vel0 is a column vector. If it's not, throw error
    if size(x0, 2) ~= 1
        error('x must be a column vector');
    end

    % create parts of update matrix
    F_pos = get_position_update_matrix(dt);
    F_vel = get_velocity_update_matrix(dt);
    F_accel = get_acceleration_update_matrix();
    F_euler = get_eulerAngles_update_matrix();
    F_obstacle = get_obstacle_update_matrix(dt);
    F_source = get_source_update_matrix();

    % combine smaller matrices into complete update matrix
    F = vertcat(...
        F_pos, F_vel, F_accel, F_euler, F_obstacle, F_source);
end

```

```
function F_pos = get_position_update_matrix(dt)
% get_position_update_matrix - compute position part of update matrix

    % matrix terms refer to the term that is multiplied by the state.
    pos_term = eye(3);
    vel_term = dt .* eye(3);
    accel_term = 1/2 .* dt^2 .* eye(3);
    eulerangle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    F_pos = ...
        [pos_term, vel_term, accel_term, eulerangle_term, obstacle_term, source_term];

end

function F_vel = get_velocity_update_matrix(dt)
% get_velocity_update_matrix - compute velocity part of update matrix

    pos_term = zeros(3);
    vel_term = eye(3);
    accel_term = dt .* eye(3);
    eulerangle_term = zeros(3);
```

```
    obstacle_term = zeros(3);
    source_term = zeros(3);

    F_vel = ...
        [pos_term, vel_term, accel_term, eulerangle_term,
          obstacle_term, source_term];
end

function F_accel = get_acceleration_update_matrix()
    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = eye(3);
    eulerangle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    F_accel = ...
        [pos_term, vel_term, accel_term, eulerangle_term,
          obstacle_term, source_term];
end

function F_euler = get_eulerAngles_update_matrix()

    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerangle_term = eye(3);
```

```
    obstacle_term = zeros(3);
    source_term = zeros(3);

    F_euler = ...
        [pos_term, vel_term, accel_term, eulerangle_term,
           obstacle_term, source_term];
end

function F_obstacle = get_obstacle_update_matrix(dt)

    pos_term = zeros(3);
    vel_term = -dt .* eye(3);
    accel_term = zeros(3);
    eulerangle_term = zeros(3);
    obstacle_term = eye(3);
    source_term = zeros(3);

    F_obstacle = ...
        [pos_term, vel_term, accel_term, eulerangle_term,
           obstacle_term, source_term];

end

function F_source = get_source_update_matrix()
% this function does nothing for now

    pos_term = zeros(3);
```

```
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerangle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    F_source = ...
        [pos_term, vel_term, accel_term, eulerangle_term, obstacle_term, source_term];
end
```

```
function R = measurement_covariance_matrix()
%measurement_covariance_matrix - defines measurement covariance matrix
%
% Syntax: R = measurement_covariance_matrix()
%
% INPUTS
%   ySize: Used to determine size of R matrix

% get sensor specification structures
[accelerometer, ~, gps, altimeter, lidar, source] = get_sensor_specs();

% gps terms
sigma_gps_pos = gps.sigmaPos;
sigma_gps_vel = gps.sigmaVel; % gps vel (m/s)
```

```
% altimeter term
sigma_altimeter = altimeter.sigma; % altimeter altitude (m)

% lidar term
sigma_lidar = lidar.sigma; % lidar distance (m)

% accelerometer term
sigma_accelerometer = accelerometer.sigma; % m/s^2
sigma_accelerometer_roll_pitch = 0.1; % rad/s^2
% source terms
sigma_source = source.sigma;

% R matrix
R = diag([...
    sigma_gps_pos, sigma_gps_pos, sigma_gps_pos, sigma_gps_vel, sigma_gps_vel, ...
    sigma_altimeter, sigma_lidar, sigma_accelerometer, sigma_accelerometer, sigma_accelerometer, ...
    sigma_accelerometer_roll_pitch, sigma_accelerometer_roll_pitch, ...
    sigma_source, sigma_source, sigma_source, sigma_source]).^2;

end
```

```
function [region_struct, region_num, source_pos_estimate] =
    measurement_inside_region(region_struct, x0_plus, counts,
    region_num)
%measurement_inside_region - Record counts, position, and time
    inside a region
%
% Syntax: [counts, pos_ned, time] = measurement_inside_region()
%
%

    % unpack inputs
    [uav_pos_ned, ~, ~, ~, ~, ~] = unpack_state_vector(x0_plus);

    % initialize source_pos_estimate
    source_pos_estimate = [];
    % set thresholds to define region
    region_diameter = 0.1;    % 1m diameter region. This means a
        0.5m deviation from r_region will stop that region.

    if region_num == 0
        if ~isempty(region_struct(1).center)
            error('Region is set, but region number is zero.')
        else
            region_num = 1;
            region_struct(region_num).center = uav_pos_ned;
        end
    end
```

```
    % distance from region center to uav
    dist_region_center_to_uav = norm(uav_pos_ned - region_struct(
        region_num).center);

    % if we are out of the region, increment region_num and save
        the center of
    % the region
    if dist_region_center_to_uav > region_diameter
        region_num = region_num + 1;
        if region_num <= 3
            region_struct(region_num).center = uav_pos_ned;
        end
    end

    % If our region num is less than or equal to three, save the
        uav position
    % and counts
    % if region_num <= 3
    %     region_struct(region_num).uav_pos_ned = [region_struct(
        region_num).uav_pos_ned, uav_pos_ned];
    %     region_struct(region_num).counts = [region_struct(
        region_num).counts, counts];
    % else % else, compute source position estimate
    %     A0 = 100;    % 100 counts per second
    %     A_background = 10;

```

```
%     % determine region with smallest number of counts
%     region_least_counts = min([length(region_struct(1).counts), length(region_struct(2).counts), length(region_struct(3).counts)]);

%     avg_counts_per_sec = zeros(1, 3);
%     for i = 1:3
%         secs_in_region = length(region_struct(i).counts) / 100;  % 10 counts per second
%         avg_counts_per_sec(i) = sum(region_struct(i).counts) / secs_in_region;
%     end

%     activity_ratio = (A0 - A_background) ./ avg_counts_per_sec;

%     source_pos_estimate = [0, 0, 0]';
%     for i = 3:-1:2
%         for j = 2:-1:1
%             source_pos_estimate = source_pos_estimate + (activity_ratio(i) .* region_struct(i).center./norm(region_struct(i).center)) - (activity_ratio(j) .* region_struct(j).center./norm(region_struct(j).center));
%         end
%     end

```

```
        % all_pos = [region_struct(1).uav_pos_ned, region_struct
            (2).uav_pos_ned, region_struct(3).uav_pos_ned];
        % all_counts = [region_struct(1).counts, region_struct(2).
            counts, region_struct(3).counts];

        % all_unit_pos = zeros(3, length(all_pos));
        % for i = 1:length(all_pos)
        %     all_unit_pos(:, i) = all_pos(:, i) ./ norm(all_pos
            (:, i));
        % end

        % activity_ratio = (1./all_counts .* (A0-A_background))
            .^(1/2);

        % source_pos_estimate = sum(diff(activity_ratio, 1, 2) .*
            diff(all_unit_pos, 1, 2), 2);

        % reset region_struct and region_num
        region_struct = struct('center', [], 'uav_pos_ned', [], '
            counts', []);
        region_num = 0;
    end

end
```

```
function [x1_plus, P1_plus] = measurement_update(x1_minus,
    P1_minus, y1)
```

```
%measurement_update - Perform measurement update step
%
% Syntax: [x1_plus, P1_plus] = measurement_update(x1_minus,
    P1_minus)
%
% INPUTS
%   x1_minus [18 x 1]: state vector for kth state without
    measurement update
%   P1_minus [18 x 18]: covariance matrix for kth state without
    measurement
%        update
%   y1 [N x 1]: measurement vector for kth state. Max size = 11,
    min size = 6
%   originLLA [3 x 1]: origin of NED position as [latitude,
    longitude, altitude]
%
% OUTPUTS
%   x1_plus [18 x 1]: best state vector estimate for kth state
%   P1_plus [18 x 18]: best error covariance matrix estimate

    % Check if measurement_update can be performed. Skip if all
        measurement
    % values have been set to zero.
    if any(y1 ~= 0)
        % obtain state variables in measurement units
        hx = state_to_measurement(x1_minus, y1);

```

```
        % construct state update matrix
        H = linearized_measurement_matrix(x1_minus, hx);

        % obtain measurement covariance matrix
        R = measurement_covariance_matrix();

        % Remove rows in H and R if there are no respective measurements.
        % Remove elements in hx
        % REMOVE IF USING COUNTS HERE
        y1(13) = 0;
        hx(y1 == 0) = [];
        H(y1 == 0, :) = [];
        R(y1 == 0, :) = [];
        R(:, y1 == 0) = [];
        y1(y1 == 0) = [];

        % compute Kalman Gain
        K = P1_minus * H' * (H * P1_minus * H' + R)^(-1);

        % measurement update of state estimate
        x1_plus = x1_minus + K * (y1 - hx);

        % measurement update of error covariance
        P1_plus = (eye(length(P1_minus)) - K*H) * P1_minus;
    else % no measurements are available. Skip measurement step
        x1_plus = x1_minus;
```

```
            P1_plus = P1_minus;
        end
end
```

```
function plots_kalmanfilter(yout_truth, yout_kalman, zout)
%plots_kalmanfilter - plot kalman filter outputs against truth
   outputs
%
% Syntax: plots_kalmanfilter(yout_truth, yout_kalman)
%
% Long description

    % unpack inputs
    % truth state
    x_truth = yout_truth.x;
    accel_truth = yout_truth.dxdt(4:6, :);
    t_truth = yout_truth.t;

