ASE 479W Laboratory 4 Report
Path Planning

Aaron Pandian

March 22, 2024

1 Introduction

Quadrotors become increasingly difficult to control if pushed to provide more complex dynamic functionality. Given a deep understanding of the employed physics, one can establish a high fidelity model to simulate this unmanned aerial vehicle (UAV). Implementing this model with feedback control allows for a scalable approach to achieving target motion. The paper *Direct Acceleration Feedback Control of Quadrotor Aerial Vehicles* states this method "alleviates the need for accurate estimation of platform parameters." One key aspect of this system is the path planning algorithm used to create a reference trajectory, given an environmental map.

For this laboratory assignment, a high fidelity quadrotor simulator was designed from the theory of Newtonian dynamics and applied in Matlab. To ease model control, a feedback system was designed. Furthermore, three sensors were modeled: an inertial measurement unit (IMU), a global navigation satellite system (GNSS) receiver, and a camera. These sensors output realistic noisy values, and the current state is estimated using these measurements. This state attempts to minimize the error between itself and the reference trajectory using the developed feedback system. This reference trajectory is created using a path planning algorithm which attempts to reach a stated goal from initialization.

The theory, application, and experiment results of the path planning method are discussed in the following report.

2 Theoretical Analysis

The simulation of a quadcopter relies on multiple underlying principles. This section poses as an introduction prior to an expanded analysis in the following sections.

2.1 Modifying 2D Waypoint Coordinate Path to Position, Velocity, and Acceleration

The primary path planning algorithm used in the simulation is called A*. This method utilizes the pathfinding method found in Dijkstra's algorithm with the addition of a heuristic. Using a heuristic in this setting means employing a problem solving shortcut, such as taking advantage of the fact that the location of the end goal is known. By integrating this knowledge into A*, an algorithm that commonly outperforms its predecessor is developed. These methods, along with their baseline- Depth First Search, are explained in the following section. However, this context is important.

A consideration with the path planning algorithms developed is the fact that the output trajectory varies from the necessary input into the feedback loop. The generated reference trajectory is defined by position, velocity, and acceleration time histories in the inertial frame. However, the path planning algorithm provides

2D coordinates, representing the “environment” grid points, of the intended path to be taken. For this experiment, the goal was to create a trajectory along the X-Y plane, as a result, the Z axis was neglected in the path planning and set to zero for simulation. Regardless, the question on converting the list of 2D coordinates to position, velocity, and acceleration remains.

The conducted solution utilizes time history derivation. At the very least, the 2D coordinates generated by A* are convertible to 3D coordinates using the degree of freedom restriction mentioned above. Each coordinate is associated with an X-Y value pair. Using the coordinate path, 8th degree polynomials are fitted over the waypoints to create a continuous trajectory that adheres to the generated path. Using a sampling rate, the sample coordinates from the polynomial are provided a timestamp. Furthermore, the velocity, and acceleration values are initialized at time equals zero.

After the first two timestamps, the change of position with respect to time, or velocity, can be calculated for the *second* timestep. Similarly, using the initialized value of velocity and the calculated velocity, the rate of change of velocity with respect to time, or acceleration, can be calculated for the *second* timestep. With this approach, using the position, the values of velocity and acceleration in the X and Y directions can be determined. The values in the Z axis are negligible for the reference trajectory.

Nevertheless, this solution is not utilized in practice. In an ideal scenario, it is wiser to constrain velocity and acceleration in addition to position. By calculating these values for the reference trajectory based on the generated position, specific constraints on velocity and acceleration are withheld. Even if these values can still be minimized within the path planning algorithm, limitations or certain controls for, for example, specific regions of the environment, cannot be implemented. Moreover, this method inherently provides error to the velocity and acceleration values. A derivative is an estimation of the rate of change taken with anything larger than an infinitesimally small change in time. As a result, the sample frequency of generated position points drive the accuracy of the velocity and acceleration reference inputs. High sample frequencies can lead to computationally intensive programs and delays, thus this approach is not ideal to mitigate estimation errors. In reality, this is not the best approach to path planning, but, for simplicity, it is implemented for this experiment.

3 Implementation

To implement the high fidelity quadcopter simulation, a step-by-step approach was implemented. The code used is highlighted in brief demonstrations and discussed as to how functions relate to each other. This section works to explain the pieces of the simulation control algorithm for the quadrotor vehicle.

As mentioned prior, creating the base of the model is the euler rotation theorem. The equation is entirely defined except for the skew-symmetric cross product equivalent, for which the following function is generated.

```
Unset
function [uCross] = crossProductEquivalent(u)
```

```
u1 = u(1,1); % extracted values from row 1, column 1
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];

end
```

Figure 3: Cross product equivalent function

The application above outputs the cross-product-equivalent matrix *uCross* for an arbitrary 3-by-1 vector *u*. This function eases evaluating the cross product between two matrices, finding the vector that is perpendicular to both, which is critical when working with multiple directions and reference frames.

```
Unset
function [R] = rotationMatrix(aHat,phi)

I = [1 0 0; 0 1 0; 0 0 1];
aHatTranspose = aHat.';
R1 = cos(phi)*I;
R2 = (1-cos(phi))*aHat*aHatTranspose;
R3 = sin(phi)*crossProductEquivalent(aHat);
R = R1+R2-R3;

end
```

Figure 4: Rotation matrix function

To complicate the previous function, a rotation matrix function was developed. This function generates a rotation matrix through an angle about a specified axis.

By utilizing the script in Fig. 4, a function defining a rotation sequence is defined. To explain, a quadrotor vehicle undergoes asymmetrical rotation to complete maneuvers, meaning it has control over its roll, pitch, and yaw. These allow the aircraft to rotate about the X, Y, and Z axis respectively. For this simulation, a 3-1-2 rotation sequence was enacted, for which the quadcopter rotates itself about three body frame axes in specific order to induce a change in attitude. Each rotation includes one axis and one angular shift about that axis, called Euler axes and Euler angles, and can be modeled using (1). Put together, an attitude matrix C can depict the 3-1-2 rotation, which is a rotation in the “ZXY” sequence.

$$C(\psi, \phi, \theta) = R_2(\boldsymbol{e}_2, \theta)R_1(\boldsymbol{e}_1, \phi)R_3(\boldsymbol{e}_3, \psi) \tag{13}$$

$$\boldsymbol{e}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{13.1}$$

The notation above depicts the euler axes **e** as X, Y, and Z in order from left to right in (13.1). As a result, it is obvious to state that the Euler angles ψ, ϕ, and θ are responsible for the yaw, roll, and pitch in order. Note that the 3-1-2 sequence is written in rotation matrices from right to left.

```
Unset
function [R_BW] = euler2dcm(e)

phi = e(1,1);
theta = e(2,1);
psi = e(3,1);

R1 = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R2 = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R3 = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];

R_BW = R2e*R1e*R3e;

end
```

Figure 5: Euler angles to rotation matrix

The formula in Fig. 5 converts matrix *e* containing Euler angles ψ, ϕ, θ (in radians) into a directed cosine matrix for a 3-1-2 rotation. Assuming the inertial or world and body frame are initially aligned, *R_BW* can then be used to cast a vector expressed in inertial frame coordinates as a vector in the body frame. An alternative function denoting the inverse transformation was also developed.

```
Unset
function [e] = dcm2euler(R_BW)

% Euler angles in radians:
% phi = e(1), theta = e(2), and psi = e(3).  By convention, these
% should be constrained to the following ranges: -pi/2 <= phi <=
% pi/2, -pi <= theta < pi, -pi <= psi < pi.

phi = asin(R_BW(2,3));
assert(sin(phi)~=-pi/2 && sin(phi)~=pi/2,'Conversion is singular.');
theta = atan2(-(R_BW(1,3)),R_BW(3,3));
if theta == pi
   theta = -pi;
end
```

```
psi = atan2(-(R_BW(2,1)),R_BW(2,2));
if psi == pi
   psi = -pi;
end
e = [phi; theta; psi]';
end
```

Figure 6: Rotation matrix to euler angles

The function in Fig. 6 does the inverse operation from Fig. 5, whereby a rotation matrix is transformed to a matrix of euler angles, restricted by the limits above. The primary consideration is when the second rotation angle produces a singularity. In the case of a 3-1-2 sequence, when the roll angle $\phi = \pm \pi/2$, then the first and third rotations have equivalent effects, and thus, cannot be distinguished from one another. As a result, this function outputs an error instead of the euler angle matrix *in this situation*.

With these defined functions, the next step of implementation is to develop a model for the quadcopter dynamics. This function, given an initial state, will compute its derivative. This method will be iterated over using the Runge Kutta numerical ODE solver in Matlab. An oriented schematic of the quadcopter is shown in Fig. 7.

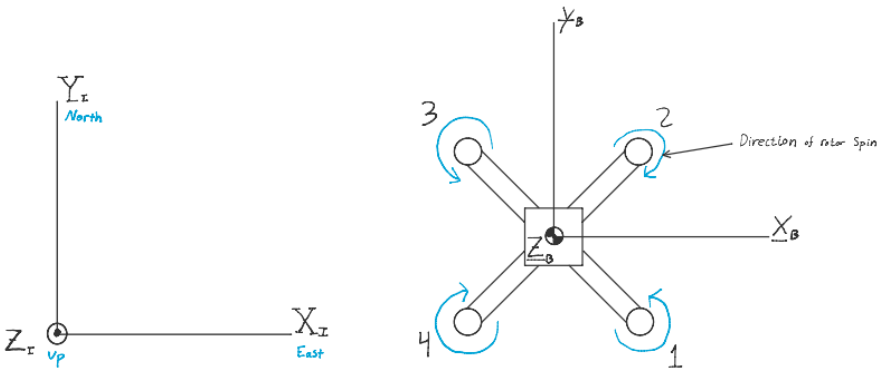


Figure 5: Quadcopter visualization

First it is important to understand the definition of state of the quadrotor, seen in (14).

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{r}_{\mathrm{I}} \\ \boldsymbol{e} \\ \boldsymbol{v}_{\mathrm{I}} \\ \boldsymbol{\omega}_{\mathrm{B}} \end{bmatrix} \tag{14}$$

In the above matrix, r_{I} represents the position vector of the quadcopter center of mass in the inertial frame and v_{I} is the velocity vector with respect to the inertial frame, in the inertial frame. The euler angle vector

is as defined in Fig. 4 and ω_B is the angular velocity vector in the body frame. Considering the 3-dimensional size of each input matrix, the state matrix is a 12x1 matrix with all variables listed out-instead of demarketing four input matrices within the state matrix. To find the derivative of the state $\dot{X}$, the following equations will need to be computed for each state input.

$$\dot{\boldsymbol{r}}_I = \boldsymbol{v}_I \tag{15}$$

$$m\dot{\boldsymbol{v}}_I = m\ddot{\boldsymbol{r}}_I = -mg\boldsymbol{z}_I + \sum_{i=1}^{4} \boldsymbol{F}_{iI} + \boldsymbol{d}_I = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{BI}^T \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ F_i \end{bmatrix} + \boldsymbol{d}_I \tag{16}$$

$$\dot{\boldsymbol{\omega}}_B = J^{-1}\left(\boldsymbol{N}_B - [\boldsymbol{\omega}_B\times]J\boldsymbol{\omega}_B\right) \quad \text{where} \quad \boldsymbol{N}_B = \sum_{i=1}^{4}\left(\boldsymbol{N}_{iB} + \boldsymbol{r}_{iB}\times\boldsymbol{F}_{iB}\right) \tag{17}$$

$$\dot{R}_{BI} = -[\boldsymbol{\omega}_B\times]R_{BI} \tag{18}$$

Functions (15) and (18) are explicit in that (15) simply defines the rate of change of position as velocity and (18) implements the theory that the derivative of the rotation matrix is derived by finding the negative cross product between it and the quadcopter angular rate vector ω_B. The latter equations are translational and rotational applications of Newton's second law of motion. However, due to the fact that this experiment generated a high fidelity model, two neglected effects are supplemented to update these latter equations.

$$m\ddot{\boldsymbol{r}}_I = -d_a\boldsymbol{v}_I^u - mg\boldsymbol{Z}_I + \sum_{i=1}^{4}\boldsymbol{F}_{iI} + \boldsymbol{d}_I \tag{19}$$

The translation equation of motion (16) is updated to account for the drag force d_a in the direction opposite to velocity $-\boldsymbol{v}_I^u$. Furthermore, to account for the fact that the rotor angular rates do not react instantaneously to a controlled voltage change, the transfer function detailing the relationship can be seen in (19).

$$\frac{\Omega(s)}{E_a(s)} = \frac{c_m}{\tau_m s + 1} \tag{19}$$

This phenomenon is further explained in the following section. Due to the fact that the angular rates are now varying with time, the rotor angular rate rate of change vector $\acute{\omega}_i$ must be added to the state variable in our ODE function to be solved by Range-Kutta. To model this effect, the $\acute{\omega}_i$ can be solved using (20) which utilizes a simplification of the transfer function (19).

$$\acute{\omega}_i = (E_a C_m - \omega_i)(\tau_m)^{-1} \tag{20}$$

The values C_{m} and τ_{m} are constants denoting the factor to convert motor voltage to motor angular rate in steady state in units rad/sec/volt and the unitless time constant governing the response time for input voltage both for each rotor motor in a 4x1 vector. The 4x1 vector for the applied voltage to each rotor motor is denoted as E_a and with C_{m} is used to estimate the target angular rate.

The formula in (17) is defined because the euler angles from the initial state matrix will be transformed, using the function in Fig. 5, to the equivalent rotation matrix. This is done to avoid the singularity error spurred by working with euler angles. Thus, the derivative state vector is formed, where $\dot{e}$ is replaced with the elements of $\dot{R}_{\mathrm{BI}}$ (unpacked column by column) to provide a now 22x1 state matrix. An overview of the process can be seen in Fig. 7.

```
Unset
function [Xdot] = quadOdeFunction(t,X,omegaVec,distVec,P)

% Determine forces and torques for each rotor from rotor angular rates.  The
% ith column in FMat is the force vector for the ith rotor, in B.  The ith
% column in NMat is the torque vector for the ith rotor, in B.
FMat = [zeros(2,4);(P.quadParams.kF.*(omegaVec.^2))'];
NMat = [zeros(2,4);(P.quadParams.kN.*(omegaVec.^2).*(-P.quadParams.omegaRdir)')'];

omegaBx = crossProductEquivalent(omegaB);
zI = RBI(3,:)';

% Calculate drag coefficient
epsilon_vI = 1e-3;
da = 0; vIu = [1;0;0];
if(norm(vI) > epsilon_vI)
 vIu = vI/norm(vI);
 fd = abs(zI'*vIu)*norm(vI)^2;
 da = 0.5*P.quadParams.Cd*P.quadParams.Ad*P.constants.rho*fd;
end

% Find derivatives of state elements
rIdot = vI;
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + distVec - da*vIu)/mq;
RBIdot = -omegaBx*RBI;
NB = sum(NMat,2);
for ii=1:4
 NB = NB + cross(P.quadParams.rotor_loc(:,ii),FMat(:,ii));
end
omegaBdot = inv(Jq)*(NB - omegaBx*Jq*omegaB);
omegaVecdot = (eaVec.*P.quadParams.cm - omegaVec)./P.quadParams.taum;

% Output state derivative vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];
```

Figure 7: Quadrotor ordinary differential equation function

The output of this function is the derivative state vector $\dot{X}$, like (14) in construct, with the addition of the rotor angular velocity rate of change. This function provides the true state, which will be used to model the noisy sensor measurements that must be used to estimate the state. Though this system may seem counter-intuitive, the application of modeling noisy sensor measurements allows for a more realistic simulation of UAV flight. As a result, the entire system of true state calculation, sensor measurement modeling, and noisy measurement state estimation can be compiled into the quadrotor dynamics plant denoted in the feedback schematic below.