    % kalman filter state
    x_kalman = yout_kalman.x;
    t_kalman = yout_kalman.t;

    ned2enu = [0, 1, 0; 1, 0, 0; 0, 0, -1];

    if nargin > 2
        % measurements
        gps = zout.gps;
```

```
        lidar = zout.lidar;
        imu = zout.imu;
        source = zout.source;
    end

    if nargin == 2

        % Position
        figure();
        subplot(2, 1, 1 );
        plot(t_truth, x_truth(1, :), t_kalman, x_kalman(1, :));
        ylabel('North (m)');
        title('Position vs Time')
        legend('Truth', 'Kalman')

        subplot(2, 1, 2);
        plot(t_truth, x_truth(2, :), t_kalman, x_kalman(2, :));
        ylabel('East (m)');
        xlabel('Time (s)');

        figure()
        posTruthNED = ned2enu * x_truth(1:3, :);
        posKalmanNED = ned2enu * x_kalman(1:3, :);
        subplot(3, 1, 1);
        plot(t_truth, posTruthNED(1, :), t_kalman, posKalmanNED(1,
            :));
        ylabel('East (m)');
```

```
        title('Position vs Time');
        legend('Truth', 'Kalman');
        subplot(3, 1, 2);
        plot(t_truth, posTruthNED(2, :), t_kalman, posKalmanNED(2,
            :));
        ylabel('North (m)');
        subplot(3, 1, 3);
        plot(t_truth, posTruthNED(3, :), t_kalman, posKalmanNED(3,
            :));
        xlabel('Time (sec)');
        ylabel('Up (m)');

        % Altitude
        figure()
        plot(t_truth, -x_truth(3, :), t_kalman, -x_kalman(3, :));
        ylabel('Altitude (m)');
        xlabel('Time (s)');
        title('Altitude vs Time')
        legend('Truth', 'Kalman');

        % velocity
        figure();
        subplot(3, 1, 1);
        plot(t_truth, x_truth(4, :), t_kalman, x_kalman(4, :));
        ylabel('v_x (m/s)');
        title('Inertial Velocity vs Time')
        legend('Truth', 'Kalman');
```

```

    subplot(3, 1, 2)
    plot(t_truth, x_truth(5, :), t_kalman, x_kalman(5, :));
    ylabel('v_y (m/s)');

    subplot(3, 1, 3)
    plot(t_truth, x_truth(6, :), t_kalman, x_kalman(6, :));
    ylabel('v_z (m/s');
    xlabel('Time (s)');

    % Accelerations
    figure();
    subplot(3, 1, 1);
    plot(t_truth, accel_truth(1, :), t_kalman, x_kalman(7, :))
        ;
    title('Inertial Acceleration vs Time')
    ylabel('a_x (m/s^2)')
    legend('Truth', 'Kalman');

    subplot(3, 1, 2);
    plot(t_truth, accel_truth(2, :), t_kalman, x_kalman(8, :))
        ;
    ylabel('a_y (m/s^2)');

    subplot(3, 1, 3);
    plot(t_truth, accel_truth(3, :), t_kalman, x_kalman(9, :))
        ;
```

```
        ylabel('a_z (m/s^2)');
        xlabel('Time (s)');

        % Euler angles
        figure();
        subplot(3, 1, 1);
        plot(t_truth, rad2deg(x_truth(10, :)), t_kalman, rad2deg(x_kalman(10, :)));
        ylabel('Roll (deg)');
        title('Euler Angles vs Time')
        legend('Truth', 'Kalman');

        subplot(3, 1, 2);
        plot(t_truth, rad2deg(x_truth(11, :)), t_kalman, rad2deg(x_kalman(11, :)));
        ylabel('Pitch (deg)');

        subplot(3, 1, 3);
        plot(t_truth, rad2deg(x_truth(12, :)), t_kalman, rad2deg(x_kalman(12, :)));
        ylabel('Yaw (deg)');
        xlabel('Time (s)');

        %% Obstacle distance
        % figure();
        % subplot(3, 1, 1);
```

```
    % plot(t_truth, x_truth(13, :), t_kalman, x_kalman(13, :));
    % ylabel('\chi_N (m)')
    % title('Inertial Obstacle distance vs Time')
    % legend('Truth', 'Kalman');

    % subplot(3, 1, 2);
    % plot(t_truth, x_truth(14, :), t_kalman, x_kalman(14, :));
    % ylabel('\chi_E (m)')

    % subplot(3, 1, 3);
    % plot(t_truth, x_truth(15, :), t_kalman, x_kalman(15, :));
    % ylabel('\chi_D (m)');
    % xlabel('Time (s)')

    % source
    % source position relative to UAV
    source_pos_ned = x_truth(1:3, 1) + x_truth(16:18, 1);
    kalman_source_pos_from_uav = [source_pos_ned(1) - x_kalman(1, :); source_pos_ned(2) - x_kalman(2, :); source_pos_ned(3) - x_kalman(3, :)];
    figure();
    subplot(3, 1, 1);
    plot(t_truth, x_truth(16, :), t_kalman, kalman_source_pos_from_uav(1, :))
```

```
        ylabel('s_x');
        title('NED Source position from UAV vs Time');

        subplot(3, 1, 2);
        plot(t_truth, x_truth(17, :), t_kalman,
           kalman_source_pos_from_uav(2,  :))
        ylabel('s_y');

        subplot(3, 1, 3);
        plot(t_truth, x_truth(18, :), t_kalman,
           kalman_source_pos_from_uav(3, :))
        xlabel('Time (s)');
        ylabel('s_z');
        legend('Truth', 'Kalman')

        % source distance from UAV
        source_dist_truth = zeros(1, length(x_truth(15, :)));
        source_dist_kalman= zeros(1, length(x_kalman(15, :)));
        for i = 1:length(x_truth)
            source_dist_truth(i) = norm(x_truth(16:18, i));
        end
        for i = 1:length(x_kalman)
            source_dist_kalman(i) = norm(
               kalman_source_pos_from_uav(1:3, i));
        end
        figure()
```

```
        plot(t_truth, source_dist_truth, t_kalman,
            source_dist_kalman)
        xlabel('Time (s)')
        ylabel('Dist (m)');
        title('Source distance from UAV');
        legend('Truth', 'Kalman');

    end

    if nargin > 2
        gps_pos_ned = gps.position;
        figure();
        subplot(2, 1, 1);
        plot(t_truth, x_truth(1, :), t_kalman, x_kalman(1, :), gps
            .t, gps_pos_ned(1, :));
        ylabel('North (m)');
        title('Position vs Time')
        legend('Truth', 'Kalman', 'GPS')

        subplot(2, 1, 2);
        plot(t_truth, x_truth(2, :), t_kalman, x_kalman(2, :), gps
            .t, gps_pos_ned(2, :));
        ylabel('East (m)');
        xlabel('Time (s)');

        % Altitude
        figure()
```

```
        plot(t_truth, -x_truth(3, :), t_kalman, -x_kalman(3, :), gps.t, -gps_pos_ned(3, :));
        ylabel('Altitude (m)');
        xlabel('Time (s)');
        title('Altitude vs Time')
        legend('Truth', 'Kalman', 'GPS');

        % velocity
        figure();
        subplot(3, 1, 1);
        plot(t_truth, x_truth(4, :), t_kalman, x_kalman(4, :));
        ylabel('v_x (m/s)');
        title('Inertial Velocity vs Time')
        legend('Truth', 'Kalman');

        subplot(3, 1, 2)
        plot(t_truth, x_truth(5, :), t_kalman, x_kalman(5, :));
        ylabel('v_y (m/s)');

        subplot(3, 1, 3)
        plot(t_truth, x_truth(6, :), t_kalman, x_kalman(6, :));
        ylabel('v_z (m/s');
        xlabel('Time (s)');

        % Accelerations
        figure();
        subplot(3, 1, 1);
```

```
    plot(t_truth, accel_truth(1, :), t_kalman, x_kalman(7, :));
    title('Inertial Acceleration vs Time')
    ylabel('a_x (m/s^2)')
    legend('Truth', 'Kalman');

    subplot(3, 1, 2);
    plot(t_truth, accel_truth(2, :), t_kalman, x_kalman(8, :));
    ylabel('a_y (m/s^2)');

    subplot(3, 1, 3);
    plot(t_truth, accel_truth(3, :), t_kalman, x_kalman(9, :));
    ylabel('a_z (m/s^2)');
    xlabel('Time (s)');

    % Euler angles
    figure();
    subplot(3, 1, 1);
    plot(t_truth, rad2deg(x_truth(10, :)), t_kalman, rad2deg(x_kalman(10, :)));
    ylabel('Roll (deg)');
    title('Euler Angles vs Time')
    legend('Truth', 'Kalman');

    subplot(3, 1, 2);
```

```
        plot(t_truth, rad2deg(x_truth(11, :)), t_kalman, rad2deg(
            x_kalman(11, :)));
        ylabel('Pitch (deg)');

        subplot(3, 1, 3);
        plot(t_truth, rad2deg(x_truth(12, :)), t_kalman, rad2deg(
            x_kalman(12, :)));
        ylabel('Yaw (deg)');
        xlabel('Time (s)');

        % Obstacle distance
        lidar_dist_ned = DCM_body2ned(lidar.distance, imu.
            eulerangles);
        figure();
        subplot(3, 1, 1);
        plot(t_truth, x_truth(13, :), t_kalman, x_kalman(13, :),
            lidar.t, lidar_dist_ned(1, :));
        ylabel('\chi_N (m)')
        title('Inertial Obstacle distance vs Time')
        legend('Truth', 'Kalman', 'LIDAR');

        subplot(3, 1, 2);
        plot(t_truth, x_truth(14, :), t_kalman, x_kalman(14, :),
            lidar.t, lidar_dist_ned(2, :));
        ylabel('\chi_E (m)')

        subplot(3, 1, 3);
```

```
        plot(t_truth, x_truth(15, :), t_kalman, x_kalman(15, :),
            lidar.t, lidar_dist_ned(3, :));
        ylabel('\chi_D (m)');
        xlabel('Time (s)')

        % source estimate
        % source position relative to UAV
        figure();
        subplot(3, 1, 1);
        plot(t_truth, x_truth(16, :), t_kalman, x_kalman(16, :))
        ylabel('s_x');
        title('NED Source position from UAV vs Time');

        subplot(3, 1, 2);
        plot(t_truth, x_truth(17, :), t_kalman, x_kalman(17, :));
        ylabel('s_y');

        subplot(3, 1, 3);
        plot(t_truth, x_truth(18, :), t_kalman, x_kalman(18, :));
        xlabel('Time (s)');
        ylabel('s_z');

        % counts plot
        figure();
        plot(source.t, source.counts);
        xlabel('Time (s)');
        ylabel('Counts');
```