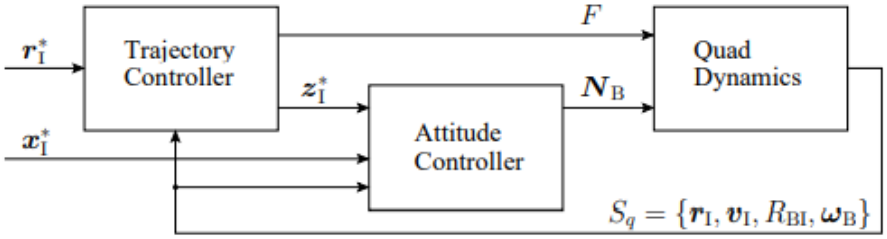


Figure 8: Top-level quadrotor control architecture

The next phase of implementation is to create the two control schemes seen in Fig. 8. The trajectory controller takes in the reference center of mass position vector in the inertial frame $\boldsymbol{r}_I^*$ and outputs the total thrust force the quadcopter needs to generate F and the desired direction of the body axis expressed in the inertial frame $\boldsymbol{z}_I^*$.

$$\boldsymbol{F}_I^* = k\boldsymbol{e}_r + k_d\dot{\boldsymbol{e}}_r + mg\boldsymbol{e}_3 + m\ddot{\boldsymbol{r}}_I^* \tag{21}$$

$$\boldsymbol{e}_r = \boldsymbol{r}^*_I - \boldsymbol{r}_I \tag{22}$$

$$\boldsymbol{z}_I^* = \frac{\boldsymbol{F}_I^*}{\|\boldsymbol{F}_I^*\|} \tag{23}$$

The position error vector $\boldsymbol{e}_r$ derived from the reference and calculated position vectors facilitate the calculation of the desired force using the proportional and derivative control vectors, the desired acceleration vector $\ddot{r}_I^*$, and gravity. With the desired force vector $\boldsymbol{F}_I^*$, the the calculation of z_I^* can be seen in (23). Equations (21) and (23) are brought together to calculate the output of the trajectory control in (24).

$$F = \boldsymbol{F}_I^* \cdot \boldsymbol{z}_I = (\boldsymbol{F}_I^*)^T R_{BI}^T \boldsymbol{e}_3 \tag{24}$$

This process is laid out in Fig. 9 with the code for the trajectory controller.

```
Unset
function [Fk,zIstark] = trajectoryController(R,S,P)

% Find error vectors
er = rIstark - rI;
er_dot = vIstark - vI;

% Control constants
K = [4 0 0; 0 4 0; 0 0 4];
Kd = [1.5 0 0; 0 1.5 0; 0 0 1.5]; % Both 3x3

% Finding total force
FIstark = K*er + Kd*er_dot + [0; 0; m*g] + m*aIstark;
zIstark = FIstark./norm(FIstark);
e3 = [0 0 1]';
zI = (RBI')*e3;
Fk = (FIstark')*zI;
```

Figure 9: Trajectory controller function

To create the attitude controller evident in Fig. 8, we instead substitute $-\omega_B$ for $\dot{e}_E$ because of the complexity of deriving $\dot{e}_E$ from the attitude controller inputs z_I^* and x_I^*. This can be done for small error angles where the desired angular rate ω_B is zero evident from (25).

$$\dot{e}_E \approx \omega_B^* - \omega_B \tag{25}$$

$$N_B = Ke_E - K_d\omega_B + [\omega_B\times]J\omega_B \tag{26}$$

Thus, we arrive at (26) for the attitude controller.

```
Unset
function [NBk] = attitudeController(R,S,P)

% Small angle assumption
eE_dot = wB;

% Control parameters
```

```
K = [1 0 0; 0 1 0; 0 0 1];
Kd = [0.3 0 0; 0 0.3 0; 0 0 0.3];

% Deriving RE using equation (8)
b = cross(zIstark, xIstark)/(norm(cross(zIstark, xIstark))); % 3x1
a = cross(b,zIstark); % 3x1
RBIstark = [a, b, zIstark]'; % 3x3
RE = RBIstark*(RBI');

% Using equation (9)
eE = [RE(2,3)-RE(3,2), RE(3,1)-RE(1,3), RE(1,2)-RE(2,1)]'; % 3x1
NBk = K*eE - Kd*eE_dot + crossProductEquivalent(wB)*J*wB;
```

Figure 10: Attitude control function

The final piece in the control loop is a hidden function from Fig 8. where the total force F and control torque $\boldsymbol{N}_B$ vectors, output values from both control functions, are converted to voltages for each rotor motor to be *input* into the plant ODE function. The following equation is used to calculate the respective forces necessary for each rotor to generate to satisfy the control parameters.

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ y_1 & y_2 & y_3 & y_4 \\ -x_1 & -x_2 & -x_3 & -x_4 \\ -k_T & k_T & -k_T & k_T \end{bmatrix}^{-1} \begin{bmatrix} \min(F, 4\beta F_{\max}) \\ \alpha \boldsymbol{N}_B \end{bmatrix} \tag{27}$$

The last vector in the above equation consists of the minimum value between the control total thrust force and the max force output of each rotor F_{max} multiplied by four, and the elements of the control torque vector $\boldsymbol{N}_B$. Furthermore, k_T represents the ratio between the k_N and k_F torque and thrust constants. The alpha and beta values are used to ensure some minimum amount of rotor thrust is allocated to applying the torque N_B. Finally, x_i and y_i demarcate the ±1 coordinate positions of the rotors apparent in Fig. 5. The calculation of the rotor forces must satisfy the limitation condition of $0 \leq F_i \leq F_{max}$, where the maximum force output from a rotor motor F_{max} can be calculated using (28) and (29).

$$F_{max} = k_F(\omega^2_{max}) \tag{28}$$

$$\omega_{max} = C_m e_{a,max} \tag{29}$$

The known maximum voltage vector $e_{a,max}$ applied to each rotor allows the derivation of the maximum rotor force F_{max}. With the generation of the rotor force vector F_i from (27) equations (28) and (29) are used again to determine the desired voltage vector for each rotor motor to be input into the quadcopter ODE function.

With the control architecture developed, the primary alteration of this experiment can blossom. For ease of understanding, the model simulator is depicted. Given an arbitrary initial state structure, the Runge Kutta ordinary differential equation solver in Matlab allows for visualization of the quadrotor dynamics across a preset sample time. The basic implementation of the control system in Fig. 8 is presented in the Fig. 11 simulator function.

```
Unset
function [Q] = simulateQuadrotorEstimationAndControl(R,S,P)


for kk=1:N-1
 % True state output
 statek.rI = Xk(1:3);
 statek.RBI(:) = Xk(7:15);
 statek.vI = Xk(4:6);
 statek.omegaB = Xk(16:18);
 statek.aI = Xdotk(4:6);
 statek.omegaBdot = Xdotk(16:18);
 Sm.statek = statek;

 % Simulate measurements
 M.tk=dtIn*(kk-1);
 [M.rpGtilde,M.rbGtilde] = gnssMeasSimulator(Sm,P);
 M.rxMat = [];
 for ii=1:Nf
   rx = hdCameraSimulator(S.rXIMat(ii,:)',Sm,P);
   if(isempty(rx))
     M.rxMat(ii,:) = [NaN,NaN];
   else
     M.rxMat(ii,:) = rx';
   end
 end
 [M.ftildeB,M.omegaBtilde] = imuSimulator(Sm,P);

 % Call estimator
 E = stateEstimatorUKF(Se,M,P);
 if(~isempty(E.statek))
   % Call trajectory and attitude controllers
   [Fk,Rac.zIstark] = trajectoryController(Rtc,Sc,P);
   NBk = attitudeController(Rac,Sc,P);
   % Convert commanded Fk and NBk to commanded voltages
   eaVeck = voltageConverter(Fk,NBk,P);
 else
   % Apply no control if the state estimator's output is empty.
   eaVeck = zeros(4,1);
   distVeck = [0;0;P.quadParams.m*P.constants.g];
 end
```

Figure 11: Quadcopter dynamics simulator

This function outputs a structure containing matrix values to define all states through every iteration of time, till the defined sampling time. With this, a visualization is processed. As evident from the simulation model, the true state using the function in Fig. 5 is determined. The true state generated, as mentioned before, is used to simulate noisy sensor measurements to *estimate* the quadrotor state. This estimated state is fed into the control loop, and using the reference trajectory, the trajectory controllers determine the next state to minimize the difference. Upon the next iteration, this state is fed into the high fidelity dynamics function to derive the next true state for which this process continues in the simulator above.

To make this system functional, the GNSS sensor model is depicted below.

```
Unset
function [rpGtilde,rbGtilde] = gnssMeasSimulator(S,P)

%% Primary
% Transform the inertial ECEF frame to the ENU frame, where vEnu = R*vEcef
RLG = Recef2enu(r0G);

% Find RpG
RpG = inv(RLG)*RpL*inv(RLG');

% Finding coordinates of primary antenna in I
rpI = rI + (RBI')*ra1B;

% Finding coordinates of primary antenna in G
rpG = (RLG')*rpI;

% Simulate noise vector
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rpGtilde
rpGtilde = rpG + w;

%% Baseline
% Finding coordinates of secondary antenna in I
rbI = rI + (RBI')*ra2B;

% Finding coordinates of secondary antenna in G
rbG = (RLG')*rbI;

% Find RbG
RbG = ((norm(rbG)^2)*(sigmab^2)).*(eye(3) - (rubG*(rubG'))) + epsilon.*eye(3);

% Simulate noise vector
w = mvnrnd(zeros(3,1), covarMatrix)';
```

```
% Finding rbGtilde with constraint norm(rbGtilde) = norm(rbG)
rbGtilde = norm(rbG).*rbGtildeUnit;
```

Figure 12: Noisy GNSS sensor measurement model

The responsibility of the GNSS model is to derive measurement values for the position of the primary and secondary antennas. The method for the primary and baseline measurement can be shown in (1) (2) and (3). However, for the primary measurement, the covariance matrix R_{pG} is found using (30) instead of (4).

$$R_{pL} = R_{LG} R_{pG} R_{LG}^{T} \tag{30}$$

In solving for R_{pG} the value of R_{LG} is the known rotation matrix from G to the L frame, where G is the Earth Centered Earth Fixed (ECEF) frame and L is approximately the Inertial I frame; the value of R_{pL} is the known error covariance matrix for either primary or secondary antenna expressed in the L frame. Further calculations viewed in Fig. 12.

```
Unset
function [ftildeB,omegaBtilde] = imuSimulator(S,P)

% Find RI double dot
RIdd = aI;

% Find noise vector
va = mvnrnd(zeros(3,1), covarMatrix1)';
va2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find accelerometer bias, update upon each call to the model persistent ba
if (isempty(ba))
    % Set ba's initial value
    QbaSteadyState = Qa2/(1 - alphaa^2);
    ba0 = mvnrnd(zeros(3,1), QbaSteadyState)';
    ba = [ba, ba0];
else
    bak1 = alphaa.*ba(:,end) + va2;
    ba = [ba, bak1];
end

% Find fBtilde
ftildeB = RBI*(RIdd + g.*e3) + ba(:,end) + va;
```

```
%% Angular Rates
% Find noise vector
vg = mvnrnd(zeros(3,1), covarMatrix1)';
vg2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find gyroscope bias, update upon each call to the model persistent bg;
bgk1 = alphag.*bg(:,end) + vg2;
bg = [bg, bgk1];

% Find omegaBtilde
omegaBtilde = omegaB + bg(:,end) + vg;
```

Figure 13: Noisy IMU sensor measurement model

The crux of the IMU sensor measurement model is (8) and (12). Thus, the output of the model becomes the IMU measurement of specific force and quadrotor angular rate. The values are updated upon each call to the function using persistent variable types in Matlab.

```
Unset
function [rx] = hdCameraSimulator(rXI,S,P)

% Solving for RCI
RCI = RCB * RBI;

% Solving for t
t = -RCI*(rI + ((RBI')*rocB));

% Solving for projection matrix P
K = [K,zeroMatrix'];
PMat = K*[RCI t; zeroMatrix 1];

% Solving for x using PX
xh = PMat*X;

% If feature is not in camera image plane, rx = []
if xh(3) <= 0
    return % If x(3) is negative, the feature is behind the camera (z-plane)
end

% Finding xc
x = xh(1)/xh(3);
y = xh(2)/xh(3);
```

```
xc = [x y]';

% Finding noise vector wc
w = mvnrnd(zeros(2,1), covarMatrix)';

% Finding xctilde
xctilde = (1/pixelSize).*xc + w;

% Check if xctilde is inside the camera detection plane
if featureCameraX <= imagePlaneX && featureCameraY <= imagePlaneY
    rx = xctilde;
else
    rx = [];
end
```

Figure 14: Noisy camera measurement model

For the last of the sensor models, the camera outputs the measured position of the feature point projection on the camera's image plane. These features are predefined in 3D space and, if not in the camera's field of view, do not contribute to the estimation of the quadrotor state. Using the homogenous point projection discussed in section 2.2, the noisy position vector is determined. Given some set of homogeneous coordinates X = [X, Y, Z, 1]^T in the I frame, it is possible to map the coordinates to the camera image plane $\boldsymbol{x}_{ci}$ using (31).

$$\underbrace{\begin{bmatrix} fX_c \\ fY_c \\ Z_c \end{bmatrix}}_{\boldsymbol{x}_{ci}} = \left[\underbrace{\begin{matrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{matrix}}_{K} \middle| \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}\right] \begin{bmatrix} R_{\mathrm{CI}} & \boldsymbol{t}_{\mathrm{C}} \\ 0_{1\times 3} & 1 \end{bmatrix} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\boldsymbol{X}_{\mathrm{I}}} \tag{31}$$

The above equation uses the focal length of the camera f, the rotation matrix from the inertial to the camera frame R_{CI}, and the translation vector $t_{\mathrm{C}} = R_{\mathrm{CI}}t_{\mathrm{I}}$ expressed in the camera frame C. The implementation and derivation of further values, such as t_{I}, can be found in Fig. 14. After finding $\boldsymbol{x}_{ci}$, the 3D vector was converted to the 2D image plane using the $[x/z, y/z]^T$ method previously mentioned. Then to create the noisy measurement of feature position in the C frame, the following equation was implemented.

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} = \frac{1}{p_s} \begin{bmatrix} x \\ y \end{bmatrix} + \underbrace{\begin{bmatrix} w_x \\ w_y \end{bmatrix}}_{\boldsymbol{w}_c} \tag{32}$$

The pixel size p_s is used to scale the true projection and a generated noise vector, using a predefined covariance matrix and similar methods as the previous sensor models, is used to slightly alter it.

With the creation of the sensor models, combining the highly nonlinear systems can prove extremely complex. The sensor measurements are used to reconstruct the state and to do this- a provided function for an unscented Kalman filter (UKF) is used for estimation. To initialize the UKF, a function created to solve Wahba's problem is used to get a reasonable first estimate of attitude. The application can be seen in Fig. 15.

```
Unset
function [RBI] = wahbaSolver(aVec,vIMat,vBMat)

% Find B
B = zeros(3);
for i=1:length(aVec)
    a = aVec(i);
    V = vIMat(i,:)';
    U = vBMat(i,:)';
    B = B + a.*U*V';
end

% Find U and V from B = USV' given B
[U,S,V] = svd(B);

% Initilize M
M = [1 0 0; 0 1 0; 0 0 det(U)*det(V)];

% Solve for
RBI = U*M*V';
```

Figure 15: Wabha's Problem solver

Next, to create the state estimator, a measurement function is created which is used to output a measurement matrix of the GNSS and camera sensor using the previously defined functions. The calculation of (33) can be seen in the following figure.

$$\boldsymbol{z}(k) = \boldsymbol{h}[\boldsymbol{x}(k)] + \boldsymbol{w}(k) \tag{33}$$

Where $\boldsymbol{z}(k)$ is the vector of measurements and $\boldsymbol{h}[\boldsymbol{x}(k)]$ is a conversion function of the true state.

```
Unset
function [zk] = h_meas(xk,wk,RBIBark,rXIMat,mcVeck,P)
```

```
% Find predicted RBI matrix
RBI = euler2dcm(er)*RBIBark;

% Set location of C frame origin in I
rcI = rI + (RBI')*rocB;

% Solve for vector pointing from primary to secondary
rbB = ra2B - ra1B;
rubB = rbB./norm(rbB);

% Solve for first two vectors of z(k)
zk = [rI + (RBI')*ra1B; (RBI')*rubB] + wk(1:6);

% Solve for the feature vectors in z(k)
for i = 1:length(mcVeck)
    % If detected (nonzero), assess
    if mcVeck(i) ~= 0
        % Find vector pointing from camera center to the ith 3D feature in I
        viI = rXIMat(i,:)' - rcI;

        % Normalize for attitude calculation
        vuiI = viI./norm(viI);

        % Finding 3x1 noise vector from wk
        wki = wk(startIndex:endIndex);

        % Solve for vector in C frame
        vuiC = RCB*RBI*vuiI + wki;

        % Update zk for features
        zk = [zk;vuiC];
    end
end
```

Figure 16: Sensor measurement model

Similarly, a dynamics model is created to derive the propagated state from the IMU measurements $\boldsymbol{u}(k)$, current state $\boldsymbol{x}(k)$, and noise measurements $\boldsymbol{v}(k)$ functions.

$$\boldsymbol{x}(k+1) = \boldsymbol{f}\left[\boldsymbol{x}(k), \boldsymbol{u}(k), \boldsymbol{v}(k)\right] \tag{34}$$

In expanding (34) to define the vector-valued function $\boldsymbol{f}$, the equation (35) is applied to the following function to propagate the measured state.

$$\underbrace{\begin{bmatrix} \boldsymbol{r}_{\mathrm{I}}(k+1) \\ \boldsymbol{v}_{\mathrm{I}}(k+1) \\ \boldsymbol{e}(k+1) \\ \boldsymbol{b}_a(k+1) \\ \boldsymbol{b}_g(k+1) \end{bmatrix}}_{\boldsymbol{x}(k+1)} = \underbrace{\begin{bmatrix} \boldsymbol{r}_{\mathrm{I}}(k) + \Delta t \boldsymbol{v}_{\mathrm{I}}(k) + \frac{1}{2}(\Delta t)^2 \boldsymbol{a}_{\mathrm{I}}(k) \\ \boldsymbol{v}_{\mathrm{I}}(k) + \Delta t \boldsymbol{a}_{\mathrm{I}}(k) \\ \boldsymbol{e}(k) + \Delta t \dot{\boldsymbol{e}}(k) \\ \alpha_a \boldsymbol{b}_a(k) + \boldsymbol{v}_{a2}(k) \\ \alpha_g \boldsymbol{b}_g(k) + \boldsymbol{v}_{g2}(k) \end{bmatrix}}_{\boldsymbol{f}[\boldsymbol{x}(k),\boldsymbol{u}(k),\boldsymbol{v}(k)]} \tag{35}$$

The estimated state is defined as the position in the I frame $\boldsymbol{r}_{\mathrm{I}}$, velocity in the I frame $\boldsymbol{v}_{\mathrm{I}}$, euler angles $\boldsymbol{e}$, accelerometer bias $\boldsymbol{b}_{\mathrm{a}}$, and gyro bias $\boldsymbol{b}_{\mathrm{g}}$, as shown above. These state values, alongside the output of the function in Fig. 16 and the Wahba problem solver drive the UKF function.

```
Unset
function [xkp1] = f_dynamics(xk,uk,vk,delt,RBIHatk,P)

% Find aIk
ge3 = [0 0 g]';
aIk = ((RBIk')*(fBtildek - bak - vak)) - ge3;

% Find edotk
omegabk = omegaBtildek - bgk - vgk;
S = (1/cos(phik)).*[cos(phik)*cos(thetak), 0, cos(phik)*sin(thetak);
                    sin(phik)*sin(thetak), cos(phik), -cos(thetak)*sin(phik);
                    -sin(thetak), 0, cos(thetak)];
edotk = S*omegabk;

% Put together output vectors
rIkp1 = rIk + deltat.*vIk + 0.5*(deltat^2)*aIk;
vIkp1 = vIk + deltat.*aIk;
ekp1 = ek + deltat.*edotk;
bakp1 = alphaa*bak + va2k;
bgkp1 = alphag*bgk + vg2k;

% Output
xkp1 = [rIkp1;vIkp1;ekp1;bakp1;bgkp1];
```

Figure 17: Measurement-based dynamics model

With the application of the function in Fig. 17, the provided UKF function for this experiment becomes operational. In integrating these functions, a top level function to call the simulator is employed to assess the control application in this experiment.

```
Unset
rng('shuffle');

% Total simulation time, in seconds
Tsim = 11;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;

% Populate reference trajectory
R.tVec = tVec;
R.rIstar = readmatrix("position.txt");
R.vIstar = readmatrix("velocity.txt");
R.aIstar = readmatrix("acceleration.txt");
% The desired xI points toward the origin. The code below also normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);

% Matrix of disturbance forces acting on the body, in Newtons, expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [0 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 0]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 2;
% Feature locations in the I frame
S.rXIMat = [];
```

Figure 18: Quadrotor top level simulation script

As evident in Fig. 18, the generated position, velocity, and acceleration from the path planning algorithm populate the reference trajectory structure. The reference yaw over time is created using the generated position.

For this experiment, various path planning methods were compared. The baseline was created with the Depth First Search method. Using a grid laid over a predetermined map, a start and end node are indicated. The algorithm keeps track of the explored grid nodes and the indicated nodes to explore as the agent moves. Upon each iteration, the method checks if the node has been explored before and if that node is the goal. If neither, the current node is entered into the explored nodes list and the neighboring nodes are entered into the nodes to explore list. This method is detailed in Fig. 19.

Unset

```
// Use these data structures
std::stack<NodeWrapperPtr> to_explore; // nodes to explore
std::vector<NodeWrapperPtr> explored; // explored nodes
PathInfo path_info; // Output structure

// Create a NodeWrapperPtr
NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
    nw_ptr->parent = nullptr;
    nw_ptr->node_ptr = start_ptr;
    nw_ptr->cost = 0;
    to_explore.push(nw_ptr);

// Check if the exploration pointer is empty
while (!to_explore.empty()) {

// Find the item to explore using first value of to_explore and remove
NodeWrapperPtr item_to_explore = to_explore.top();
to_explore.pop();

// Check if item_to_explore has already been explored
if (is_present(item_to_explore, explored)) {
      continue;
}

// Add node to explored vector
explored.push_back(item_to_explore);

// Check if agent reached the end node
if (*item_to_explore->node_ptr == *end_ptr) {
      break;
}

// If not the end then find next path

// Use graph to find the neighbor nodes of the node_to_explore
auto edges = graph.Edges(item_to_explore->node_ptr);

// Push all neighbors nodes to to_explore
for(auto edge : edges) {

      // Create a NodeWrapperPtr for each neighbor node
      NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();

      // Update exploring node to parent node
      neighbor_ptr->parent = item_to_explore;

      // Set neighbor node pointer coordinates
```

```
        neighbor_ptr->node_ptr = edge.Sink();

        // Set cost of neighbor exploration
        neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

        // Push neighbor instance to the to_explore stack for next iteration
        to_explore.push(neighbor_ptr);
}

return path_info;
```

Figure 19: Depth First Search algorithm

Note that the *to_explore* variable is a stack, which stacks the nodes in the order they were appended. Thus, the next explored node is always the first item from the stack. This method is then expanded by accounting for the cost of travel to each node. In an attempt to minimize the cost of arriving at the goal, Depth First Search was adapted using Dijkstra's algorithm. The implementation can be seen below, where the alterations are highlighted.

Unset

```
// Use these data structures
std::priority_queue<
      NodeWrapperPtr,
      std::vector<NodeWrapperPtr>,
      std::function<bool(
             const NodeWrapperPtr&,
             const NodeWrapperPtr& )>>
      to_explore(NodeWrapperPtrCompare); // To explore priority queue
std::vector<NodeWrapperPtr> explored; // Explored vector
PathInfo path_info; // Output structure

// Create a NodeWrapperPtr
NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
    nw_ptr->parent = nullptr;
    nw_ptr->node_ptr = start_ptr;
    nw_ptr->cost = 0;
    to_explore.push(nw_ptr);

// Check if the exploration pointer is empty
while (!to_explore.empty()) {

// Find the item to explore using first value of to_explore and remove
NodeWrapperPtr item_to_explore = to_explore.top();
to_explore.pop();
```

```
// Check if item_to_explore has already been explored
if (is_present(item_to_explore, explored)) {
       continue;
}

// Add node to explored vector
explored.push_back(item_to_explore);

// Check if agent reached the end node
if (*item_to_explore->node_ptr == *end_ptr) {
       break;
}

// If not the end then find next path

// Use graph to find the neighbor nodes of the node_to_explore
auto edges = graph.Edges(item_to_explore->node_ptr);

// Push all neighbors nodes to to_explore
for(auto edge : edges) {

       // Create a NodeWrapperPtr for each neighbor node
       NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();

       // Update exploring node to parent node
       neighbor_ptr->parent = item_to_explore;

       // Set neighbor node pointer coordinates
       neighbor_ptr->node_ptr = edge.Sink();

       // Set cost of neighbor exploration
       neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

       // Push neighbor instance to the to_explore stack for next iteration
       to_explore.push(neighbor_ptr);
}

return path_info;
```

Figure 20: Dijkstra's algorithm

The red section in Fig. 20 denotes the change of the *to_explore* variable from a stack to a priority queue. This enables the algorithm to rank the nodes to be explored in order of increasing cost. As a result, the next explored node will be the node of least cost amongst the neighbors found in the previous iteration. To enhance this algorithm, a heuristic is implemented in a method known as A*. As hinted in the previous section, the developed A* algorithm was given knowledge of the euclidean distance from the current to the goal node. Using this information, the algorithm obtains a sense of location awareness with reference to the

goal. The addition can be seen in Fig. 21.

```
Unset
// HEURISTIC FUNCTION
double Heuristic(
             const std::shared_ptr<Node2D>& current_ptr,
             const std::shared_ptr<Node2D>& end_ptr) {
       double deltax = end_ptr->Data().x() - current_ptr->Data().x();
       double deltay = end_ptr->Data().y() - current_ptr->Data().y();
       float euclidean = pow(((deltax * deltax) + (deltay * deltay)), 0.5);
       return euclidean;
}

// Use these data structures
std::priority_queue<
      NodeWrapperPtr,
      std::vector<NodeWrapperPtr>,
      std::function<bool(
             const NodeWrapperPtr&,
             const NodeWrapperPtr& )>>
      to_explore(NodeWrapperPtrCompare); // To explore priority queue
std::vector<NodeWrapperPtr> explored; // Explored vector
PathInfo path_info; // Output structure

// Create a NodeWrapperPtr
NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
    nw_ptr->parent = nullptr;
    nw_ptr->node_ptr = start_ptr;
    nw_ptr->cost = 0;
    to_explore.push(nw_ptr);

// Check if the exploration pointer is empty
while (!to_explore.empty()) {

// Find the item to explore using first value of to_explore and remove
NodeWrapperPtr item_to_explore = to_explore.top();
to_explore.pop();

// Check if item_to_explore has already been explored
if (is_present(item_to_explore, explored)) {
       continue;
}

// Add node to explored vector
explored.push_back(item_to_explore);

// Check if agent reached the end node
if (*item_to_explore->node_ptr == *end_ptr) {
```

```
        break;
}

// If not the end then find next path

// Use graph to find the neighbor nodes of the node_to_explore
auto edges = graph.Edges(item_to_explore->node_ptr);

// Push all neighbors nodes to to_explore
for(auto edge : edges) {

        // Create a NodeWrapperPtr for each neighbor node
        NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();

        // Update exploring node to parent node
        neighbor_ptr->parent = item_to_explore;

        // Set neighbor node pointer coordinates
        neighbor_ptr->node_ptr = edge.Sink();

        // Update heuristic value
        neighbor_ptr->heuristic = Heuristic(neighbor_ptr->node_ptr, end_ptr);

        // Set cost of neighbor exploration
        neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

        // Push neighbor instance to the to_explore stack for next iteration
        to_explore.push(neighbor_ptr);
}

return path_info;
```

Figure 21: A* algorithm

The added heuristic function can be isolated in Fig. 21 to the highlighted portion. By providing the algorithm with this knowledge, the priority queue is also able to rank neighboring nodes in terms of exploration cost and the added heuristic cost- ensuring the agent moves closer to the goal optimally. With the completion of the path planning algorithm, the developed nodal path needs to be converted to a continuous trajectory over time. As a result, a polynomial planning script was developed. The primary function comes from the P4 repository and allows the optimization of a select polynomial derivative, like velocity or acceleration, to be minimized.

```
Unset
int main(int argc, char** argv) {
```

```
  DerivativeExperiments();
  ArrivalTimeExperiments();
  NumWaypointExperiments();

  return EXIT_SUCCESS;
}

void DerivativeExperiments() {
  // Time in seconds
  const std::vector<double> times = {1,2,3,4,5};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ////////////////////////////////////////////////////////////////////
    // TODO: CREATE A SQUARE TRAJECTORY
    ////////////////////////////////////////////////////////////////////

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constraints position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constraints position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The third node constraints position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,1),

    // The fifth node constraints position
    p4::NodeEqualityBound(0,4,0,0),
    p4::NodeEqualityBound(1,4,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
```

```
  solver_options.derivative_order = 0;   // TODO: VARY THE DERIVATIVE ORDER

  }
}

void ArrivalTimeExperiments() {
  // Time in seconds
  const std::vector<double> times = {1,2,3,4,5};
  // const std::vector<double> times = {0.1,0.2,0.3,0.4,0.5};
  // const std::vector<double> times = {10,20,30,40,50}

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ////////////////////////////////////////////////////////////////////
    // SQUARE TRAJECTORY
    ////////////////////////////////////////////////////////////////////

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constraints position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constraints position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The third node constraints position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,1),

    // The fifth node constraints position
    p4::NodeEqualityBound(0,4,0,0),
    p4::NodeEqualityBound(1,4,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
```

```
  solver_options.derivative_order = 4;   // Minimize snap

  }
}

void NumWaypointExperiments() {
  // Independent variables
  double N = 4;
  int r = 1;

  // Time in seconds
  std::vector<double> times = {1,2,3,4,5};
  // std::vector<double> times = {1,2,3,4,5,6,7,8,9};
  // std::vector<double> times = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};

  // Initialize the vector
  std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ////////////////////////////////////////////////////////////////////
    // CIRCULAR TRAJECTORY
    ////////////////////////////////////////////////////////////////////

    // The first node must constrain position, velocity, and acceleration at (1,0)
    p4::NodeEqualityBound(0,0,0,r),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,r),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,r),
    p4::NodeEqualityBound(1,0,2,0),

  };

  // Add the other nodes

  // Set PI
  const double PI = M_PI;

  // Set the angle variation
  double angle_shift = (2*PI)/N;
  double curr_angle = angle_shift;

  // Create a for loop that varies with N waypoints
  for (double i = 1; i <= N; i++) {

    // Find the next point using the angle variation
    double x = r*cos(curr_angle);
    double y = r*sin(curr_angle);

    // Create waypoint node coordinates constrining position
```

```
    node_equality_bounds.push_back(p4::NodeEqualityBound(0,i,0,x));
    node_equality_bounds.push_back(p4::NodeEqualityBound(1,i,0,y));

    // Update curr_angle
    curr_angle += angle_shift;
  }


  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 4;   // Minimize snap

  }
}
```

Figure 22: Polynomial fitting script

In addition to implementing the polynomial fitting function, the script above showcases multiple tests that are discussed in the following section.

Finally, another script was constructed to run the A* algorithm on a specific map grid. In running the algorithm, the path is fed into the P4 polynomial fitting function. The trajectory position is then assessed to derive the velocity and acceleration time histories. These values are then saved and are written to an external file which is used to save the trajectory data. With these, the datafiles are then transported to the MATLAB simulator script found in Fig. 18 to populate the reference trajectory matrices. This final script, used to verify the general trajectory in conjunction with the simulator, is found below.