```
        title('Counts vs Time');
    end
end
```

```
function Q = process_noise_matrix(dt)
% process_noise_matrix - outputs process noise matrix

    k = 0.1; % manually tuned gain for filter

    [~, gyro, ~, ~, ~, ~] = get_sensor_specs();

    % process noise parts
    Q_pos = position_process_noise();
    Q_vel = velocity_process_noise();
    Q_acceleration = acceleration_process_noise();
    Q_euler = euler_angle_process_noise(dt, gyro);
    Q_obstacle = obstacle_process_noise();
    Q_source = source_process_noise();

    % process noise matrix
    Q = k .* vertcat(Q_pos, Q_vel, Q_acceleration, Q_euler,
        Q_obstacle, Q_source);
end

function Q_pos = position_process_noise()
% position part of process noise matrix

```

```
    k_pos = 0;
    pos_term = k_pos .* eye(3);
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerAngle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);
    Q_pos = [pos_term, vel_term, accel_term, eulerAngle_term,
        obstacle_term, source_term];

end

function Q_vel = velocity_process_noise()
% velocity part of process noise matrix

    k_vel = 0.1;
    pos_term = zeros(3);
    vel_term = k_vel .* eye(3);
    accel_term = zeros(3);
    eulerAngle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    Q_vel = [pos_term, vel_term, accel_term, eulerAngle_term,
        obstacle_term, source_term];
end

```

```
function Q_acceleration = acceleration_process_noise()
    k_accel = 0.1;
    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = k_accel .* eye(3);
    eulerAngle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    Q_acceleration = [pos_term, vel_term, accel_term,
       eulerAngle_term, obstacle_term, source_term];
end

function Q_euler = euler_angle_process_noise(dt, gyro)
    k_euler = 1;
    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerAngle_term = 1*k_euler .* gyro.sigma^2 .* dt^2 .* diag
       ([10, 10, .1]);
    obstacle_term = zeros(3);
    source_term = zeros(3);

    Q_euler = [pos_term, vel_term, accel_term, eulerAngle_term,
       obstacle_term, source_term];
end
```

```
function Q_obstacle = obstacle_process_noise()
    k_obstacle = 0.1;
    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerAngle_term = zeros(3);
    obstacle_term = k_obstacle .* eye(3);
    source_term = zeros(3);
    Q_obstacle = [pos_term, vel_term, accel_term, eulerAngle_term,
        obstacle_term, source_term];

end

function Q_source = source_process_noise()
    pos_term = zeros(3);
    vel_term = zeros(3);
    accel_term = zeros(3);
    eulerAngle_term = zeros(3);
    obstacle_term = zeros(3);
    source_term = 10 .* eye(3);
    Q_source = [pos_term, vel_term, accel_term, eulerAngle_term,
       obstacle_term, source_term];
end
```

```
function [x_out, P_out, t_out] = run_kalman_filter(x0_plus,
   P0_plus, tspan, zout)
%run_kalman_filter - Run kalman filter from initial inputs
```

```
%
% Syntax: [x_out, P_out, t_out] = run_kalman_filter(x, P, tspan)
%
% Long description

% Need accelerometer and gyro specs

    [~, gyro_specs, ~, ~, ~, ~] = get_sensor_specs();

    % Populate UAV specs
    UAV = initialize_uav();

    % time settings
    t = tspan(1);
    t_final = tspan(2);
    dt = 1/gyro_specs.sampleRate; % timestep should be set to
        accelerometer sample time.
    t_out = t:dt:t_final; % array of times

    % unpack imu measurements
    gyro_meas = zout.imu.gyroscope;
    imu_index = find(zout.imu.t == t); % where to start imu
        measurements

    % initialize outputs
    x_out = zeros(18, length(t_out));
    P_out = zeros(18, 18, length(t_out));
```

```
% run kalman filter for the length of array t_out
t = t + dt; % kalman filter needs to start at t0 + dt
for kalman_index = (imu_index + 1):length(t_out)

    % obtain imu measurement and perform time update
    gyro_reading = gyro_meas(:, kalman_index - 1);
    [x1_minus, P1_minus] = time_update(x0_plus, P0_plus,
        gyro_reading, dt, UAV);

    % update
    if mod(t, 0.01) == 0
        % obtain measurement vector
        [y1] = simulate_measurement_vector(t, zout);
        % perform measurement update
        [x1_plus, P1_plus] = measurement_update(x1_minus,
            P1_minus, y1);
    else
        x1_plus = x1_minus;
        P1_plus = P1_minus;
    end

    % update current time
    t = t + dt;
    t = round(t, 6);

    % store state and covariance matrix
```

```
        x_out(:, kalman_index) = x1_plus;
        P_out(:, :, kalman_index) = P1_plus;

        x0_plus = x1_plus;
        P0_plus = P1_plus;

    end

end
```

```
function [yout_truth, yout_kalman, zout] = sim_kalman_filter(
    savedflight_filepath)
%sim_kalman_filter - Simulate kalman filter
%
% Syntax: [yout_truth, yout_kalman, zout] = sim_kalman_filter(
    savedflight_filepath)

% Need to run simulation to populate measurements
addpath(genpath('./../Simulation/'));
addpath(genpath('./../Utilities/'));

% run sim without plots
if nargin == 0
    [yout_truth, zout] = initialize_sim(0);
elseif nargin == 1 % load input saved flight
    saved_flight_vars = load(savedflight_filepath);
    yout_truth = saved_flight_vars.yout;
```

```
    zout = saved_flight_vars.zout;
else
    error('sim_kalman_filter should only have 0 or 1 input');
end

% time settings
tSpan = [0, yout_truth.t(end)]'; % dt = 0.01 sec

% Get initial state
x_init = yout_truth.x(:, 1);
x_init(end-2:end) = [0, 0, 0]';

% Get initial covariance matrix
P_init = initial_covariance_matrix();
tic
[x_out, P_out, t_out] = run_kalman_filter(x_init, P_init, tSpan,
   zout);
toc
yout_kalman.x = x_out;
yout_kalman.P = P_out;
yout_kalman.t = t_out;

plots_kalmanfilter(yout_truth, yout_kalman);

end
```

```
function [y1] = simulate_measurement_vector(time, zout)
```

```
%simulate_measurement_vector - Simulate measurement vector based on sample times
%
% Syntax: [y1, counts] = simulate_measurement_vector(time, zout)
%
% if gps is available:
%   y1 = [y_gps, y_altimeter, y_lidar, y_imu, y_source]'
% if gps is not available, but altimeter is available:
%   y1 = [y_altimeter, y_lidar, y_imu, ysource]'
% if gps and altimeter are not available, lidar, imu, and source should always be
% available.
%   y1 = [y_lidar, y_imu, y_source]'

    % unpack inputs
    gps_measurements = zout.gps;
    altimeter_measurements = zout.altimeter;
    lidar_measurements = zout.lidar;
    imu = zout.imu;
    source = zout.source;

    % Fix time overflow. Rounding to 6 decimal places should be more than enough
    time = round(time, 6);

    % Populate sensor specs. No need for altimeter or gyro
```

```
[accelerometer_specs, ~, gps_specs, altimeter_specs, ~,
    source_specs] = get_sensor_specs();

% If gps would be available, find simulated measurement
if mod(time, round(1/gps_specs.sampleRate, 6)) == 0
    gps_index = find(gps_measurements.t == time);
    y_gps = [gps_measurements.position(:, gps_index);
        gps_measurements.velocity(:, gps_index)];
    % if altitude reading for gps is less than zero, disregard
        the measurement.
    if y_gps(3) < 0
        y_gps(3) = 0;
    end
else
    y_gps = [0; 0; 0; 0; 0];
end

% check for altimeter

if mod(time, round(1/altimeter_specs.sampleRate, 6)) == 0
    y_altimeter = altimeter_measurements.altitude(:,
        altimeter_measurements.t == time);
else
    y_altimeter = 0;
end

% remaining sensors should always be available
```

```
    % lidar
    y_lidar = lidar_measurements.distance(1, lidar_measurements.t == time);

    % disregard lidar measurement if it is greater than 20m or equal to zero.
    if y_lidar == 0 || y_lidar > 20
        y_lidar = 0;
    elseif y_lidar < 0 % lidar should never be negative
        error("lidar reading negative: %d", y_lidar);
    end

    % imu
    if mod(time, round(1/accelerometer_specs.sampleRate, 6)) == 0
        y_accelerations = imu.accelerometer(:, imu.t == time);
        y_accelerometer_ind = find(imu.t == time);
        if time > 1.01
            last_second_accel = imu.accelerometer(:, y_accelerometer_ind-100:y_accelerometer_ind);
            last_second_accel_roll_pitch = imu.accelerometer_roll_pitch(:, y_accelerometer_ind-100:y_accelerometer_ind);
            norm_last_second_accel = zeros(1, length(last_second_accel));
            for i = 1:length(norm_last_second_accel)
                norm_last_second_accel(i) = norm(last_second_accel(:, i));
```

```
            end
            if abs(9.8 - mean(norm_last_second_accel)) > 0.001
                y_accelerometer_roll_pitch = [0; 0];
            else
                y_accelerometer_roll_pitch = mean(
                   last_second_accel_roll_pitch, 2);
                if any(rad2deg(y_accelerometer_roll_pitch) > 5)
                    y_accelerometer_roll_pitch = [0;0];
                end

            end
        else
            y_accelerometer_roll_pitch = [0; 0];
        end
    else
        error('IMU should be available every timestep');
    end

    % source
    y_source = [0; 0; 0; 0]; % initialize y_source
    if mod(time, round(1/source_specs.sampleRate, 6)) == 0
        counts = source.counts(source.t == time);
        y_source(1) = counts;
    end
    % find the most recent source_position estimate if one exists
    % source_pos_estimate_ind = find(time >= source.
       position_estimate(1, :), 1, 'last');
```

```
    % if ~isempty(source_pos_estimate_ind)
    %       y_source(2:4) = source.position_estimate(2:4,
        source_pos_estimate_ind);
    %       if time > source.position_estimate(1,
        source_pos_estimate_ind)+.009
    %           y_source(2:4) = [0;0;0];
    %       end
    % end

    % create output
    y1 = [y_gps; y_altimeter; y_lidar; y_accelerations;
        y_accelerometer_roll_pitch; y_source];

    if length(y1) ~= 16
        error('y1 was not fully populated.');
    end
end
```

```
function hx = state_to_measurement(x1_minus, y1)
% state_to_measurement - convert state vector to measurement units
%
% Measurements are as follows:
%   GPS: [latitude, longitude, altitude, groundspeed, course] (deg
    , deg, m, m/s, deg)
%   Altimeter: [altitude] (m)
%   LIDAR: [distance] (m)
%   External sensor (Geiger counter): [counts, p2, p3, p4]
```

```
%        note p2 - p4 are unused

    % unpack state
    [pos, vel, accel, eulerangles, obstacle_pos, source_pos] =
        unpack_state_vector(x1_minus);

    % initialize output

    % Populate hx if y1 contains a corresponding measurement.