Unset

```
auto path_info_ret = RunAStar(graph, &occupancy_grid, start_node, end_node);
auto node_path = path_info_ret.path;

// Find the number of waypoints
int N = node_path.size();

// Time in seconds, a second per waypoint
std::vector<double> times = {};
for (double i = 0; i <= N; i++) {
      times.push_back(i);
}
```

```
  // Find the first node coordinates
  auto starting_node = node_path[0];
  auto x1 = starting_node->Data().x();
  auto y1 = starting_node->Data().y();

  // Initialize the waypoint vector
  std::vector<p4::NodeEqualityBound> node_equality_bounds = {

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,x1),
    p4::NodeEqualityBound(1,0,0,y1),
    p4::NodeEqualityBound(0,0,1,x1),
    p4::NodeEqualityBound(1,0,1,y1),
    p4::NodeEqualityBound(0,0,2,x1),
    p4::NodeEqualityBound(1,0,2,y1),

  };

  // Add the other nodes
  for (double i = 1; i < N; i++) {

    // Find node coordinates
    auto current_node = node_path[i];
    auto x = current_node->Data().x();
    auto y = current_node->Data().y();

    // Create waypoint node coordinates constrining position
    node_equality_bounds.push_back(p4::NodeEqualityBound(0,i,0,x));
    node_equality_bounds.push_back(p4::NodeEqualityBound(1,i,0,y));
  }

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 2;   ////////// CHANGE  Minimize the 2nd order

  // Download the time history of the desired derivative
  std::ofstream MyFile("./data.txt");

  // Create easiest format file
  for (int i = 0; i < t_hist.size(); i++) {
    MyFile<<x_hist[i]<<", "<<y_hist[i]<<", 0"<<std::endl;
  }

  return EXIT_SUCCESS;
}
```

```
// HELPER FUNCTION
PathInfo RunAStar(
    const Graph2D& graph,
    const OccupancyGrid2D* occupancy_grid,
    const std::shared_ptr<Node2D>& start_node,
    const std::shared_ptr<Node2D>& end_node) {

  // Run A*
  AStar2D a_star;
  PathInfo path_info = a_star.Run(graph, start_node, end_node);
  return path_info;
}
```

Figure 23: Full stack path planning executable script

These functions set up the foundation for conducting in-depth dynamical analysis on a quadcopter for various path planning configurations.

4 Results and Analysis

In this section, results of the lab experiments are discussed. Furthermore, the code and theory are tested to verify accurate implementation.

4.1 Depth First Search

With the algorithm presented in Fig. 19, Depth First Search is employed on a provided grid, denoted below, where it is unable to find the optimal path. The starting points and goal points are also indicated.

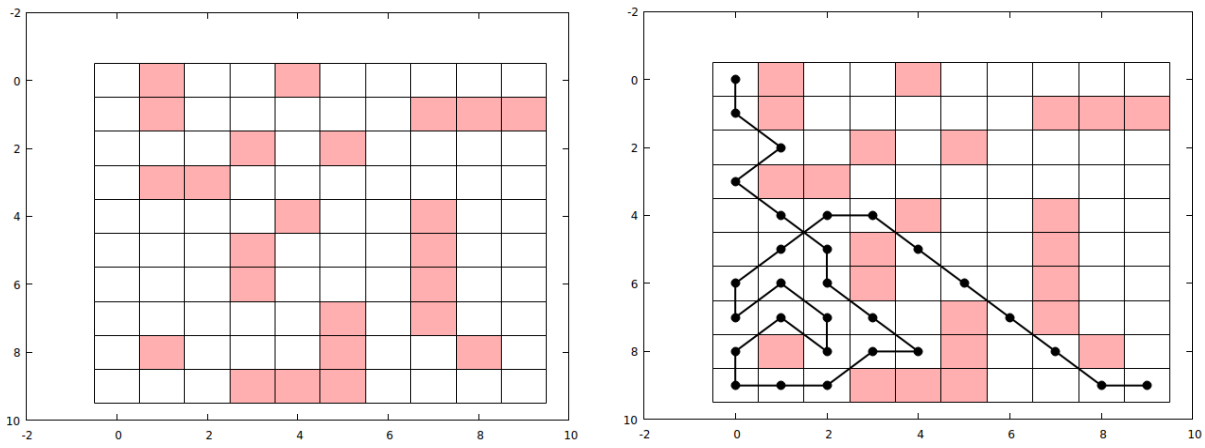


Figure 24a: Grid to be navigated

Figure 24b: Generated path using Depth First Search; start = (0,0); goal = (9,9)

From Fig. 24a, the optimal cost to the goal point is 13.90 measurement units, whereas from Fig. 24b, the cost of the path taken by the algorithm is much higher at 35.50 measurement units. The length of the path taken totals to 29 nodes, but for an optimal path, the desired number of nodes is 12.

4.2 Dijkstra's Algorithm

Moving onto Dijkstra's method of path planning, the algorithm in Fig. 20 is run on the same grid denoted in Fig. 24a. In comparison to the Depth First Search algorithm, Dijkstra's algorithm runs much slower with a runtime of 477 microseconds compared to the Depth First Search runtime of 96 microseconds. Furthermore, by accounting for cost of exploration, Dijkstra's algorithm generates a path with a total cost of 13.90 measurement units, significantly less than the cost of the Depth First Search algorithm denoted in section 4.1. It is also important to mention that Dijkstra's algorithm explored a total of 75 nodes where the Depth First Search algorithm only explored 31, contributing to the almost five times slower runtime. The generated path from Dijkstra's algorithm can be seen below.

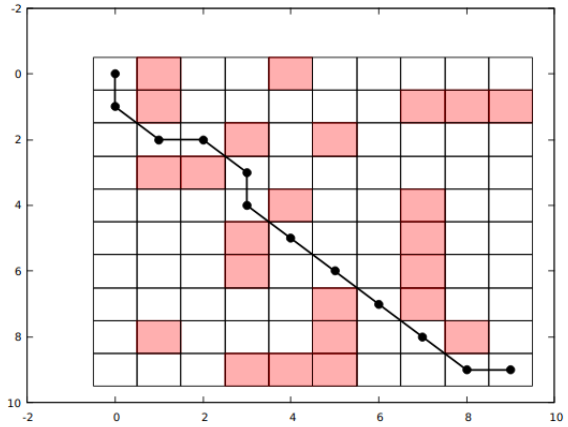


Figure 25: Generated path using Dijkstra's algorithm; start = (0,0); goal = (9,9)

As evident from Fig. 25, the generated path using this algorithm is much shorter than the initial. With reference to the optimal path cost, previously mentioned as 13.90, Dijkstra's algorithm is able to achieve this. However, as previously mentioned, the number of explored nodes is extremely high. In fact, this grid only has 76 explorable nodes, and neglecting the starting point- the algorithm iterated through all explorable nodes on the map. This is highly impractical and, alongside the high runtime respective to the baseline approach, presents an inefficient solution. Thus, the path planning approach transitions to the A* algorithm, discussed in the subsequent section.

4.3 A* Cost Function Overestimation and Underestimation

For A* to work, the heuristic function must be optimistic. This is called admissibility. As stated by *Engati*, "[for] a heuristic to be admissible to a search problem, [it] needs to be lower than or equal to the actual cost

of reaching the goal." This is the reason the euclidean distance is used as the heuristic of choice, because the calculated distance will always at *most* equal to the cost of reaching the goal. This is due to the fact that the euclidean straight line from point A to point B represents the shortest path between the two points.

```
Unset
double Heuristic(
              const std::shared_ptr<Node2D>& current_ptr,
              const std::shared_ptr<Node2D>& end_ptr) {
       double deltax = end_ptr->Data().x() - current_ptr->Data().x();
       double deltay = end_ptr->Data().y() - current_ptr->Data().y();
       float euclidean = pow(((deltax * deltax) + (deltay * deltay)), 0.5);
       return euclidean;
}
```

Figure 26: Euclidean distance heuristic function

To compare the performance of A* using a non-admissible heuristic, the manhattan distance is used to overestimate the search cost. This value represents a clear overestimation of the search cost by summing the X and Y distances from the goal. In using a non-admissible heuristic, the algorithm does not guarantee finding an optimal path, or the path of least cost. It is possible it can find a lower cost path than its admissible counterpart, but this inconsistency is why overestimating heuristics are not used.

```
Unset
double Heuristic(
              const std::shared_ptr<Node2D>& current_ptr,
              const std::shared_ptr<Node2D>& end_ptr) {
       double deltax = end_ptr->Data().x() - current_ptr->Data().x();
       double deltay = end_ptr->Data().y() - current_ptr->Data().y();
       float manhattan = abs(deltax) + abs(deltay);
       return manhattan;
}
```

Figure 27: Manhattan distance heuristic function

In running multiple tests, the above heuristic functions are integrated with A*to produce an underestimated and overestimated cost. The path results are seen below.

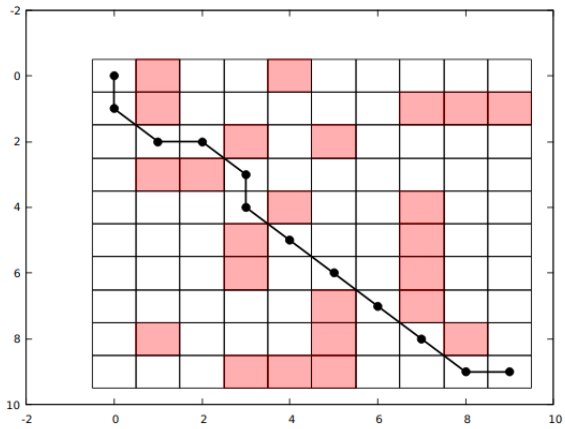


Figure 28a: Underestimated A* generated path; start = (0,0); goal = (9,9)

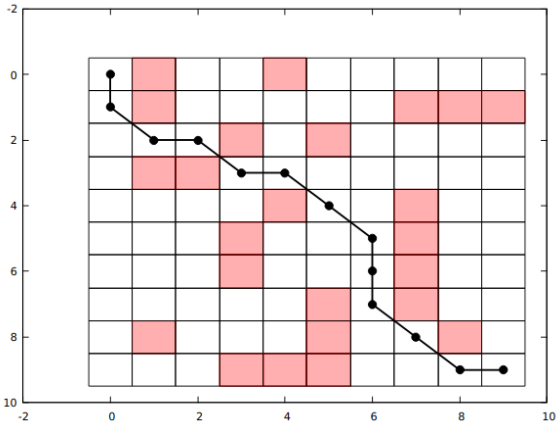


Figure 28b: Overestimated A* generated path; start = (0,0); goal = (9,9)

In the underestimated case, the A* algorithm explored 21 nodes and generated a path of 12 nodes with the optimal cost of 13.90 measurement units. Furthermore, this was completed within 89 microseconds. For the path shown in Fig. 28b, the path cost increased to 14.50 measurement units with 13 nodes in the path. The runtime remained relatively similar at 86 microseconds. The varied path due to the non-admissible heuristic resulted in a non-optimal or increased path cost. Comparing both to the prior algorithm, the number of explored nodes and runtime decreased. Thus, for the final simulation, the underestimating heuristic is used to most efficiently find the optimal path.

4.4 Designing Two Distinct Heuristic Functions for A*

In understanding the function of heuristics, an admissible heuristic can really be anything, so long as it constantly underestimates the path cost. The same can be said for a non-admissible heuristic- defined by constant overestimation. To showcase this concept, a variation of the euclidean and manhattan distance calculations were developed.

To pronounce the effect of underestimation and overestimation, the Euclidean distance was divided by two and the Manhattan distance was multiplied by two, as seen below.

```
Unset
double Heuristic(
             const std::shared_ptr<Node2D>& current_ptr,
             const std::shared_ptr<Node2D>& end_ptr) {
       double deltax = end_ptr->Data().x() - current_ptr->Data().x();
       double deltay = end_ptr->Data().y() - current_ptr->Data().y();
       float euclidean = pow(((deltax * deltax) + (deltay * deltay)), 0.5);
       float distinct = euclidean/2;
       return distinct;
}
```

Figure 29: Distinct admissible heuristic function

```
Unset
double Heuristic(
                    const std::shared_ptr<Node2D>& current_ptr,
                    const std::shared_ptr<Node2D>& end_ptr) {
       double deltax = end_ptr->Data().x() - current_ptr->Data().x();
       double deltay = end_ptr->Data().y() - current_ptr->Data().y();
       float manhattan = abs(deltax) + abs(deltay);
       float distinct = manhattan*2;
       return distinct;
    }
```

Figure 30: Distinct non-admissible heuristic function

These functions still work as heuristics, and their performance is evaluated in Fig. 31.

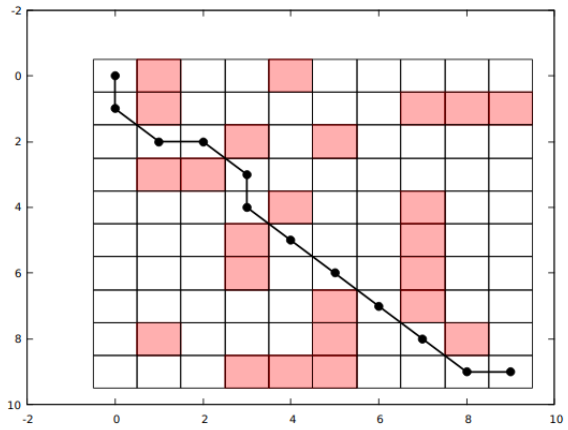


Figure 31a: Distinct admissible A* heuristic path; start = (0,0); goal = (9,9)

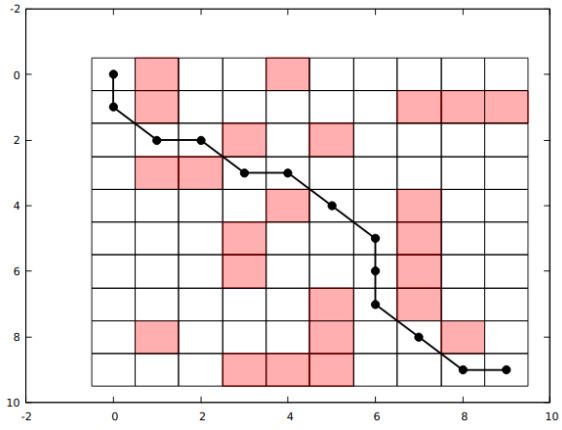


Figure 31b: Distinct non-admissible A* heuristic path; start = (0,0); goal = (9,9)

As shown from above, of the two distinct heuristic functions, the extra-underestimated function outperforms the extra-overestimated function. The path in Fig. 31a explored 69 nodes and achieved the optimal path cost of 13.90 measurement units with an understandably increased runtime. Contrastingly, the non-optimal path shown in Fig. 31b retained a path cost of 14.50 measurement units with 13 nodes in the path. The number of explored nodes was also 13, indicative of the tests' much shorter runtime.

The best performance of the A* algorithm occurs when the heuristic function is on the borderline of *breaking* the admissible definition. This can be seen by the fact that both paths displayed a worse performance, in terms of cost or efficiency, than the Euclidean distance heuristic used in section 4.4. As a result, this will be the heuristic function that A* employs for the best path generation.

4.5 Comparing A* Leveraging Zero Heuristic Function to Dijkstra's Algorithm

As indicated through the implementation section, the A* algorithm builds on Dijkstra's algorithm by adding a heuristic function. This allows the algorithm to produce a path that is more directed towards the end goal, instead of incurring wide, inefficient exploration seen in section 4.2.

In concept, by removing the heuristic, the A* algorithm reverts back to Dijkstra's algorithm. This is similar to creating a zero heuristic, or a heuristic function that always equates to zero, thus providing no information to the cost function. This theory is tested using a heuristic function that, regardless of the current node position, always returns zero for A*. The results of the A* algorithm with this configuration can be seen subsequently.

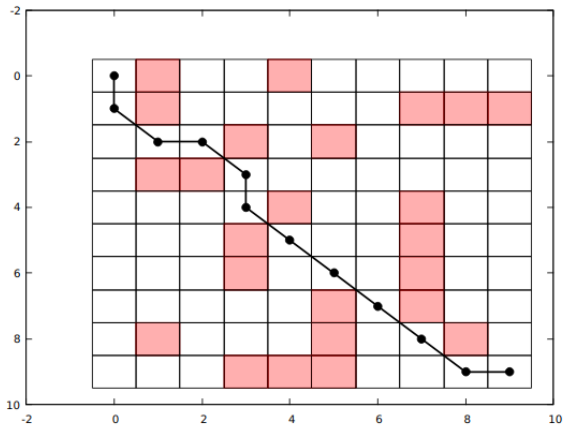


Figure 32: Generated path using A* algorithm with zero heuristic; start = (0,0); goal = (9,9)

As evident over the two tests, the algorithms perform almost identically. The paths created by the A* algorithm in Fig. 32 and Dijkstra's algorithm in Fig. 25 reflect this conclusion. Aside from the generated path, both algorithms explore 75 nodes and find the optimal path of cost 13.50 measurement units and 12 nodal steps. The runtime on both are similar with the path in Fig. 32 taking 463 microseconds.