    % gps
    hx_gps = state_to_gps(pos, vel);
    hx_gps(y1(1:5) == 0) = 0;

    % altimeter
    if y1(6) ~= 0
        hx_altimeter = state_to_altimeter(pos);
    else
        hx_altimeter = 0;
    end

    % lidar
    if y1(7) ~= 0
        hx_lidar = state_to_lidar(eulerangles, obstacle_pos);
    else
        hx_lidar = 0;
    end
```

```

    % imu
    if y1(8) ~= 0
        hx_imu = state_to_imu(accel, eulerangles);
    else
        hx_imu = [0; 0; 0; 0; 0];
    end
    % angle reading from accelerometer may not be good
    if y1(11) == 0
        hx_imu(4:5) = [0; 0];
    end

    % source
    hx_source = state_to_source(source_pos);

    % output
    hx = [hx_gps; hx_altimeter; hx_lidar; hx_imu; hx_source];

    % throw error if y1 and hx do not have the same length.
    if length(y1) ~= length(hx)
        error('y1 and hx should have the same length')
    end

end

function hx_gps  = state_to_gps(pos_ned, vel_ned)
% convert state vector into GPS measurement units
```

```matlab

    % gps only measures groundspeed.
    vel_ne = vel_ned(1:2);
    hx_gps = [pos_ned; vel_ne];

end

function hx_altimeter = state_to_altimeter(pos_ned)
% convert state vector into altimeter measurement units

    hx_altimeter = -pos_ned(3); % meters

end

function hx_lidar = state_to_lidar(eulerangles, obstacle_pos_ned)
% convert state vector into lidar measurement units
%
% lidar is assumed to point in the +bx direction

    obstacle_pos_body = DCM_ned2body(obstacle_pos_ned, eulerangles
        );

    hx_lidar = obstacle_pos_body(1);

end

function hx_imu = state_to_imu(accel_ned, eulerangles)
```

```
% observables: [accel_body, roll, pitch]
    accel_body = DCM_ned2body(accel_ned + [0; 0; 9.8], eulerangles
        );
    roll = eulerangles(1);
    pitch = eulerangles(2);

    hx_imu = [accel_body; roll; pitch];

end

function hx_source = state_to_source(~)
% convert state vector into source units

    hx_source = [0, 0, 0, 0]';
end
```

```
function x1 = state_update(x0, dt, UAV, gyro)
% state_update - non-linear state update function.

%% Unpack state

% x0
[pos0, vel0, accel0, eulerangles0, obstacle_pos0, source_pos0] =
    unpack_state_vector(x0);

% UAV
g = 9.8; % m/s^2
```

```
m = UAV.mass;

% Thrust is assumed to be equal and opposite to the weight of the UAV. Note however that thrust always acts in the -body z direction
thrust_body = [0, 0, -m*g / (cos(eulerangles0(1)) * cos(eulerangles0(2)))]';
thrust_ned = DCM_body2ned(thrust_body, eulerangles0);

% alter gyro input
eulerrates0 = angVel_to_eulerRates(eulerangles0) * gyro;

%% State Update

pos1 = pos0 + vel0 .* dt + 1/2 .* accel0 .* dt^2;
vel1 = vel0 + accel0 .* dt; % velocity in z is assumed to be almost always constant
accel1 = [thrust_ned(1)/m; thrust_ned(2)/m; accel0(3)];
eulerangles1 = eulerangles0 + eulerrates0 * dt;
obstacle_pos1 = obstacle_pos0 - vel0 .* dt;

source_pos1 = source_pos0;

% updated state
x1 = [pos1; vel1; accel1; eulerangles1; obstacle_pos1; source_pos1];
end
```

```matlab
function [x1_minus, P1_minus] = time_update(x0_plus, P0_plus,
    gyro_meas, dt, UAV)
% time_update - Performs time_update step for the Kalman filter
%
% Usage: [x1_minus, P1_minus] = time_update(x0_plus, P0_plus, Q,
    UAV, imu_meas)
%
% INPUTS
%   x0_plus: [18 x 1] + state vector for k-1th state
%   P0_plus: [18 x 18] + covariance matrix for -1th state
%   imu_meas: [6 x 1] IMU measurements for k-1th state (accel,
    gyro)
%   Q: [18 x 18] process noise matrix
%   UAV: [struct] structure containing UAV parameters
%
% OUTPUTS
%   x1_minus: [18 x 1] state vector for kth state without
    measurement update
%   P1_minus: [18 x 18] covariance matrix for kth state without
    measurement update

% Populate process noise covariance matrix
Q = process_noise_matrix(dt);

% compute partial derivative matrix
F0 = linearized_state_update_matrix(x0_plus, dt);
```

```
% perform time update
x1_minus = state_update(x0_plus, dt, UAV, gyro_meas);
P1_minus = F0 * P0_plus * F0' + Q;


end
```

APPENDIX D

UTILITIES

```matlab
function angVel_to_eulerRates_matrix = angVel_to_eulerRates(eulerAngles)
% output 3x3 transformation matrix converting angular velocity to euler
% rates
%
% Syntax: angVel_to_eulerRates_matrix = angVel_to_eulerRates(eulerAngles)
%
% INPUTS
%   eulerAngles: [3 x 1] containing euler angles in radians
% OUTPUTS
%   angVel_to_eulerRates_matrix: [3x3] containing angular velocity to euler
%       rates transformation

phi = eulerAngles(1);
theta = eulerAngles(2);

angVel_to_eulerRates_matrix = [...
    1, sin(phi)*tan(theta), cos(phi)*tan(theta); ...
    0, cos(phi), -sin(phi); ...
    0, sin(phi)*sec(theta), cos(phi)*sec(theta)];

```

```
end
```

```
function [vNED, iTb] = DCM_body2ned(vBody, eulerAngles)
% DCM_body2ned - Transform vector in NED frame to body frame
%
% Syntax: [vNED, iTb] = DCM_body2ned(vBody, eulerAngles)
%
% INPUTS
%   vBody:            [3 x n] vectors in NED frame
%   eulerAngles:      [3 x n] vectors of euler angles [roll, pitch,
    yaw]' (rad)
% OUTPUTS
%   vNED:             [3 x n] column vectors in NED frame
%   iTb:              [3 x 3 x n] body to inertial transformation

% initialize outputs
numVect = size(vBody, 2);
vNED = zeros(3, numVect);   % vector in NED
iTb = zeros(3, 3, numVect); % body to inertial DCM

% unpack Euler Angles
phi = eulerAngles(1, :);   % roll
theta = eulerAngles(2, :); % pitch
psi = eulerAngles(3, :);   % yaw

% 3-2-1 DCM giving iTb
```

```
Tphi = @(roll) [1, 0, 0; 0, cos(roll), sin(roll); 0, -sin(roll), cos(roll)];
Ttheta = @(pitch) [cos(pitch), 0, -sin(pitch); 0, 1, 0; sin(pitch), 0, cos(pitch)];
Tpsi = @(yaw) [cos(yaw), sin(yaw), 0; -sin(yaw), cos(yaw), 0; 0, 0, 1];

for i = 1:numVect
    bTi = Tphi(phi(i)) * Ttheta(theta(i)) * Tpsi(psi(i));
    iTb(:, :, i) = bTi';
    vNED(:, i) = iTb(:, :, i) * vBody(:, i);
end


end
```

```
function [vWind, wTb] = DCM_body2wind(vBody, aoa, ssa)
% DCM_ned2body - Transform vector in NED frame to body frame
%
% Syntax: [vWind, wTb] = DCM_body2wind(vBody, aoa, ssa)
%
% INPUTS
%   vBody:  [3 x n] column vectors in body frame
%   aoa:    [1 x n] vector of angle of attack (rad)
%   ssa:    [1 x n] vector of side slip angle (rad)
% OUTPUTS
%   vWind:            [3 x n] column vectors in wind frame
```

```
%   wTb:             [3 x 3 x n] body to wind transformation

% initialize outputs
numVect = size(vBody, 2);
vWind = zeros(3, numVect);
wTb = zeros(3, 3, numVect);     % initialize inertial to body
    transformation matrix

% Transforms
Taoa = @(alpha) [cos(alpha), 0, -sin(alpha); 0, 1, 0; sin(alpha),
    0, cos(alpha)];
Tssa = @(beta) [cos(beta), sin(beta), 0; -sin(beta), cos(beta), 0;
     0, 0, 1];

for i = 1:numVect
    wTb(:, :, i) = Tssa(ssa(i)) * Taoa(aoa(i));
    vWind(:, i) = wTb(:, :, i) * vBody(:, i);
end


end
```

```
function [vBody, bTi] = DCM_ned2body(vNED, eulerAngles)
% DCM_ned2body - Transform vector in NED frame to body frame
%
% Syntax: [vBody, bTi] = DCM_ned2body(vNED, eulerAngles)
%
```

```
% INPUTS
%   vNED:           [3 x n] column vectors in NED frame
%   eulerAngles:    [3 x n] vectors of euler angles [roll, pitch, yaw]' (rad)
% OUTPUTS
%   vBody:          [3 x n] column vectors in body frame
%   bTi:            [3 x 3 x n] inertial to body transformation

% initialize outputs
numVect = size(vNED, 2);
vBody = zeros(3, numVect);
bTi = zeros(3, 3, numVect);     % initialize inertial to body transormation matrix

% unpack Euler Angles
phi = eulerAngles(1, :);   % roll
theta = eulerAngles(2, :); % pitch
psi = eulerAngles(3, :);   % yaw

% 3-2-1 DCM
Tphi = @(roll) [1, 0, 0; 0, cos(roll), sin(roll); 0, -sin(roll), cos(roll)];
Ttheta = @(pitch) [cos(pitch), 0, -sin(pitch); 0, 1, 0; sin(pitch), 0, cos(pitch)];
Tpsi = @(yaw) [cos(yaw), sin(yaw), 0; -sin(yaw), cos(yaw), 0; 0, 0, 1];

```

```
for i = 1:numVect
    bTi(:, :, i) = Tphi(phi(i)) * Ttheta(theta(i)) * Tpsi(psi(i));
    vBody(:, i) = bTi(:, :, i) * vNED(:, i);
end


end
```

```
function [vWind, wTi] = DCM_ned2wind(vNED, eulerAngles, aoa, ssa)
% DCM_ned2body - Transform vector in NED frame to body frame
%
% Syntax: [vWind, wTi] = DCM_ned2wind(vNED, eulerAngles, aoa, ssa)
%
% INPUTS
%   vNED:            [3 x n] column vectors in NED frame
%   eulerAngles:     [3 x n] vectors of euler angles [roll, pitch,
    yaw]' (rad)
%   aoa:             [1 x n] vector of angle of attack (rad)
%   ssa:             [1 x n] vector of side slip angle (rad)
% OUTPUTS
%   vWind:           [3 x n] column vectors in wind frame
%   wTi:             [3 x 3 x n] inertial to body transformation

% initialize outputs
numVect = size(vNED, 2);
vWind = zeros(3, numVect);
```

```
wTi = zeros(3, 3, numVect);     % initialize inertial to body
    transormation matrix

% unpack Euler Angles
phi = eulerAngles(1, :);   % roll
theta = eulerAngles(2, :); % pitch
psi = eulerAngles(3, :);   % yaw

% 3-2-1 NED to body DCM
Tphi = @(roll) [1, 0, 0; 0, cos(roll), sin(roll); 0, -sin(roll),
    cos(roll)];
Ttheta = @(pitch) [cos(pitch), 0, -sin(pitch); 0, 1, 0; sin(pitch)
    , 0, cos(pitch)];
Tpsi = @(yaw) [cos(yaw), sin(yaw), 0; -sin(yaw), cos(yaw), 0; 0,
    0, 1];

% Body to wind rotations
Taoa = @(alpha) [cos(alpha), 0, -sin(alpha); 0, 1, 0; sin(alpha),
    0, cos(alpha)];
Tssa = @(beta) [cos(beta), sin(beta), 0; -sin(beta), cos(beta), 0;
     0, 0, 1];

for i = 1:numVect
    bTi = Tphi(phi(i)) * Ttheta(theta(i)) * Tpsi(psi(i));
    wTb = Tssa(ssa(i)) * Taoa(aoa(i));

    wTi(:, :, i) = wTb * bTi;
```

```
    vWind(:, i) = wTi(:, :, i) * vNED(:, i);
end


end
```

```
function [vBody, bTw] = DCM_wind2body(vWind, aoa, ssa)
% DCM_ned2body - Transform vector in NED frame to body frame
%
% Syntax: [vBody, bTw] = DCM_wind2body(vWind, aoa, ssa)
%
% INPUTS
%   vWind:  [3 x n] column vectors in wind frame
%   aoa:    [1 x n] vector of angle of attack (rad)
%   ssa:    [1 x n] vector of side slip angle (rad)
% OUTPUTS
%   vBody:          [3 x n] column vectors in body frame
%   bTw:            [3 x 3 x n] wind to body transformation

% initialize outputs
numVect = size(vWind, 2);
vBody = zeros(3, numVect);
bTw = zeros(3, 3, numVect);     % initialize inertial to body
    transormation matrix

% Transforms
```

```
Taoa = @(alpha) [cos(alpha), 0, -sin(alpha); 0, 1, 0; sin(alpha),
    0, cos(alpha)];
Tssa = @(beta) [cos(beta), sin(beta), 0; -sin(beta), cos(beta), 0;
     0, 0, 1];

for i = 1:numVect
    wTb = Tssa(ssa(i)) * Taoa(aoa(i));
    bTw(:, :, i) = wTb';
    vBody(:, i) = bTw(:, :, i) * vWind(:, i);
end


end
```

```
function [vNED, iTw] = DCM_wind2ned(vWind, eulerAngles, aoa, ssa)
% DCM_ned2body - Transform vector in NED frame to body frame
%
% Syntax: [vNED, iTw] = DCM_wind2ned(vWind, eulerAngles, aoa, ssa)
%
% INPUTS
%    vwind:            [3 x n] column vectors in wind frame
%    eulerAngles:      [3 x n] vectors of euler angles [roll, pitch,
    yaw]' (rad)
%    aoa:              [1 x n] vector of angle of attack (rad)
%    ssa:              [1 x n] vector of side slip angle (rad)
% OUTPUTS
%    vNED:            [3 x n] column vectors in NED frame
```

```
%   wTi:              [3 x 3 x n] inertial to wind transformation

% initialize outputs
numVect = size(vWind, 2);
vNED = zeros(3, numVect);
iTw = zeros(3, 3, numVect);      % initialize inertial to body
    transormation matrix

% unpack Euler Angles
phi = eulerAngles(1, :);    % roll
theta = eulerAngles(2, :); % pitch
psi = eulerAngles(3, :);    % yaw

% 3-2-1 NED to body DCM
Tphi = @(roll) [1, 0, 0; 0, cos(roll), sin(roll); 0, -sin(roll),
    cos(roll)];
Ttheta = @(pitch) [cos(pitch), 0, -sin(pitch); 0, 1, 0; sin(pitch)
    , 0, cos(pitch)];
Tpsi = @(yaw) [cos(yaw), sin(yaw), 0; -sin(yaw), cos(yaw), 0; 0,
    0, 1];

% Body to wind rotations
Taoa = @(alpha) [cos(alpha), 0, -sin(alpha); 0, 1, 0; sin(alpha),
    0, cos(alpha)];
Tssa = @(beta) [cos(beta), sin(beta), 0; -sin(beta), cos(beta), 0;
     0, 0, 1];

```

```
for i = 1:numVect
    bTi = Tphi(phi(i)) * Ttheta(theta(i)) * Tpsi(psi(i));
    wTb = Tssa(ssa(i)) * Taoa(aoa(i));
    wTi = wTb * bTi;
    iTw(:, :, i) = wTi';
    vNED(:, i) = iTw(:, :, i) * vWind(:, i);
end


end
```

```
function [aoa,ssa] = get_aoa_ssa(vel_uav_body)
% get_aoa_ssa - compute angle of attack and side slip angle from velocity
% in body frame
%
% Syntax: [aoa, ssa] = get_aoa_ssa(vel_uav_body)
%
% INPUTS:
%   vel_uav_body: [3 x n] velocity vector of UAV in body frame (m/s)
% OUTPUTS:
%   aoa:          angle of attack (rad)
%   ssa:          side slip angle (rad)

u = vel_uav_body(1, :); % body x-velocity
v = vel_uav_body(2, :); % body y-velocity
```

```
w = vel_uav_body(3, :); % body z-velocity

aoa = atan2(w, u); % angle of attack (rad)
ssa = real(asin(v / norm([u, w]))); % side slip angle (rad)
if isnan(ssa)
    ssa = 0;
end

if isnan(ssa)
    ssa = 0;
end

end
```

```
function eulerrates_to_angvel_matrix = get_eulerrates_to_angvel_matrix(eulerangles)
%get_eulerrates_to_angvel_matrix - Output matrix to convert eulerrates to
%angular velocities
%
% Syntax: eulerrates_to_angvel_matrix = get_eulerrates_to_angvel_matrix(eulerangles)

    phi = eulerangles(1);
    theta = eulerangles(2);

    eulerrates_to_angvel_matrix = [...
```

```
    1, 0, -sin(theta);
    0, cos(phi), sin(phi)*cos(theta);
    0, -sin(phi), cos(phi)*cos(theta)];

end
```

```
function [accelerometer, gyro, gps, altimeter, lidar, source] =
   get_sensor_specs()
%get_sensor_specs - outputs specs as a struct for each sensor
%
% Syntax: [accelerometer, gyro, gps, altimeter, lidar, source] =
   get_sensor_specs()

    g = 9.8; % acceleration due to gravity

    % Accelerometer
    accelerometer.initBias = 0; % initial accelerometer bias (m/s
       ^2) [1e-3 * 80*g]
    accelerometer.biasStability = 1e-5; % bias stability (drifts
       by this much per second) (m/s^2)
    accelerometer.sampleRate = 100; % Hz
    accelerometer.noiseDensity = 1e-6 * (150 * g); % accel output
       noise density (m/s^2) * Hz^(-1/2)
    accelerometer.sigma = (accelerometer.noiseDensity * sqrt(
       accelerometer.sampleRate)); % RMS of accelerometer noise (m
       /s^2)

```

```
% Gyroscope
gyro.initBias = deg2rad(0); % (rad/s) [1]
gyro.biasStability = deg2rad(.0083); % bias stability - 30 deg
    /hr (rad/hr)
gyro.noiseDensity = deg2rad(0.014); % noise density (rad/s) *
    Hz^(-1/2)
gyro.sampleRate = accelerometer.sampleRate;
gyro.sigma = gyro.noiseDensity * sqrt(gyro.sampleRate);
gyro.eulerDrift = deg2rad(20) / 3600; % 20 degrees per hour (
    rad/s)

% GPS
gps.sigmaPos = 2.5/3; % position standard deviation (m)
gps.sigmaVel = 0.1/3; % velocity standard deviation (m)
gps.sampleRate = 1; % (Hz)

% altimeter
altimeter.sigma = 0.11; % (m)
altimeter.sampleRate = 50; % (Hz)

% lidar
lidar.sigma = 0.1; % (m)
% keep sample rate same as accelerometer. Typical is actually
    270 Hz
lidar.sampleRate = accelerometer.sampleRate; % (Hz)

% source
```

```
    source.sigma = 0;
    source.sampleRate = 10; % frequency at which to take a
        measurement
end
```

```
function UAV = initialize_uav()
% uav_initialization - initialize struct containing UAV parameters
%
% OUTPUTS:
%   UAV - Structure containing physical and aerodynamic paramaters
%       unique
%    to the UAV.
%
% NOTE: propellers are numbered as follows:
%    [1 2]
%    [4 3]