Comparing the tested A* algorithm to its more optimal counterpart from section 4.3, it is possible to observe the impact of a functional heuristic. The generated path for the zero heuristic algorithm conducts much more exploration, even if still accounting for movement cost. This is due to the fact that without indication of where the goal is, the algorithm attempts to explore in all directions in the most cost effective way to cover all regions of the map.

4.6 Impact of Varying Derivative Minimized on Polynomial Smoothing

In conducting polynomial smoothing over the generated nodal path, the method outlined in Fig. 22 allows the user to indicate the order of derivative to be minimized for the 8th degree position polynomial. The derivative of the position polynomial on the 0th to the 6th order represents position, velocity, acceleration, jerk, snap, crackle and pop in sequence. Accounting for each property, and minimizing the respective value drastically alters the resulting trajectory.

For this assessment, derivatives of order zero through four are tested. Essentially, the effect of minimizing

position, velocity, acceleration, snap, and jerk on the generated trajectory is explored. The following test is run on the same square grid to compare trajectory configurations.

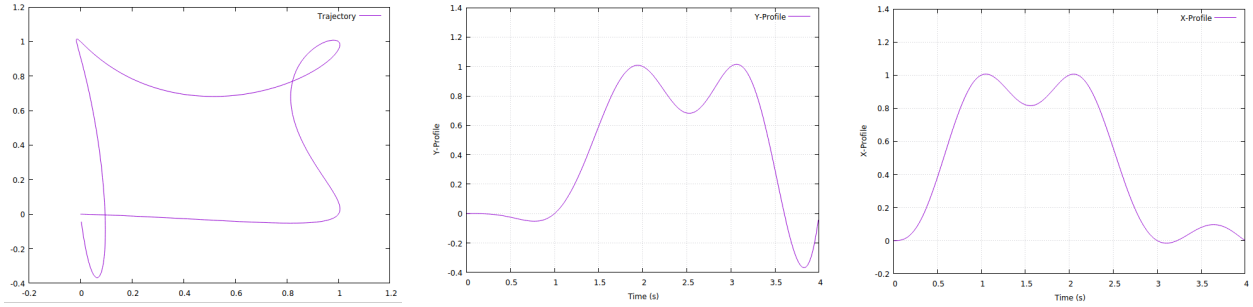


Figure 33: Generated trajectory minimizing position; zeroth order derivative

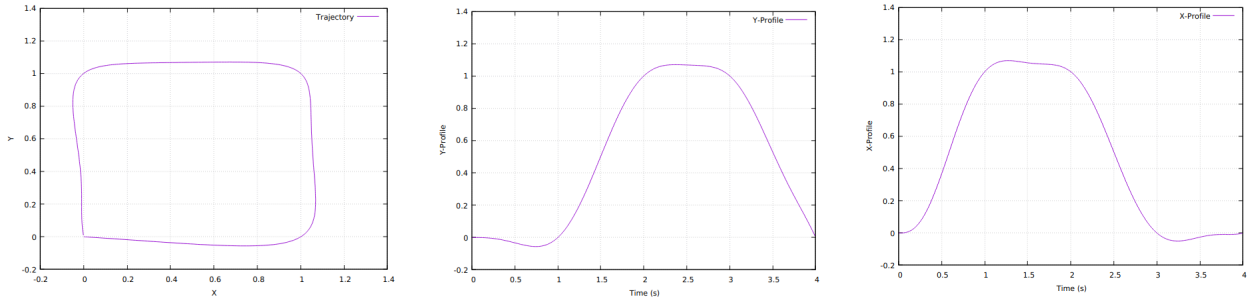


Figure 34: Generated trajectory minimizing velocity; first order derivative

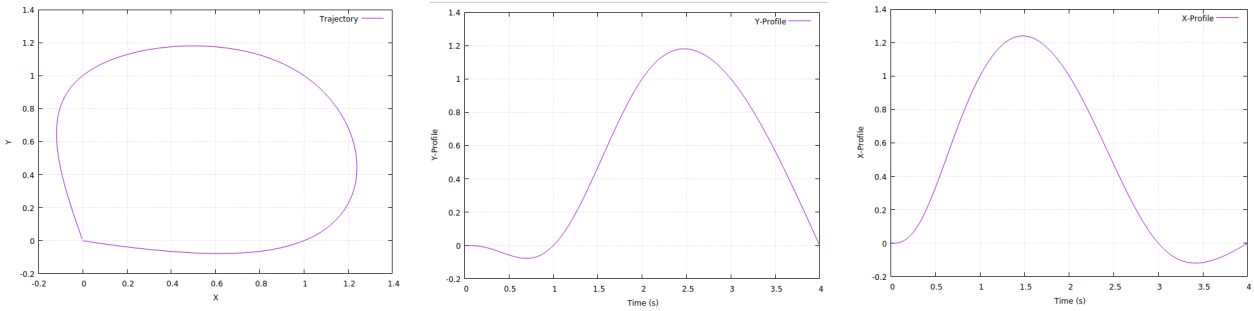


Figure 35: Generated trajectory minimizing acceleration; second order derivative

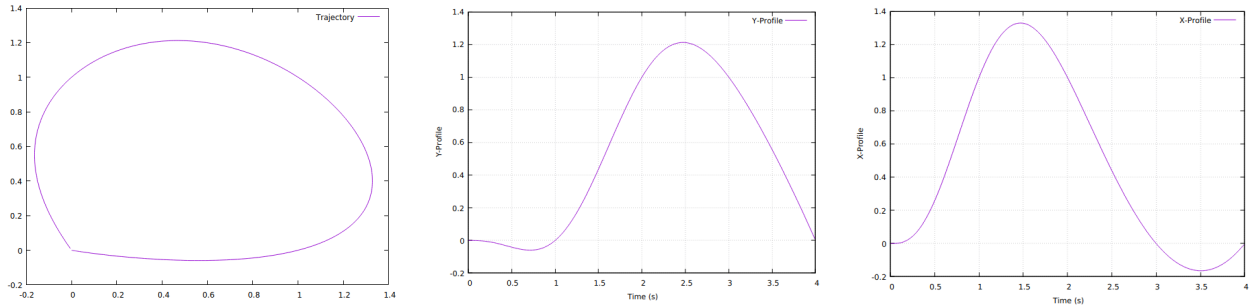


Figure 36: Generated trajectory minimizing jerk; third order derivative

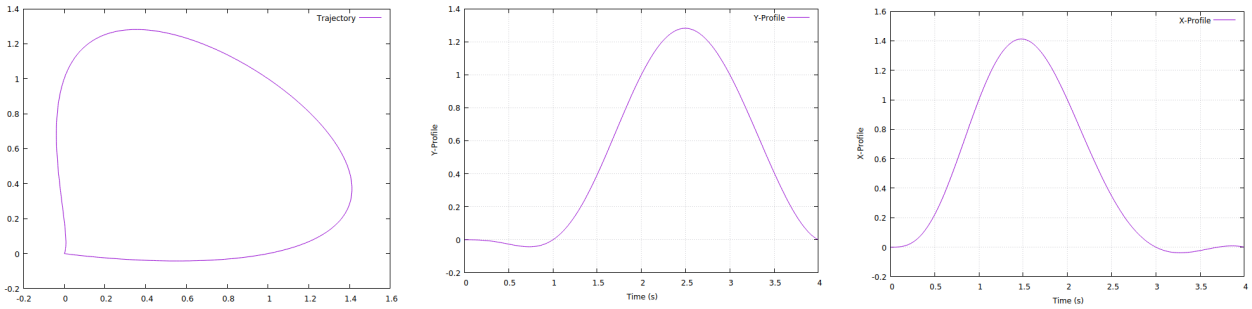


Figure 37: Generated trajectory minimizing snap; fourth order derivative

As seen from the results in Fig. 34, minimizing the velocity created the most accurate square from the path waypoints. Furthermore, upon increasing derivative order, the X and Y axis positions with respect to time form a more accurate bell-shaped curve. The position trajectory, after the first order, shifts further away from the intended square position. As evident in Fig. 37, the X and Y position charts depict the largest peak values amongst the other graphs of minimized physical properties.

In reaching a sweet spot between intended trajectory shape and smoothness, one can discern that Fig. 35 displays the best medium. By minimizing acceleration, the generated trajectory for the quadcopter becomes easier to control for, as changes in rotor speed directly affect the acceleration of the UAV.

4.7 Impact of Varying Arrival Time on Polynomial Smoothing

Another variation in creating the polynomials that form the quadrotor trajectory is the allocation of time allowed in between each node. Lessening the arrival time can induce a trajectory that, for example, takes tighter turns around obstacles. Minimizing the arrival time has practical implications however excessive minimization can push the limits of what the quadrotor is dynamically capable of. On the other hand, providing an excessively high time constraint can allow for the rotor to take a safer path, but this is obviously slower and can exceed the battery hours of quadrotor operation. As a result, there is an efficient medium which works for the specific use case of a UAV.

In this test, the arrival time is varied to be unreasonably short, understandable, and excessively long. An

understandable time of one second in between path waypoints was initialized as the understandable baseline. To alter the baseline for the other configurations, the time vector used was scaled. The unreasonably short arrival time was scaled by dividing by a factor of fifty, thus allocating near a hundredth of a second to reach the next waypoint. The excessively long setting was scaled by multiplying a factor of ten, providing ten seconds to reach the next waypoint generated by A*. In incorporating the various arrival time vectors into the polynomial smoothing function from Fig. 22, the following trajectories were generated on the same square grid.

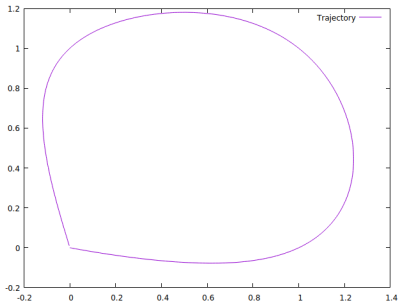


Figure 38a: Reasonable arrival time trajectory

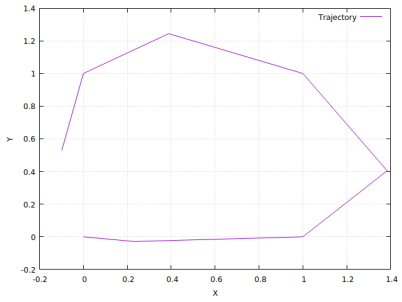


Figure 38b: Unreasonably short arrival time trajectory

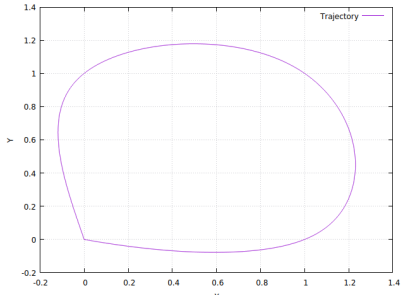


Figure 38c: Excessively long arrival time trajectory

From Fig. 38, the impact of arrival time can be observed. As previously mentioned, in comparison to the appropriate baseline arrival time, the unreasonably short trajectory undercuts the intended shape of the trajectory significantly more than the original. The unreasonably short case, presented in Fig. 38b, also becomes much less smooth as the trajectory attempts to use straight lines to move through the waypoints faster. On the other hand, as depicted in Fig. 38c, excessive time has little impact on the trajectory shape compared to the reasonable baseline in Fig. 38a.

The velocity and acceleration profiles are explored in the subsequent figures for each arrival time instance.

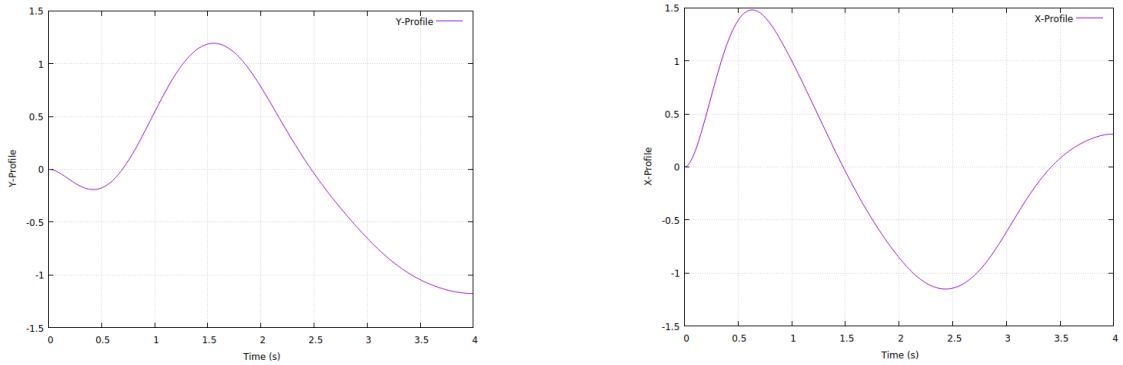
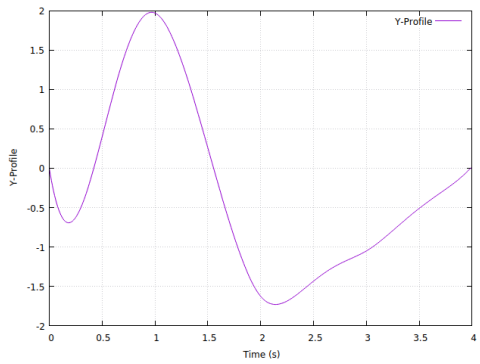


Figure 39: Reasonable arrival time velocity profile



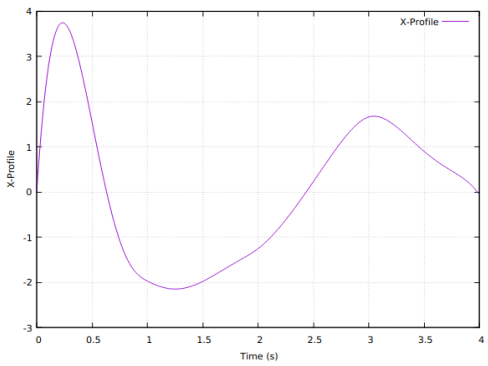
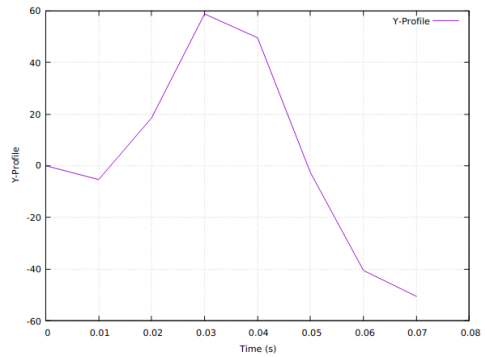


Figure 40: Reasonable arrival time acceleration profile



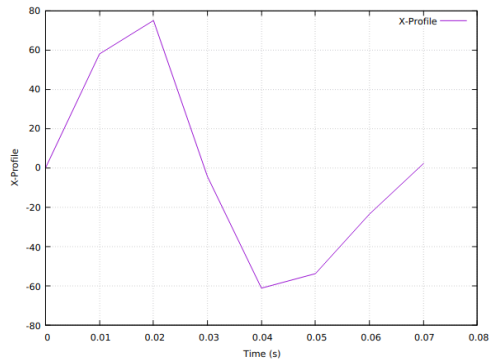
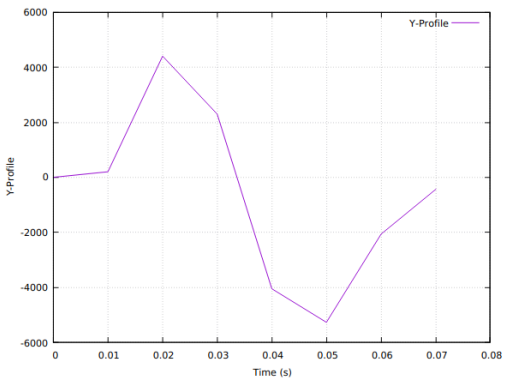