% mass and moments of inertia
mass = 0.6; % mass of UAV (kg)

% prop dimensions (meters)
propLength = 0.1;
propWidth = 0.02;
propArea = propLength*propWidth;

% principal moments (kg*m^2) may want to change this to moment of
    inertia
```

```
% of two thin rods crossed
Ixx = 0.02;
Iyy = 0.02;
Izz = 0.05;
% off-diagonal elements (kg*m^2)
Ixy = 0;
Ixz = 0;
Iyx = 0;
Iyz = 0;
Izx = 0;
Izy = 0;

inertiaMatrix = [Ixx, Ixy, Ixz; Iyx, Iyy, Iyz; Izx, Izy, Izz]; %
    Inertia matrix for aircraft in body frame (kg*m^2)

% body frame moment arms (meters)
rProp1 = propLength*sind(45)*[1; -1; 0];
rProp2 = propLength*sind(45)*[1; 1; 0];
rProp3 = propLength*sind(45)*[-1; 1; 0];
rProp4 = propLength*sind(45)*[-1; -1; 0];
rProp = [rProp1, rProp2, rProp3, rProp4];

% aerodynamic coefficients
CD = 0.1; % drag coefficient
CT = 0.7; % thrust coefficient

% maximum allowable bank and pitch for UAV
```

```
maxRotationUAV = deg2rad(5);  % radians

UAV = struct('mass', mass, 'inertiaMatrix', inertiaMatrix, ...
    'propArea', propArea, 'propVects', rProp, 'CT', CT, 'CD', CD, ...
    'maxRotation', maxRotationUAV);
```

```
function [d_iTb_d_phi, d_iTb_d_theta, d_iTb_d_psi] = jacobian_DCM_body2ned(euler_angles)
%jacobian_DCM_body2ned - Outputs the Jacobian of the body to inertial DCM
%
% Syntax: [d_iTb_d_phi, d_iTb_d_theta, d_iTb_d_psi] = jacobian_DCM_body2ned(euler_angles)

    % euler angles
    phi = euler_angles(1);
    theta = euler_angles(2);
    psi = euler_angles(3);

    % jacobian terms
    d_iTb_d_phi = [...
        0, cos(psi)*sin(theta)*cos(phi) + sin(phi)*sin(psi), cos(phi)*sin(psi) - sin(phi)*cos(psi)*sin(theta);...
        0, -sin(phi)*cos(psi) + sin(theta)*cos(phi)*sin(psi), -(sin(phi)*sin(theta)*sin(psi) + cos(psi)*cos(phi));...
        0, cos(theta)*cos(phi), -cos(theta)*sin(phi);...
```

```
        ];
    d_iTb_d_theta = [...
        -sin(theta)*cos(psi), cos(psi)*cos(theta)*sin(phi), cos(
            phi)*cos(psi)*cos(theta);...
        -sin(theta)*sin(psi), cos(theta)*sin(phi)*sin(psi), cos(
            phi)*cos(theta)*sin(psi);...
        -cos(theta), -sin(theta)*sin(phi), -sin(theta)*cos(phi)
            ;...
        ];
    d_iTb_d_psi = [...
        -cos(theta)*sin(psi), -(sin(psi)*sin(theta)*sin(phi) + cos
            (phi)*cos(psi)), sin(phi)*cos(psi) - cos(phi)*sin(psi)*
            sin(theta);...
        cos(theta)*cos(psi), -cos(phi)*sin(psi) + sin(theta)*sin(
            phi)*cos(psi), cos(phi)*sin(theta)*cos(psi) + sin(psi)*
            sin(phi);...
        0, 0, 0];

end
```

```
function [d_bTi_d_phi, d_bTi_d_theta, d_bTi_d_psi] =
    jacobian_DCM_ned2body(euler_angles)
%jacobian_DCM_ned2body - Outputs the Jacobian of the inertial to
    body DCM
%
% Syntax: [d_bTi_d_phi, d_bTi_d_theta, d_bTi_d_psi] =
    jacobian_DCM_ned2body(euler_angles)
```

```

    % jacobian terms
    [d_iTb_d_phi, d_iTb_d_theta, d_iTb_d_psi] =
        jacobian_DCM_body2ned(euler_angles);

    % outputs are the tranpose of the iTb terms
    d_bTi_d_phi = d_iTb_d_phi';
    d_bTi_d_theta = d_iTb_d_theta';
    d_bTi_d_psi = d_iTb_d_psi';

end
```

```
function [d_angvel_T_eulerrates_dphi, d_angvel_T_eulerrates_dtheta
    , d_angvel_T_eulerrates_dpsi] =
    jacobian_eulerrates_to_angvel_matrix(eulerangles)
%jacobian_eulerrates_to_angvel_matrix - Jacobian of Euler rates to
     angular
%velocity matrix
%
% Syntax: [d_angvel_T_eulerrates_dphi,
    d_angvel_T_eulerrates_dtheta, d_angvel_T_eulerrates_dpsi] =
    jacobian_eulerrates_to_angvel_matrix(eulerangles)
%
% Long description

% unpack eulerangles
phi = eulerangles(1);
```

```
theta = eulerangles(2);

% partial derivative of eulerrates matrix with respect to phi
d_angvel_T_eulerrates_dphi = [...
    0, 0, 0;
    0, -sin(phi), cos(phi)*cos(theta);
    0, -cos(phi), -sin(phi)*cos(theta)];

% partial derivative of eulerrates matrix with respect to theta
d_angvel_T_eulerrates_dtheta = [...
    0, 0, -cos(theta);
    0, 0, -sin(phi)*sin(theta);
    0, 0, -cos(phi)*sin(theta)];

% partial derivative of eulerrates matrix with respect to psi
d_angvel_T_eulerrates_dpsi = zeros(3);

end
```

```
function posNED = lla_to_ned(originLLA, posLLA)
% lla_to_ned - converts geodetic coordings (latitude, longitude,
    altitude)
% to local NED frame (assuming origin is [0, 0, 0]).
%
% Syntax: posNED = lla_to_ned(originLLA, posLLA)
%
% INPUTS
```

```matlab
%   originLLA: [1 x N] Reference coordinates and altitude. Where local NED
%        frame is at its origin
%   posLLA: [3 x N] position in geodetic coordinates [lat; long; altitude]
%
% OUTPUTS
%   posNED: [3 x N] position in NED coordinates

 if size(posLLA, 1) ~= 3
     error('All inputs must have dimensions [3 x N]. posLLA has dimensions %d', posLLA);
end

rEarth = 1e3 * 6371; % radius of Earth (m)

northPos = (posLLA(1, :) - originLLA(1)) .* pi/180 .* rEarth;
eastPos = (posLLA(2, :) - originLLA(2)) .* pi/180 .* rEarth;
downPos = -posLLA(3, :);

posNED = [northPos; eastPos; downPos];
end
```

```matlab
function v = random_vector(magnitude)
    %random_vector - outputs a random vector with a constant magnitude

```

```
    v = randn(3, 1);
    v_normalized = v ./ norm(v);
    v = magnitude .* v_normalized;
end
```

```
function [all_final_source_distance_meters,
    all_first_iteration_percent_dist_covered,
    all_num_iterations_within_1m, convergence_array,
    all_first_iteration_time] = score_saved_runs(path_to_runs)
%score_saved_runs - score runs
%
% Syntax: [all_final_source_distance_meters, [
    all_first_iteration_percent_dist_covered,
    all_num_iterations_within_1m] = score_saved_runs(path_to_runs)
%

    if nargin < 1
        path_to_runs = '.';
    end
    filenames = dir(fullfile(path_to_runs, '*.mat'));
    filenames = {filenames.name};

    all_final_source_distance_meters = zeros(1, length(filenames))
        ;
    all_first_iteration_percent_dist_covered = zeros(1, length(
        filenames));
```

```
    all_first_iteration_dist_covered = zeros(1, length(filenames));
    all_num_iterations_within_1m = zeros(1, length(filenames));
    all_first_iteration_time = zeros(1, length(filenames));

    for i = 1:length(filenames)
        run_name = filenames{i};
        run_output = load(fullfile(path_to_runs, run_name));
        [first_estimate_distance_covered, first_iteration_percent_dist_covered, final_source_distance, num_iterations_within_1m, init_distance, first_iteration_time] = score_source_pos_estimation(run_output.yout);
        all_first_iteration_percent_dist_covered(i) = first_iteration_percent_dist_covered;
        all_final_source_distance_meters(i) = final_source_distance;
        all_num_iterations_within_1m(i) = num_iterations_within_1m;
        all_first_iteration_dist_covered(i) = first_estimate_distance_covered;
        all_first_iteration_time(i) = first_iteration_time;
    end

    % scoring plots

%     % final source distance
```

```
%     figure(1)
%     edges = [0:50:ceil(max(all_final_source_distance_meters).*100)+50];
%     histogram(all_final_source_distance_meters.*100, edges, 'FaceAlpha', 1);
%     xlabel('Source Distance (cm)')
%     ylabel('Runs');
%     %xticks(-1000:10:ceil(max(all_final_source_distance_meters).*100 + 1000));
%     title('Final source distance')
%
%     % first iteration percent distance
%     figure(2)
%     minBin = floor(min(all_first_iteration_percent_dist_covered)) - rem(floor(min(all_first_iteration_percent_dist_covered)), 10) - 10;
%     maxBin = floor(max(all_first_iteration_percent_dist_covered)) - rem(floor(max(all_first_iteration_percent_dist_covered)), 10) + 10;
%     edges = [minBin:5:maxBin];
%     histogram(all_first_iteration_percent_dist_covered, edges, 'FaceAlpha', 1)
%     xlabel('% closer');
%     ylabel('Runs')
%     title('First source distance reduction')
%     xticks([minBin:10:maxBin]);
%
```

```
%     % number of iterations until within 1m
%     figure(3)
    unique_iteration_counts = unique(all_num_iterations_within_1m)
        ;
    total_iteration_counts = zeros(1, length(
        unique_iteration_counts));
%     for i = 1:length(total_iteration_counts)
%         iterationNum = unique_iteration_counts(i);
%         total_iteration_counts(i) = sum(
   all_num_iterations_within_1m == iterationNum);
%     end
%     bar(unique_iteration_counts, total_iteration_counts);
%     xlabel('Number of iterations until <= 1m');
%     ylabel('Number of runs')
%     title('Iterations for distance <= 1m')