Figure 41: Unreasonably short arrival time velocity profile



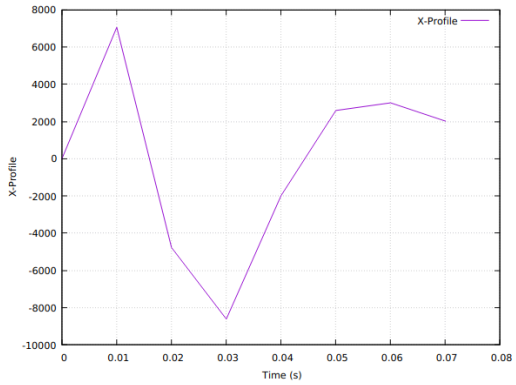


Figure 42: Unreasonably short arrival time acceleration profile

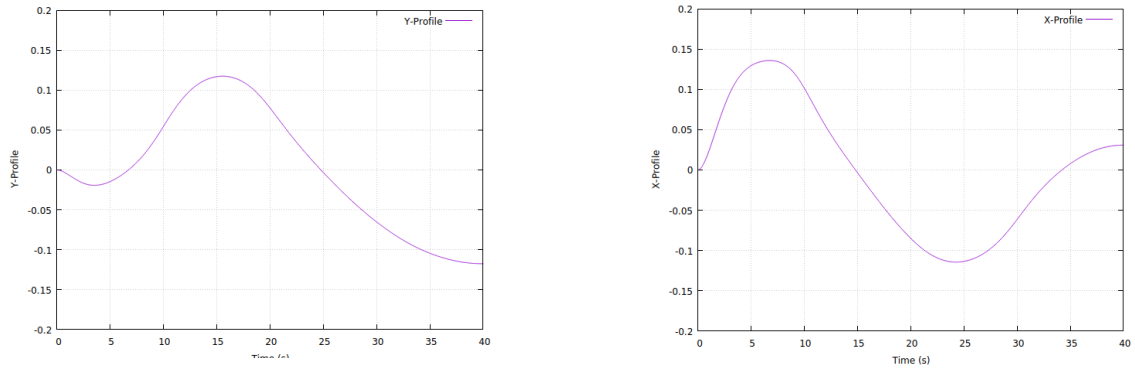


Figure 43: Excessively long arrival time velocity profile

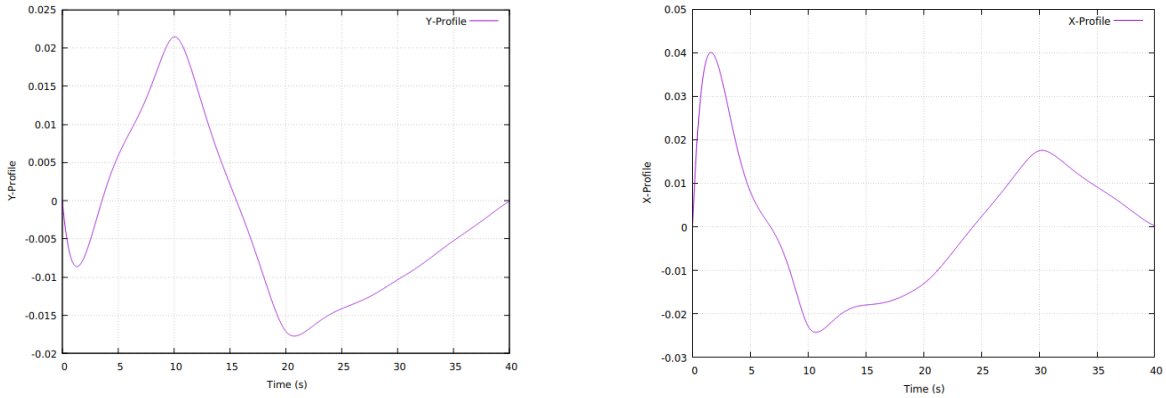


Figure 44: Excessively long arrival time acceleration profile

As evident from Fig. 43 and 44, the shape of the velocity and acceleration in both the X and Y axes are relatively the same as the reasonable time profiles from Fig. 39 and 40. However, the main distinction is the magnitude of the relative profiles. The excessively long arrival time allows the quadrotor to move much slower through the trajectory which translates to significantly lower velocity and acceleration. This is, in part, due to the fact that, as shown in Fig. 38a and 38c, the trajectories between the baseline and the long arrival time scenario are largely unchanged. This means that to cover the same distance in longer time, the allocated acceleration and resulting speed will reduce greatly.

From Fig. 41 and 42, the same pattern as the trajectory for the unreasonably short arrival time is evident- the functions shift more piecewise. The smoothness for the velocity and acceleration profiles decrease with allocated arrival time. Additionally, because of the rapid movement the quadcopter must take to complete the intended trajectory in time, the resulting velocity and acceleration are extremely high. The values of abstract units in Fig. 41 and 42 hint at acceleration values impossible for the quadrotor to realistically induce to pass all waypoints within the arrival time window.

4.8 Impact of Number of Waypoints on Generated Circular Trajectory

For this assessment, a circular path was created using initialized waypoints on a grid. The circular path is

initialized to an arbitrary N number of uniform waypoints. The goal here is to discern the impact the number of waypoints outlining a trajectory have on the generated polynomial trajectory. From a practical sense, if a path with a lower number of generated waypoints creates a polynomial trajectory largely the same as with a higher number- computational power and runtime can be optimized. In essence, it is important to find the grid size needed to represent the environment map. This will translate to an output path from A* with a respectively higher or lower number of nodes. Below, we observe the impact this has on a circular trajectory.

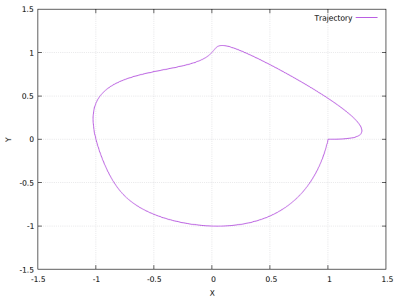


Figure 45a: Generated trajectory with $N = 8$

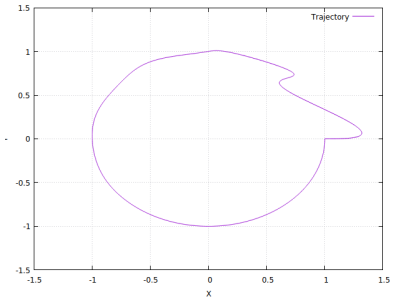


Figure 45b: Generated trajectory with $N = 16$

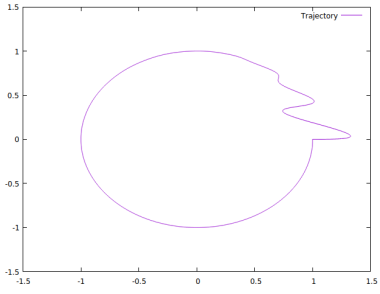


Figure 45c: Generated trajectory with $N = 32$

In the results above, acceleration is minimized and arrival time is held constant to one second per path waypoint. Four tests were performed where N equals 8, 16, and 32. The function for outlining the circle in a uniform manner can be seen in Fig. 22, where the radius of the circle was set to 1 measurement unit. For the trajectory generated in Fig. 45, the starting point is (1,0).

Fig. 45 denotes that increasing N results in a trajectory that aligns more closely with the original intention. In the $N = 8$ case, the trajectory created is a primarily non-uniform circle up to 180°, after which the trajectory denotes the intended unit circle. As the fidelity increases in the $N = 16$ case, the section of the circle that is non-uniform narrows. Approaching the circular trajectory using 32 waypoints shown in Fig. 45c, most of the trajectory is faithful to a unit circle. The initial section of the trajectory visualizes the polynomial fitting function stabilizing to the provided path outline. This feature will persist no matter how large N is.

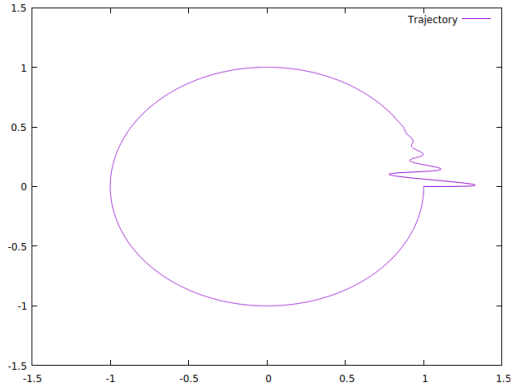


Figure 45d: Generated trajectory with $N = 100$

With Fig. 45d in comparison to Fig. 45a, it is clear that increasing the number of waypoints to define the goal path will increase the fidelity of the generated trajectory.

4.9 Path Planning Trajectory Simulation

In bridging the gap between the pat planning schematic and the control and estimation simulation system developed, the script highlighted in Fig. 23 is utilized. Giving the solver the one second in between waypoints and minimizing acceleration, the position, velocity, and acceleration are sampled at 200 Hz for a provided ten by ten grid. As mentioned prior, the data is then loaded into the top level simulation script to populate the reference trajectory.

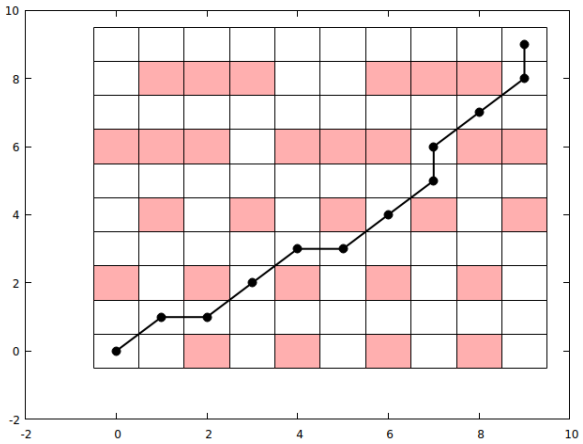


Figure 46: A* path planning algorithm for final test map

With the path generated in Fig. 46, a total of 29 nodes are explored and an optimal path of 12 nodes is shown, producing a path cost of 13.90. Using this path, the polynomial fitting function depicted in Fig. 22 is input to generate a smooth trajectory across 11 seconds. With a sampling rate of 200 Hz, 2200 data points are generated for position, velocity, and acceleration each in separate files. These files define the reference matrices for the control loop in the top level simulation script shown in Fig. 18. The resulting simulation trajectory on the X-Y plane is presented in Fig. 47.

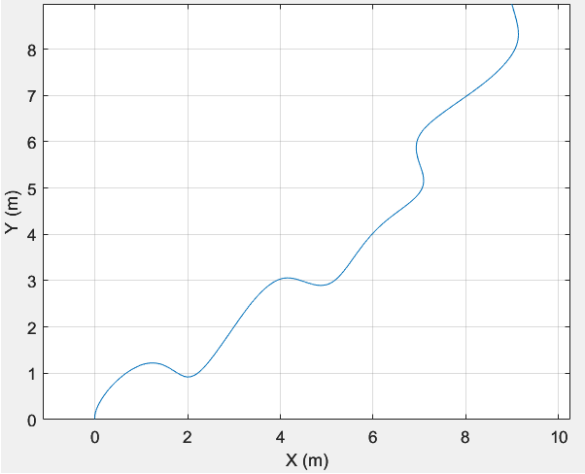


Figure 47: Horizontal position of the quadcopter center of mass

As evident from the figure above, the system developed is able to produce a simulated trajectory utilizing high fidelity quadrotor dynamics to navigate towards a goal based on the A* path planning algorithm. The path in Fig. 46 and the trajectory in Fig. 47 display large similarities. Based on the prior experiments, if the path developed had included more waypoints, the trajectory displayed in Fig. 47 would shadow it a lot more closely. Furthermore, if the sampling rate was increased from 200 Hz, the fidelity of the simulated trajectory would increase.

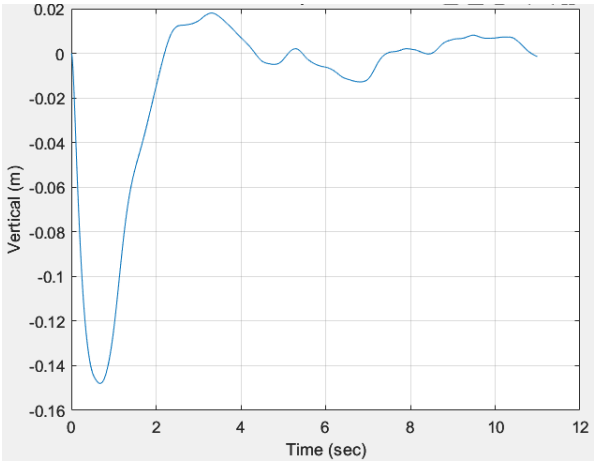


Figure 48: Vertical position of the quadcopter center of mass

Due to the fact that the reference maneuver is conducted on the X-Y plane at an altitude of zero, the quadrotor control capability is able to adjust after briefly falling from rest and then hover at the reference altitude with deviations less than 0.02 m after around 2 seconds. With this, a successful implementation of UAV path planning and simulation is proved to be conducted.

5 Conclusion

A path planning algorithm was developed to generate a reference trajectory for a simulated quadcopter to complete. A closed loop control strategy was developed to simulate the quadcopter. The method was based on a proportional-derivative (PD) controller as part of a feedback loop whereby, using the reference trajectory characteristics, trajectory and attitude controllers attempt to minimize the error between what the quadrotor dynamics predict and the target state. Models for a GNSS antenna, IMU, and camera sensor were integrated into the control loop. The models represented noisy sensor measurements used to estimate the current state. As a result, a simulation of higher fidelity was developed and tested using a trajectory generated with the A* path planning technique and curve fitting. It was found that the control loop is able to use the sensors' estimated state to *accurately* conduct the near optimal reference trajectory to navigate towards the goal location.

References

[1] M. Hamandi, M. Tognon and A. Franchi, "Direct Acceleration Feedback Control of Quadrotor Aerial Vehicles," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, Paris, France, 2020, pp. 5335-5341, doi: 10.1109/ICRA40945.2020.9196557.

[2] “Admissible Heuristic.” Engati, www.engati.com/glossary/admissible-heuristic. Accessed 24 Mar. 2024.