    % converges, will converge, does not converge within 1m
    num_runConverges = length(all_final_source_distance_meters(
        all_final_source_distance_meters < 1.0));
    num_willConverge = length(all_final_source_distance_meters(
        all_final_source_distance_meters < 9 &
        all_final_source_distance_meters > 1.0));
    num_willNotConverge = 100 - num_runConverges -
        num_willConverge;

    convergence_array = [init_distance, num_runConverges,
        num_willConverge, num_willNotConverge];
```

```
    categories = categorical({'Converges', 'Will Converge', 'Does
        not converge'});

%     figure(4)
%     bar(categories, [num_runConverges, num_willConverge,
    num_willNotConverge]);
%     title('Convergence within Sim Time')
%     ylabel('Runs')

    % first iteration time
%     figure(5);
%     edges = 0:1:ceil(max(all_first_iteration_time));
%     histogram(all_first_iteration_time, edges, 'FaceAlpha', 1);
%     xlabel('Time (sec)')
%     ylabel('Runs');
%     title('Time for activity threshold to be exceeded.');

end


function [first_estimate_distance_covered,
    first_iteration_percent_dist_covered, final_source_distance,
    num_iterations_within_1m, init_distance, first_iteration_time]
    = score_source_pos_estimation(yout)

    x = yout.x;
```

```
    % unpack state
    [pos_ned, ~, ~, ~, ~, ~] = unpack_state_vector(x);

    uav_init_pos = pos_ned(:, 1);

    true_source_pos = yout.true_source_pos;
    source_pos_estimates = yout.source.position_estimate;
    estimation_start_times = yout.source.estimation_start_times;

    init_distance = norm(true_source_pos - uav_init_pos);

    % ~~~~~~~~~~get percent distance closed on first iteration
        ~~~~~~~~~~
    % distance from first estimated position
    first_estimate_distance_covered = init_distance - norm(
        true_source_pos - source_pos_estimates(2:4, 1));

    % percent distance closed on first iteration
    first_iteration_percent_dist_covered = ((
        first_estimate_distance_covered / init_distance)) * 100;

    %~~~~~~~~~~~~~~~ get distance from source at end of sim
        ~~~~~~~~~~~~~~~~~~
    final_source_distance = norm(true_source_pos -
        source_pos_estimates(2:4, end));

```

```
    %~~~~~~~~~~~~~~~ first iteration time ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    first_iteration_time = estimation_start_times(1);

    %~~~~~~~~~~~~~~~~ get number of iterations required to get within 1m of source ~~~~~~
    source_pos_error = zeros(3, size(source_pos_estimates, 2));
    source_dist_error = zeros(1, size(source_pos_estimates, 2));
    for i = 1:size(source_pos_estimates, 2)
        source_pos_error(:, i) = true_source_pos - source_pos_estimates(2:4, i);
        source_dist_error(i) = norm(source_pos_error(:, i));
    end
    num_iterations_within_1m = find(source_dist_error <= 1, 1);
    if isempty(num_iterations_within_1m)
        num_iterations_within_1m = NaN;
    end

end
```

```
function B = skew(A)
% Converts a vector into 3x3 skew-symmetric form.
%
% Syntax: B = skew(A)

if any(size(A) > 3)
    error('A must be a 3x1 or 1x3 vector');
```

```
end

if isnumeric(A)
    B = zeros(3,3);
end

B(1,2) = -A(3);
B(1,3) = A(2);
B(2,1) = A(3);
B(2,3) = -A(1);
B(3,1) = -A(2);
B(3,2) = A(1);

end
```

```
function counts_1sec = store_1sec_counts(zout, time)
%store_1sec_counts - Store last second of counts
%
% Syntax: counts_1sec_new = store_1sec_counts(counts_1sec_old,
    counts)
%

    [~, ~, ~, ~, ~, source_specs] = get_sensor_specs();
    counts_all = zout.source.counts;
    counts_time = zout.source.t;

```

```
    % if we would be between measurements, round time to nearest tenths place to
    % construct counts_1sec
    if (mod(time, round(1/source_specs.sampleRate, 6)) ~= 0)
        time_tenths_place = get_digits_after_decimal(time, 1);
        time = floor(time) + time_tenths_place/10;
    end
    time = round(time, 6);

    if (mod(time, round(1/source_specs.sampleRate, 6)) == 0) && (time >= round(1/source_specs.sampleRate, 6))
        % get one seconds worth of counts to determine if gradient detection
        % should start
        counts_ind = find(counts_time == time);
        if time >= 1.0    % if time > 1.0, should have enough measurements
            time_1sec_earlier = round(time - 0.9, 6);
            counts_ind_1_sec_earlier = find(counts_time == time_1sec_earlier);
        else % time - 1 sec would be negative.
            counts_ind_1_sec_earlier = 1;
        end
        counts_1sec = counts_all(counts_ind_1_sec_earlier : counts_ind);

```

```
        % Pad zeros to end of counts_1sec if length is less than
            10. This should
        % only happen when time <= 1.0 sec
        if time <= 1.0
            counts_1sec = [counts_1sec, zeros(1, source_specs.
                sampleRate - length(counts_1sec))];
        elseif length(counts_1sec) ~= 10
            error('Counts_1sec has length less than 10 after one
                sec: Length = %d, Time = %d',length(counts_1sec),
                time);
        end
    elseif time < round(1/source_specs.sampleRate, 6)
        counts_1sec = zeros(1, source_specs.sampleRate);
    else
        error('Unable to create 1 sec worth of counts. Time = %d',
            time);
    end

end
```

```
function [pos_ned, vel_ned, accel_ned, eulerangles, obstacle_ned,
    source_ned] = unpack_state_vector(x)
%unpack_state_vector - Outputs state vector elements
%
% Syntax: [pos_ned, vel_ned, accel_ned, eulerangles, obstacle_ned,
    source_ned] = unpack_state_vector(x)
%
```

```
% Output state vector elements for use in other functions.
%
% INPUT
%   x [18 x N] - state vector or array of state vectors.

    % ensure x is a column vector or an array of column vectors
    if size(x, 1) ~= 18
        error('State vector "x" must contain 18 elements and must
            be given as a column vector or an array of column
            vectors.');
    end

    pos_ned = x(1:3, :);
    vel_ned = x(4:6, :);
    accel_ned = x(7:9, :);
    eulerangles = x(10:12, :);
    obstacle_ned = x(13:15, :);
    source_ned = x(16:18, :);
end
```

```
function [time_reached, command_duration] =
    time_until_waypoint_reached(yout)
%time_until_waypoint_reached - determine time until waypoint is
    reached
%
% Syntax: waypoints_reached = time_until_waypoint_reached(yout)
%
```

```
% Long description
    x = yout.x;
    pos = x(1:3, :);
    vel = x(4:6, :);
    t = yout.t;
    waypoints = yout.controls.waypoints;

    % threshold
    dist_thresh = 0.1;
    vel_mag_thresh = 0.05;

    time_reached = zeros(1, size(waypoints, 2));
    command_duration = zeros(1, size(waypoints, 2));
    for i = 1:length(waypoints)
        target_pos = waypoints(2:4, i);
        target_vel = [0, 0, 0]';
        pos_error = target_pos - pos;
        vel_error = target_vel - vel;
        dist_error = zeros(1, length(pos_error));
        vel_mag_error = zeros(1, length(vel_error));
        for j = 1:length(pos_error)
            dist_error(j) = norm(pos_error(:, j));
            vel_mag_error(j) = norm(vel_error(:, j));
        end
        ind = find(dist_error < dist_thresh & vel_mag_error <
           vel_mag_thresh, 1);
        time_reached(i) = t(ind);
```

```
        command_duration(i) = t(ind) - waypoints(1, i);
    end
end
```

```

A1 = @(r1) 1000/(r1^2);
law_cosines = @(a, b, theta_c) a^2+b^2 - 2*a*b*cos(theta_c);

dist_array = 1:1:50;
prob_wrongDir_best_all = zeros(1, length(dist_array));
prob_wrongDir_worst_all = zeros(1, length(dist_array));
prob_wrongDir_mid_all = zeros(1, length(dist_array));

count = 0;
prob_wrongDir_best = 0;
for dist = dist_array
    dist
    A_closeMean_best = A1(dist-1);
    A_farMean_best = A1(dist+1);

    dist_worst = sqrt(law_cosines(dist, 1, 90));
    A_closeMean_worst = A1(dist_worst);

    dist_mid_close = sqrt(law_cosines(dist, 1, 45));
    dist_mid_far = sqrt(law_cosines(dist, -1, 45));

    A_closeMean_mid = A1(dist_mid_close);
```

```
    A_farMean_mid = A1(dist_mid_far);

    prob_wrongDir_best = 0;
    prob_wrongDir_worst = 0;
    prob_wrongDir_mid = 0;

    for i = 0:500

        prob_Aclose_best = pdf('Poisson', i, A_closeMean_best);
        prob_Afar_greater_Aclose_best = 1-cdf('Poisson', i, A_farMean_best);
        prob_wrongDir_best = prob_wrongDir_best + prob_Aclose_best*(prob_Afar_greater_Aclose_best);

        prob_Aclose_worst = pdf('Poisson', i, A_closeMean_worst);
        prob_Afar_greater_Aclose_worst = 1-cdf('Poisson', i, A_closeMean_worst);
        prob_wrongDir_worst = prob_wrongDir_worst + prob_Aclose_worst*prob_Afar_greater_Aclose_worst;

        prob_Aclose_mid = pdf('Poisson', i, A_closeMean_mid);
        prob_Afar_greater_Aclose_mid = 1-cdf('Poisson', i, A_farMean_mid);
        prob_wrongDir_mid = prob_wrongDir_mid + prob_Aclose_mid*(prob_Afar_greater_Aclose_mid);

    end
```

```
        count = count+1;
        prob_wrongDir_best_all(count) = prob_wrongDir_best;
        prob_wrongDir_worst_all(count) = prob_wrongDir_worst;
        prob_wrongDir_mid_all(count) = prob_wrongDir_mid;

end

% plot(dist_array, prob_wrongDir_best_all, dist_array,
    prob_wrongDir_worst_all, dist_array, prob_wrongDir_mid_all);
% legend('\theta_0 = 0 \circ', '\theta_0 = 90 \circ', '\theta_0 =
    45 \circ');
% xlabel('Distance (m)');
% ylabel('Probability');
% title('Probability of Incorrect Source Direction');

plot(dist_array, prob_wrongDir_best_all, dist_array,
    prob_wrongDir_worst_all);
legend('\theta_0 = 0 \circ', '\theta_0 = 90 \circ');
xlabel('Distance (m)');
ylabel('Probability');
title('Probability of Incorrect Source Direction');
```

APPENDIX E

RADIOACTIVE SOURCE LOCALIZATION

```
function detection_flag = check_near_source_counts(counts, A_thresh, A_background)
%check_near_source_video - Check if near a source to trigger gradient test
%
% Syntax: detection_flag = check_near_source_video(counts, A_thresh)
%
% INPUTS
%   counts [N x 1, 1 x N]:  array of counts
%   A_thresh:   value that must be exceeded by sum of all elements in
%                            array "counts"
% OUTPUTS
%   detection_flag: flag raised if sum(counts) exceeds threshold.