Appendix

The complete coding script for experimentation can be found in the Appendix.

```
Unset
function [uCross] = crossProductEquivalent(u)
% crossProductEquivalent : Outputs the cross-product-equivalent matrix uCross
% such that for arbitrary 3-by-1 vectors u and v,
% cross(u,v) = uCross*v.
%
% INPUTS
%
% u ---------- 3-by-1 vector
%
%
% OUTPUTS
%
% uCross ----- 3-by-3 skew-symmetric cross-product equivalent matrix
%
%+-------------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+===============================================================================+
u1 = u(1,1); % extracted values from row 1, column 1
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
end
```

Figure 1: Cross vector equivalent function

```
Unset
function [R] = rotationMatrix(aHat,phi)
% rotationMatrix : Generates the rotation matrix R corresponding to a rotation
% through an angle phi about the axis defined by the unit
% vector aHat. This is a straightforward implementation of
% Euler's formula for a rotation matrix.
%
% INPUTS
%
% aHat ------- 3-by-1 unit vector constituting the axis of rotation,
% synonmymous with K in the notes.
%
```

```
% phi -------- Angle of rotation, in radians.
%
%
% OUTPUTS
%
% R ---------- 3-by-3 rotation matrix
%
%+--------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+==========================================================================+

function [uCross] = crossProductEquivalent(u)
u1 = u(1,1);
u2 = u(2,1);
u3 = u(3,1);
uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
end

I = [1 0 0; 0 1 0; 0 0 1];
aHatTranspose = aHat.';
R1 = cos(phi)*I;
R2 = (1-cos(phi))*aHat*aHatTranspose;
R3 = sin(phi)*crossProductEquivalent(aHat);
R = R1+R2-R3;
end
```

Figure 2: Rotation matrix development function

```
Unset
function [e] = dcm2euler(R_BW)
% dcm2euler : Converts a direction cosine matrix R_BW to Euler angles phi =
%             e(1), theta = e(2), and psi = e(3) (in radians) for a 3-1-2
%             rotation. If the conversion to Euler angles is singular (not
%             unique), then this function issues an error instead of
%              returning e.
%
% Let the world (W) and body (B) reference frames be initially aligned.  In a
% 3-1-2 order, rotate B away from W by angles psi (yaw, about the body Z
% axis), phi (roll, about the body X axis), and theta (pitch, about the body Y
% axis).  R_BW can then be used to cast a vector expressed in W coordinates as
% a vector in B coordinates: vB = R_BW * vW
%
```

```
% INPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%
% OUTPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3).  By convention, these
%              should be constrained to the following ranges: -pi/2 <= phi <=
%              pi/2, -pi <= theta < pi, -pi <= psi < pi.
%
%+--------------------------------------------------------------------------+
% References: None
%
%
% Author: Aaron Pandian
%+==========================================================================+

phi = asin(R_BW(2,3));
% assert() throws an error if condition is false
assert(sin(phi)~=-pi/2 && sin(phi)~=pi/2,'Conversion is singular.');
theta = atan2(-(R_BW(1,3)),R_BW(3,3));
if theta == pi
   theta = -pi;
end
psi = atan2(-(R_BW(2,1)),R_BW(2,2));
if psi == pi
   psi = -pi;
end
e = [phi; theta; psi]';
end
```

Figure 3: Direction cosine matrix to euler angles function

```
Unset
function [R_BW] = euler2dcm(e)
% euler2dcm : Converts Euler angles phi = e(1), theta = e(2), and psi = e(3)
%             (in radians) into a direction cosine matrix for a 3-1-2 rotation.
%
% Let the world (W) and body (B) reference frames be initially aligned.  In a
% 3-1-2 order, rotate B away from W by angles psi (yaw, about the body Z
% axis), phi (roll, about the body X axis), and theta (pitch, about the body Y
% axis).  R_BW can then be used to cast a vector expressed in W coordinates as
```

```
% a vector in B coordinates: vB = R_BW * vW
%
% INPUTS
%
% e ---------- 3-by-1 vector containing the Euler angles in radians: phi =
%              e(1), theta = e(2), and psi = e(3)
%
% OUTPUTS
%
% R_BW ------- 3-by-3 direction cosine matrix
%
%+-------------------------------------------------------------------------------+
% References: Attitude Transformations. VectorNav. (n.d.).
%https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/
%math-attitudetran
%
% Author: Aaron Pandian
%+===============================================================================+

phi = e(1,1);
theta = e(2,1);
psi = e(3,1);

% Method 1
R1e = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R2e = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R3e = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];

% Method 2
% function [R] = rotationMatrix(aHat,phi)
%
% function [uCross] = crossProductEquivalent(u)
% u1 = u(1,1); % extracted values from row 1, column 1
% u2 = u(2,1);
% u3 = u(3,1);
% uCross = [0 -u3 u2; u3 0 -u1; -u2 u1 0];
% end
%
% I = [1 0 0; 0 1 0; 0 0 1];
% aHatTranspose = transpose(aHat);
% R1 = cos(phi)*I;
% R2 = (1-cos(phi))*aHat*aHatTranspose;
% % or I + crossProductEquivilant(aHat)^2 = a*a^T
% R3 = sin(phi)*crossProductEquivalent(aHat);
% R = R1+R2-R3;
% end
%
% R3e = rotationMatrix([0;0;1],psi);
```

```
% R1e = rotationMatrix([1;0;0],phi);
% R2e = rotationMatrix([0;1;0],theta);
% Using 3-1-2 Rotation
R_BW = R2e*R1e*R3e;
end
```

Figure 4: Euler angles to direction cosine matrix

```
Unset
function [Xdot] = quadOdeFunctionHF(t,X,eaVec,distVec,P)
% quadOdeFunctionHF : Ordinary differential equation function that models
%                     quadrotor dynamics -- high-fidelity version.  For use
%                     with one of Matlab's ODE solvers (e.g., ode45).
%
%
% INPUTS
%
% t ---------- Scalar time input, as required by Matlab's ODE function
%              format.
%
% X ---------- Nx-by-1 quad state, arranged as
%
%              X = [rI',vI',RBI(1,1),RBI(2,1),...,RBI(2,3),RBI(3,3),...
%                   omegaB',omegaVec']'
%
%              rI = 3x1 position vector in I in meters
%              vI = 3x1 velocity vector wrt I and in I, in meters/sec
%             RBI = 3x3 attitude matrix from I to B frame
%          omegaB = 3x1 angular rate vector of body wrt I, expressed in B
%                   in rad/sec
%        omegaVec = 4x1 vector of rotor angular rates, in rad/sec.
%                   omegaVec(i) is the angular rate of the ith rotor.
%
%    eaVec --- 4x1 vector of voltages applied to motors, in volts.  eaVec(i)
%              is the constant voltage setpoint for the ith rotor.
%
%  distVec --- 3x1 vector of constant disturbance forces acting on the quad's
%              center of mass, expressed in Newtons in I.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
```

```
%                 control, as defined in constantsScript.m
%
% OUTPUTS
%
% Xdot ------- Nx-by-1 time derivative of the input vector X
%
%+-----------------------------------------------------------------------------+
% References:
%
%
% Author:
%+=============================================================================+

% Extract quantities from state vector
rI = X(1:3);
vI = X(4:6);
RBI = zeros(3,3);
RBI(:) = X(7:15);
omegaB = X(16:18);
omegaVec = X(19:22);

% Determine forces and torques for each rotor from rotor angular rates.  The
% ith column in FMat is the force vector for the ith rotor, in B.  The ith
% column in NMat is the torque vector for the ith rotor, in B.  Note that
% we negate P.quadParams.omegaRdir because the torque acting on the body is
% in the opposite direction of the angular rate vector for each rotor.
FMat = [zeros(2,4);(P.quadParams.kF.*(omegaVec.^2))'];
NMat = [zeros(2,4);(P.quadParams.kN.*(omegaVec.^2).*(-P.quadParams.omegaRdir)')'];

% Assign some local variables for convenience
mq = P.quadParams.m;
gE = P.constants.g;
Jq = P.quadParams.Jq;
omegaBx = crossProductEquivalent(omegaB);
zI = RBI(3,:)';

% Calculate drag coefficient
epsilon_vI = 1e-3;
da = 0; vIu = [1;0;0];
if(norm(vI) > epsilon_vI)
  vIu = vI/norm(vI);
  fd = abs(zI'*vIu)*norm(vI)^2;
  da = 0.5*P.quadParams.Cd*P.quadParams.Ad*P.constants.rho*fd;
end

% Find derivatives of state elements
rIdot = vI;
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + distVec - da*vIu)/mq;
```

```
RBIdot = -omegaBx*RBI;
NB = sum(NMat,2);
for ii=1:4
  NB = NB + cross(P.quadParams.rotor_loc(:,ii),FMat(:,ii));
end
omegaBdot = inv(Jq)*(NB - omegaBx*Jq*omegaB);
omegaVecdot = (eaVec.*P.quadParams.cm - omegaVec)./P.quadParams.taum;

% Load the output vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];

% Find derivatives of state elements
rIdot = vI;
fd = dot(zI,vIUnitVector)*(vIMag^2);
da = 0.5*Cd*Ad*rho*fd;
fDragVec = vIUnitVector*da;
fDrag = [fDragVec(1) fDragVec(2) fDragVec(3)]';
vIdot = ([0;0;-mq*gE] + RBI'*sum(FMat,2) + fDrag)/mq;
RBIdot = -omegaBx*RBI;
NB = sum(NMat,2);
for ii=1:4
 NB = NB + cross(P.quadParams.rotor_loc(:,ii),FMat(:,ii));
end
omegaBdot = inv(Jq)*(NB - omegaBx*Jq*omegaB);
% Load the output vector
Xdot = [rIdot;vIdot;RBIdot(:);omegaBdot;omegaVecdot];
```

Figure 5: Quadcopter high fidelity ordinary differential equation function

Unset

```
function [Q] = simulateQuadrotorControl(R,S,P)
% simulateQuadrotorControl : Simulates closed-loop control of a quadrotor
%                            aircraft.
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%          tVec = Nx1 vector of uniformly-sampled time offsets from the
%                 initial time, in seconds, with tVec(1) = 0.
%
%        rIstar = Nx3 matrix of desired CM positions in the I frame, in
%                 meters.  rIstar(k,:)' is the 3x1 position at time tk =
%                 tVec(k).
```

```
%
%       vIstar = Nx3 matrix of desired CM velocities with respect to the I
%                frame and expressed in the I frame, in meters/sec.
%                vIstar(k,:)' is the 3x1 velocity at time tk = tVec(k).
%
%       aIstar = Nx3 matrix of desired CM accelerations with respect to the I
%                frame and expressed in the I frame, in meters/sec^2.
%                aIstar(k,:)' is the 3x1 acceleration at time tk =
%                tVec(k).
%
%       xIstar = Nx3 matrix of desired body x-axis direction, expressed as a
%                unit vector in the I frame. xIstar(k,:)' is the 3x1
%                direction at time tk = tVec(k).
%
% S ---------- Structure with the following elements:
%
%  oversampFact = Oversampling factor. Let dtIn = R.tVec(2) - R.tVec(1). Then
%                 the output sample interval will be dtOut =
%                 dtIn/oversampFact. Must satisfy oversampFact >= 1.
%
%       state0 = State of the quad at R.tVec(1) = 0, expressed as a structure
%                with the following elements:
%
%                  r = 3x1 position in the world frame, in meters
%
%                  e = 3x1 vector of Euler angles, in radians, indicating the
%                      attitude
%
%                  v = 3x1 velocity with respect to the world frame and
%                      expressed in the world frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%      distMat = (N-1)x3 matrix of disturbance forces acting on the quad's
%                center of mass, expressed in Newtons in the world frame.
%                distMat(k,:)' is the constant (zero-order-hold) 3x1
%                disturbance vector acting on the quad from R.tVec(k) to
%                R.tVec(k+1).
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
```

```
%  sensorParams = Structure containing sensor parameters, as defined in
%                 sensorParamsScript.m
%
%
% OUTPUTS
%
% Q ---------- Structure with the following elements:
%
%          tVec = Mx1 vector of output sample time points, in seconds, where
%                 Q.tVec(1) = R.tVec(1), Q.tVec(M) = R.tVec(N), and M =
%                 (N-1)*oversampFact + 1.
%
%         state = State of the quad at times in tVec, expressed as a
%                 structure with the following elements:
%
%                rMat = Mx3 matrix composed such that rMat(k,:)' is the 3x1
%                       position at tVec(k) in the I frame, in meters.
%
%                eMat = Mx3 matrix composed such that eMat(k,:)' is the 3x1
%                       vector of Euler angles at tVec(k), in radians,
%                       indicating the attitude.
%
%                vMat = Mx3 matrix composed such that vMat(k,:)' is the 3x1
%                       velocity at tVec(k) with respect to the I frame
%                       and expressed in the I frame, in meters per
%                       second.
%
%           omegaBMat = Mx3 matrix composed such that omegaBMat(k,:)' is the
%                       3x1 angular rate vector expressed in the body frame in
%                       radians, that applies at tVec(k).
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

N = length(R.tVec);
dtIn = R.tVec(2) - R.tVec(1);
dtOut = dtIn/S.oversampFact;
RBIk = euler2dcm(S.state0.e);

% Initial angular rates in rad/s, initialize here as per lab document
S.state0.omegaVec = [0 0 0 0]';

Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;S.state0.omegaVec];
Pa.quadParams = P.quadParams;
```

```
Pa.constants = P.constants;
Pa.sensorParams = P.sensorParams;

XMat = []; tVec = [];

% Initialize Sk values
Sk.statek.RBI = euler2dcm(S.state0.e); % Initial e to RBI for function input
Sk.statek.rI = S.state0.r;
Sk.statek.vI = S.state0.v;
Sk.statek.omegaB = S.state0.omegaB;
Sk.statek.omegaVec = S.state0.omegaVec;

for kk=1:N-1
  tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
  distVeck = S.distMat(kk,:)';
  % Get Rk values, P structure is constant
  Rk.tVec = R.tVec(kk);
  Rk.rIstark = R.rIstar(kk,:)';
  Rk.vIstark = R.vIstar(kk,:)';
  Rk.aIstark = R.aIstar(kk,:)';
  Rk.xIstark = R.xIstar(kk,:)';

  [Fk, zIstark] = trajectoryController(Rk,Sk,P); % Where S structure input needs
to be updated after initial state

  Rk.zIstark = zIstark;

  [NBk] = attitudeController(Rk,Sk,P);
  [eak] = voltageConverter(Fk,NBk,P);
  eaVeck = eak;

  [tVeck,XMatk] = ...
      ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,Pa),tspan,Xk); % Think
through omegaVec input between two simulators

  % Update Sk values after Quadrotor Function
  XstateNow = XMatk(end,:);
  Sk.statek.rI = XstateNow(1:3)';
  Sk.statek.vI = XstateNow(4:6)';
  Sk.statek.omegaB = XstateNow(16:18)';
  Sk.statek.RBI = [XstateNow(7) XstateNow(10) XstateNow(13); % 3 x 3
                   XstateNow(8) XstateNow(11) XstateNow(14);
                   XstateNow(9) XstateNow(12) XstateNow(15)];
  % Can also add omegaVec here but not needed since already stored in XMat
  % and not needed elsewhere, since its not part of state.

  if(length(tspan) == 2)
    tVec = [tVec; tVeck(1)];
```

```
    XMat = [XMat; XMatk(1,:)];
  else
    tVec = [tVec; tVeck(1:end-1)];
    XMat = [XMat; XMatk(1:end-1,:)];
  end
  Xk = XMatk(end,:)';
  if(mod(kk,10) == 0)
    RBIk(:) = Xk(7:15);
    [UR,SR,VR]=svd(RBIk);
    RBIk = UR*VR'; Xk(7:15) = RBIk(:);
  end
end
XMat = [XMat;XMatk(end,:)];
tVec = [tVec;tVeck(end,:)];

M = length(tVec);
Q.tVec = tVec;
Q.state.rMat = XMat(:,1:3);
Q.state.vMat = XMat(:,4:6);
Q.state.omegaBMat = XMat(:,16:18);
% Q.state.omegaVec = XMat(:,19:22); % OmegaVec not included in state but
% recorded in QuadODE, so can optionally include in output.
Q.state.eMat = zeros(M,3);
RBI = zeros(3,3);
for mm=1:M
  RBI(:) = XMat(mm,7:15);
  Q.state.eMat(mm,:) = dcm2euler(RBI)';
end
```