    % A_thresh is number of counts required to trigger a gradient
    % measurement
    if nargin < 2
        error('Activity threshold unset')
    end

    if sum(counts) - A_background >= A_thresh
        detection_flag = 1;
```

```
    else
        detection_flag = 0;
    end

end
```

```
function at_target_flag = check_uav_at_target(x, r_measurement_ned
    )
    % check if we are at the waypoint
    deadzone = 0.1;
    % unpack inputs
    r_uav_ned = [x(1), x(2), x(3)]';
    v_uav_body = [x(4), x(5), x(6)]';
    if ~any(abs(r_uav_ned - r_measurement_ned) > deadzone) && ~any
        (abs(v_uav_body - [0; 0; 0]) > deadzone)
        at_target_flag = 1;
    else
        at_target_flag = 0;
    end
end
```

```
function source_pos_ned = estimate_source_pos(counts_gradient,
    r_gradient_center_ned, measurement_positions_ned, duration, A0,
    A_background, r0, A_thresh, true_source_ned)
    %estimate_source_pos - Estimate source position in NED
    %
    % Syntax: source_pos_ned = estimate_source_pos(x,
        counts_gradient, r_gradient_center_ned)
```

```
    %
    % Long description

    % error checking
    if length(counts_gradient) ~= duration * 10
        error('counts_gradient should have a size of duration * 10
            ');
    end

    % A_thresh sets maximum distance
    dist_max = r0 * sqrt((A0 + sqrt(A0)) / abs(A_thresh));

    std_A0 = sqrt(A0);

    source_dist = [];
    vec_row_count = 0;

    for i = 1:2:6
        vec_row_count = vec_row_count + 1;
        counts_plus = counts_gradient(i, :);
        pos_plus_ned = measurement_positions_ned(:, i);
        counts_minus = counts_gradient(i + 1, :);
        pos_minus_ned = measurement_positions_ned(:, i + 1);

        delta_pos = pos_plus_ned - pos_minus_ned;
        unit_delta_pos = delta_pos ./ norm(delta_pos); % unit
            vector
```

```
        % get average counts per second
        avg_counts_plus = mean(counts_plus) * 10 - A_background;
        avg_counts_minus = mean(counts_minus) * 10 - A_background;

        delta_counts = avg_counts_plus - avg_counts_minus;

        % get source distance along axis
        source_dist_plus = (r0 * sqrt(A0 / avg_counts_plus));
        source_dist_minus = (r0 * sqrt(A0 / avg_counts_minus));
        source_dist_array = [source_dist_plus, source_dist_minus];
        source_dist_diff = abs(norm(source_dist_plus) - norm(source_dist_minus));

        source_dist_axis = sign(delta_counts) * source_dist_diff / 2 .* unit_delta_pos;

        avg_counts_array = [avg_counts_plus, avg_counts_minus];
        [max_counts, max_counts_ind] = max(avg_counts_array);

        if abs(delta_counts) > sqrt(min([avg_counts_plus, avg_counts_minus])) || (avg_counts_plus > A0 - std_A0 && avg_counts_minus > A0 - std_A0)

            if source_dist_diff > 0.9
                source_dist_axis = sign(delta_counts) * (source_dist_array(max_counts_ind)) .* unit_delta_pos;
```

```
        elseif source_dist_diff > 0.5
            source_dist_axis = sign(delta_counts) * abs(
               source_dist_array(max_counts_ind) - 1) .*
               unit_delta_pos;
        else
            source_dist_axis = sign(delta_counts) *
               source_dist_diff / 2 .* unit_delta_pos;
        end

    end

    % if one measurement is much greater than A0, just use
       that
    % position
    if max_counts + sqrt(A0) > A0
        % if source_dist_diff is less than 1, use
           source_dist_diff for
        % distance estimation
        if source_dist_diff >= 1.5

            if max_counts_ind == 1
                source_dist_axis = (pos_plus_ned -
                   r_gradient_center_ned) .* unit_delta_pos;
            else
                source_dist_axis = (pos_minus_ned -
                   r_gradient_center_ned) .* unit_delta_pos;
            end
```

```
            end

        end

        %source_dist_axis = r0 * sign(delta_counts) * sqrt(A0 ./
            abs(delta_counts)) .* unit_delta_pos;

        % check if our values are usable.
        % standard deviation of counts is given by sqrt(counts).
        % if delta_counts < max(standard_deviation), do not use
        delta_counts_thresh = max([sqrt(avg_counts_plus), sqrt(
            avg_counts_minus)]);

        % source_dist_plus - source_dist_minus should equal 2. Set
             thresholds
        source_dist_diff_error = 0.5;
        source_dist_diff_flag = (source_dist_diff > 1 -
            source_dist_diff_error) && (source_dist_diff < 1 +
            source_dist_diff_error + 1);
        source_dist_diff_flag = 1;

        if any(abs(source_dist_axis) > dist_max) ||~
            source_dist_diff_flag || any(isnan(source_dist_axis))
             source_dist_axis = [0, 0, 0]';
        end

```

```
            source_dist = [source_dist, source_dist_axis];

        end

        source_pos_ned = sum(source_dist, 2) + r_gradient_center_ned;
        source_pos_error = true_source_ned - source_pos_ned;
        % source is not underground
        if source_pos_ned(3) > 0
            source_pos_ned(3) = 0;
        end

end
```

```
function [measurement_positions_ned, gradient_center_ned] =
    get_gradient_waypoints(x)
%get_waypoint_gradient - output UAV waypoint from measurement
    number and state
%
% Syntax: measure_gradient_waypoint = get_waypoint_gradient(x, t,
    measurement_num)
%
% Long description

    region_radius = 1;

    r_uav_ned = [x(1), x(2), x(3)]';

```

```
    gradient_center_ned = r_uav_ned;
    measurement_positions_ned(:, 1) = gradient_center_ned + [
       region_radius; 0; 0];
    measurement_positions_ned(:, 2) = gradient_center_ned - [
       region_radius; 0; 0];
    measurement_positions_ned(:, 3) = gradient_center_ned + [0;
       region_radius; 0];
    measurement_positions_ned(:, 4) = gradient_center_ned - [0;
       region_radius; 0];
    measurement_positions_ned(:, 5) = gradient_center_ned + [0; 0;
        region_radius];
    measurement_positions_ned(:, 6) = gradient_center_ned - [0; 0;
        region_radius];

    if measurement_positions_ned(3, 5) > -0.1
        difference = gradient_center_ned(3) - (-0.1);
        measurement_positions_ned(3, 5) = -0.1;
        measurement_positions_ned(3, 6) = gradient_center_ned(3) +
            difference;
    end

end
```

```
function [measuring_gradient_flag, at_target_flag,
   gradient_center_ned, target_pos_ned] =
   radiation_source_gradient(...
```

```
        x, time, at_target_flag, gradient_center_ned, measurement_num, measurement_positions)
measuring_gradient_flag = 1;
    if at_target_flag == 0
        [measurement_positions, gradient_center_ned, at_target_flag, measurement_num] = get_waypoint_gradient(x, gradient_center_ned, measurement_num, measurement_positions);
    end
    target_pos_ned = measurement_positions(:, measurement_num);
    %UPDATE WAYPOINTS OUTSIDE OF FUNCTION
    % if t_stop_meas is empty and we are at our target, set t_stop_meas
    if isempty(t_stop_meas)
        if at_target_flag == 1
            t_stop_meas = time + 30;
            t_stop_meas = round(t_stop_meas, 6);
        end
    else
        % if we are at the target and t < t_stop_meas, record counts
        if time <= round(t_stop_meas, 6) && at_target_flag == 1
            gradient_counts = [gradient_counts, counts_dt];
        elseif time > t_stop_meas % if t > t_stop_meas, save our recorded counts and increase measurement_num
```

```
            t_stop_meas = [];
            gradient_store_counts(measurement_num, :) = gradient_counts;
            gradient_counts = [];
            measurement_num = measurement_num + 1;
            at_target_flag = 0;
        end
    end
    % if measurement_num == 7, we are done saving measurements. Estimate
    % source position from gradient_store_counts
    if measurement_num == 7
        measuring_gradient_flag = 0;
        source_pos_ned = estimate_source_pos(gradient_store_counts, gradient_center_ned, measurement_positions, meas_duration, A0, r0, A_thresh);
        source_pos_ned_store = [source_pos_ned_store, [time; source_pos_ned]];
        do_not_measure_deadtime = time + 30; % do perform another gradient measurement for 30sec
        measurement_positions = [];
        measurement_num = 0;
        at_target_flag = 0;
        t_stop_meas = [];
        target_pos_ned = source_pos_ned;
        waypoints = [waypoints, [time; target_pos_ned]];
```

```
        end
end
```

BIOGRAPHY OF THE AUTHOR

John George Goulet was born in Lewiston, Maine on April 21, 1994. He was raised there and graduated from Lewiston High School in 2012. He attended the University of Maine and graduated in 2017 with a Bachelor's degree in Engineering Physics. John George Goulet is a candidate for the Master of Engineering degree in Engineering Physics from the University of Maine in May 2020.