Figure 6: Quadcopter control simulation function

Unset

```
function [Q] = simulateQuadrotorEstimationAndControl(R,S,P)
% simulateQuadrotorEstimationAndControl : Simulates closed-loop estimation and
%                                         control of a quadrotor aircraft.
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%          tVec = Nx1 vector of uniformly-sampled time offsets from the
%                 initial time, in seconds, with tVec(1) = 0.
%
%        rIstar = Nx3 matrix of desired CM positions in the I frame, in
%                 meters.  rIstar(k,:)' is the 3x1 position at time tk =
```

```
%                 tVec(k).
%
%       vIstar = Nx3 matrix of desired CM velocities with respect to the I
%                frame and expressed in the I frame, in meters/sec.
%                vIstar(k,:)' is the 3x1 velocity at time tk = tVec(k).
%
%       aIstar = Nx3 matrix of desired CM accelerations with respect to the I
%                frame and expressed in the I frame, in meters/sec^2.
%                aIstar(k,:)' is the 3x1 acceleration at time tk =
%                tVec(k).
%
%       xIstar = Nx3 matrix of desired body x-axis direction, expressed as a
%                unit vector in the I frame. xIstar(k,:)' is the 3x1
%                direction at time tk = tVec(k).
%
% S ---------- Structure with the following elements:
%
%  oversampFact = Oversampling factor. Let dtIn = R.tVec(2) - R.tVec(1). Then
%                the output sample interval will be dtOut =
%                dtIn/oversampFact. Must satisfy oversampFact >= 1.
%
%       state0 = State of the quad at R.tVec(1) = 0, expressed as a structure
%                with the following elements:
%
%                  r = 3x1 position in the world frame, in meters
%
%                  e = 3x1 vector of Euler angles, in radians, indicating the
%                      attitude
%
%                  v = 3x1 velocity with respect to the world frame and
%                      expressed in the world frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%      distMat = (N-1)x3 matrix of disturbance forces acting on the quad's
%                center of mass, expressed in Newtons in the I frame.
%                distMat(k,:)' is the constant (zero-order-hold) 3x1
%                disturbance vector acting on the quad from R.tVec(k) to
%                R.tVec(k+1).
%
%       rXIMat = Nf-by-3 matrix of coordinates of visual features in the
%                simulation environment, expressed in meters in the I
%                frame. rXIMat(i,:)' is the 3x1 vector of coordinates of
%                the ith feature.
%
% P ---------- Structure with the following elements:
%
```

```
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%  sensorParams = Structure containing sensor parameters, as defined in
%                 sensorParamsScript.m
%
%
% OUTPUTS
%
% Q ---------- Structure with the following elements:
%
%          tVec = Mx1 vector of output sample time points, in seconds, where
%                 Q.tVec(1) = R.tVec(1), Q.tVec(M) = R.tVec(N), and M =
%                 (N-1)*oversampFact + 1.
%
%         state = State of the quad at times in tVec, expressed as a
%                 structure with the following elements:
%
%                rMat = Mx3 matrix composed such that rMat(k,:)' is the 3x1
%                       position at tVec(k) in the I frame, in meters.
%
%                eMat = Mx3 matrix composed such that eMat(k,:)' is the 3x1
%                       vector of Euler angles at tVec(k), in radians,
%                       indicating the attitude.
%
%                vMat = Mx3 matrix composed such that vMat(k,:)' is the 3x1
%                       velocity at tVec(k) with respect to the I frame
%                       and expressed in the I frame, in meters per
%                       second.
%
%           omegaBMat = Mx3 matrix composed such that omegaBMat(k,:)' is the
%                       3x1 angular rate vector expressed in the body frame in
%                       radians, that applies at tVec(k).
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+


N = length(R.tVec);
dtIn = R.tVec(2) - R.tVec(1);
dtOut = dtIn/S.oversampFact;
```

```
RBIk = euler2dcm(S.state0.e);
omegaVec0 = zeros(4,1);
Xk = [S.state0.r;S.state0.v;RBIk(:);S.state0.omegaB;omegaVec0];
Xdotk = zeros(length(Xk),1);
statek.RBI = zeros(3,3);
[Nf,~] = size(S.rXIMat);
Se.rXIMat = S.rXIMat;
Se.delt = dtIn;
XMat = []; tVec = [];

for kk=1:N-1
  % Simulate measurements
  statek.rI = Xk(1:3);
  statek.RBI(:) = Xk(7:15);
  statek.vI = Xk(4:6);
  statek.omegaB = Xk(16:18);
  statek.aI = Xdotk(4:6);
  statek.omegaBdot = Xdotk(16:18);
  Sm.statek = statek;
  % Simulate measurements
  M.tk=dtIn*(kk-1);
  [M.rpGtilde,M.rbGtilde] = gnssMeasSimulator(Sm,P);
  M.rxMat = [];
  for ii=1:Nf
    rx = hdCameraSimulator(S.rXIMat(ii,:)',Sm,P);
    if(isempty(rx))
      M.rxMat(ii,:) = [NaN,NaN];
    else
      M.rxMat(ii,:) = rx';
    end
  end
  [M.ftildeB,M.omegaBtilde] = imuSimulator(Sm,P);
  % Call estimator
  E = stateEstimatorUKF(Se,M,P);
  if(~isempty(E.statek))
    % Call trajectory and attitude controllers
    Rtc.rIstark = R.rIstar(kk,:)';
    Rtc.vIstark = R.vIstar(kk,:)';
    Rtc.aIstark = R.aIstar(kk,:)';
    Rac.xIstark = R.xIstar(kk,:)';
    distVeck = S.distMat(kk,:)';
    Sc.statek = E.statek;
    [Fk,Rac.zIstark] = trajectoryController(Rtc,Sc,P);
    NBk = attitudeController(Rac,Sc,P);
    % Convert commanded Fk and NBk to commanded voltages
    eaVeck = voltageConverter(Fk,NBk,P);
  else
    % Apply no control if state estimator's output is empty.  Set distVeck to
```

```
    % apply a normal force in the vertical direction that exactly offsets the
    % acceleration due to gravity.
    eaVeck = zeros(4,1);
    distVeck = [0;0;P.quadParams.m*P.constants.g];
  end
  tspan = [R.tVec(kk):dtOut:R.tVec(kk+1)]';
  [tVeck,XMatk] = ...
      ode45(@(t,X) quadOdeFunctionHF(t,X,eaVeck,distVeck,P),tspan,Xk);
  if(length(tspan) == 2)
    % Deal with S.oversampFact = 1 case
    tVec = [tVec; tVeck(1)];
    XMat = [XMat; XMatk(1,:)];
  else
    tVec = [tVec; tVeck(1:end-1)];
    XMat = [XMat; XMatk(1:end-1,:)];
  end
  Xk = XMatk(end,:)';
  Xdotk = quadOdeFunctionHF(tVeck(end),Xk,eaVeck,distVeck,P);
  % Ensure that RBI remains orthogonal
  if(mod(kk,100) == 0)
   RBIk(:) = Xk(7:15);
   [UR,~,VR]=svd(RBIk);
   RBIk = UR*VR'; Xk(7:15) = RBIk(:);
  end
end
XMat = [XMat;XMatk(end,:)];
tVec = [tVec;tVeck(end,:)];

M = length(tVec);
Q.tVec = tVec;
Q.state.rMat = XMat(:,1:3);
Q.state.vMat = XMat(:,4:6);
Q.state.omegaBMat = XMat(:,16:18);
Q.state.eMat = zeros(M,3);
RBI = zeros(3,3);
for mm=1:M
  RBI(:) = XMat(mm,7:15);
  Q.state.eMat(mm,:) = dcm2euler(RBI)';
end
```

Figure 7: Quadcopter control and estimation simulation function

```
Unset
% Top-level script for calling simulateQuadrotorControl or
% simulateQuadrotorEstimationAndControl
```

```
% 'clear all' is needed to clear out persistent variables from run to run
clear all; clc;
% Seed Matlab's random number: this allows you to simulate with the same noise
% every time (by setting a nonnegative integer seed as argument to rng) or
% simulate with a different noise realization every time (by setting
% 'shuffle' as argument to rng).
rng('shuffle');
%rng(1234);
% Assert this flag to call the full estimation and control simulator;
% otherwise, only the control simulator is called
estimationFlag = 1;
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;
% Populate reference trajectory
R.tVec = tVec;
R.rIstar = [r*cos(n*tVec),r*sin(n*tVec),ones(N,1)];
R.vIstar = [-r*n*sin(n*tVec),r*n*cos(n*tVec),zeros(N,1)];
R.aIstar = [-r*n*n*cos(n*tVec),-r*n*n*sin(n*tVec),zeros(N,1)];
% The desired xI points toward the origin. The code below also normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);
% Matrix of disturbance forces acting on the body, in Newtons, expressed in I
S.distMat = 0*randn(N-1,3);
% Initial position in m
S.state0.r = [r 0 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 pi]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
% Initial angular rate of body with respect to I, expressed in B, in rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 2;
% Feature locations in the I frame
%S.rXIMat = [0,0,0; 0,0,0.7];
S.rXIMat = [];
% Quadrotor parameters and constants
quadParamsScript;
```

```
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;

if(estimationFlag)
  Q = simulateQuadrotorEstimationAndControl(R,S,P);
else
  Q = simulateQuadrotorControl(R,S,P);
end

S2.tVec = Q.tVec;
S2.rMat = Q.state.rMat;
S2.eMat = Q.state.eMat;
S2.plotFrequency = 20;
S2.makeGifFlag = false;
S2.gifFileName = 'testGif.gif';
S2.bounds=2.5*[-1 1 -1 1 -0.1 1];
visualizeQuad(S2);

figure(2);clf;
plot(Q.tVec,Q.state.rMat(:,3)); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('Vertical position of CM');

figure(3);clf;
psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
meanPsiErrorInt = round(mean(psiError)/2/pi);
plot(Q.tVec,psiError - meanPsiErrorInt*2*pi);
grid on;
xlabel('Time (sec)');
ylabel('\Delta \psi (rad)');
title('Yaw angle error');

figure(5);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2));
axis equal; grid on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
```

Figure 8: Top-level simulation script

Unset

```
function P = visualizeQuad(S)
% visualizeQuad : Takes in an input structure S and visualizes the resulting
%                 3D motion in approximately real-time.  Outputs the data
%                 used to form the plot.
%
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%           rMat = 3xM matrix of quad positions, in meters
%
%           eMat = 3xM matrix of quad attitudes, in radians
%
%           tVec = Mx1 vector of times corresponding to each measurement in
%                  xevwMat
%
%  plotFrequency = The scalar number of frames of the plot per each second of
%                  input data.  Expressed in Hz.
%
%         bounds = 6x1, the 3d axis size vector
%
%    makeGifFlag = Boolean (if true, export the current plot to a .gif)
%
%    gifFileName = A string with the file name of the .gif if one is to be
%                  created.  Make sure to include the .gif exentsion.
%
%
% OUTPUTS
%
% P ---------- Structure with the following elements:
%
%          tPlot = Nx1 vector of time points used in the plot, sampled based
%                  on the frequency of plotFrequency
%
%          rPlot = 3xN vector of positions used to generate the plot, in
%                  meters.
%
%          ePlot = 3xN vector of attitudes used to generate the plot, in
%                  radians.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:  Nick Montalbano
%+==============================================================================+
 % Important params
```

```
figureNumber = 42; figure(figureNumber); clf;
fcounter = 0; %frame counter for gif maker
m = length(S.tVec);
% UT colors
burntOrangeUT = [191, 87, 0]/255;
darkGrayUT = [51, 63, 72]/255;
% Parameters for the rotors
rotorLocations=[0.105 0.105 -0.105 -0.105
   0.105 -0.105 0.105 -0.105
   0 0 0 0];
r_rotor = .062;
% Determines the location of the corners of the body box in the body frame,
% in meters
bpts=[ 120  120 -120 -120  120  120 -120 -120
   28  -28   28  -28   28  -28   28  -28
   20   20   20   20   -30   -30   -30   -30 ]*1e-3;
% Rectangles representing each side of the body box
b1 = [bpts(:,1) bpts(:,5) bpts(:,6) bpts(:,2) ];
b2 = [bpts(:,1) bpts(:,5) bpts(:,7) bpts(:,3) ];
b3 = [bpts(:,3) bpts(:,7) bpts(:,8) bpts(:,4) ];
b4 = [bpts(:,1) bpts(:,3) bpts(:,4) bpts(:,2) ];
b5 = [bpts(:,5) bpts(:,7) bpts(:,8) bpts(:,6) ];
b6 = [bpts(:,2) bpts(:,6) bpts(:,8) bpts(:,4) ];
% Create a circle for each rotor
t_circ=linspace(0,2*pi,20);
circpts=zeros(3,20);
for i=1:20
   circpts(:,i)=r_rotor*[cos(t_circ(i));sin(t_circ(i));0];
end
% Plot single epoch if m==1
if m==1
   figure(figureNumber);

   % Extract params
   RIB = euler2dcm(S.eMat(1:3))';
   r = S.rMat(1:3);

   % Translate, rotate, and plot the rotors
   hold on
   view(3)
   rotor1_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,1)*ones(1,20));
   rotor1plot = plot3(rotor1_circle(1,:), rotor1_circle(2,:),
rotor1_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor2_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,2)*ones(1,20));
   rotor2plot = plot3(rotor2_circle(1,:), rotor2_circle(2,:),
rotor2_circle(3,:),...
```

```
       'color',darkGrayUT);
   hold on
   rotor3_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,3)*ones(1,20));
   rotor3plot = plot3(rotor3_circle(1,:), rotor3_circle(2,:),
rotor3_circle(3,:),...
       'color',darkGrayUT);
   hold on
   rotor4_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,4)*ones(1,20));
   rotor4plot = plot3(rotor4_circle(1,:), rotor4_circle(2,:),
rotor4_circle(3,:),...
       'color',darkGrayUT);

   % Plot the body
   b1r=r*ones(1,4)+RIB*b1; b2r=r*ones(1,4)+RIB*b2; b3r=r*ones(1,4)+RIB*b3;
   b4r=r*ones(1,4)+RIB*b4; b5r=r*ones(1,4)+RIB*b5; b6r=r*ones(1,4)+RIB*b6;
   X = [b1r(1,:)' b2r(1,:)' b3r(1,:)' b4r(1,:)' b5r(1,:)' b6r(1,:)'];
   Y = [b1r(2,:)' b2r(2,:)' b3r(2,:)' b4r(2,:)' b5r(2,:)' b6r(2,:)'];
   Z = [b1r(3,:)' b2r(3,:)' b3r(3,:)' b4r(3,:)' b5r(3,:)' b6r(3,:)'];
   hold on
   bodyplot=patch(X,Y,Z,[.5 .5 .5]);

   % Plot the body axes
   bodyX=0.5*RIB*[1;0;0]; bodyY=0.5*RIB*[0;1;0]; bodyZ=0.5*RIB*[0;0;1];
   hold on
   axis1 = quiver3(r(1),r(2),r(3),bodyX(1),bodyX(2),bodyX(3),'red');
   hold on
   axis2 = quiver3(r(1),r(2),r(3),bodyY(1),bodyY(2),bodyY(3),'blue');
   hold on
   axis3 = quiver3(r(1),r(2),r(3),bodyZ(1),bodyZ(2),bodyZ(3),'green');
   axis(S.bounds)
   xlabel('X')
   ylabel('Y')
   zlabel('Z')
   grid on

   P.tPlot = S.tVec;
   P.rPlot = S.rMat;
   P.ePlot = S.eMat;

elseif m>1 % Interpolation must be used to smooth timing

   % Create time vectors
   tf = 1/S.plotFrequency;
   tmax = S.tVec(m); tmin = S.tVec(1);
   tPlot = tmin:tf:tmax;
   tPlotLen = length(tPlot);

   % Interpolate to regularize times
```

```
   [t2unique, indUnique] = unique(S.tVec);
   rPlot = (interp1(t2unique, S.rMat(indUnique,:), tPlot))';
   ePlot = (interp1(t2unique, S.eMat(indUnique,:), tPlot))';

   figure(figureNumber);

   % Iterate through points
   for i=1:tPlotLen

       % Start timer
       tic

       % Extract data
       RIB = euler2dcm(ePlot(1:3,i))';
       r = rPlot(1:3,i);

       % Translate, rotate, and plot the rotors
       hold on
       view(3)

rotor1_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,1)*ones(1,20));
       rotor1plot = plot3(rotor1_circle(1,:), rotor1_circle(2,:),...
           rotor1_circle(3,:), 'color', darkGrayUT);
       hold on

rotor2_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,2)*ones(1,20));
       rotor2plot = plot3(rotor2_circle(1,:), rotor2_circle(2,:),...
           rotor2_circle(3,:), 'color', darkGrayUT);
       hold on

rotor3_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,3)*ones(1,20));
       rotor3plot = plot3(rotor3_circle(1,:), rotor3_circle(2,:),...
           rotor3_circle(3,:), 'color', darkGrayUT);
       hold on

rotor4_circle=r*ones(1,20)+RIB*(circpts(:,:)+rotorLocations(:,4)*ones(1,20));
       rotor4plot = plot3(rotor4_circle(1,:), rotor4_circle(2,:),...
           rotor4_circle(3,:), 'color', darkGrayUT);

       % Translate, rotate, and plot the body
       b1r=r*ones(1,4)+RIB*b1; b2r=r*ones(1,4)+RIB*b2; b3r=r*ones(1,4)+RIB*b3;
       b4r=r*ones(1,4)+RIB*b4; b5r=r*ones(1,4)+RIB*b5; b6r=r*ones(1,4)+RIB*b6;
       X = [b1r(1,:)' b2r(1,:)' b3r(1,:)' b4r(1,:)' b5r(1,:)' b6r(1,:)'];
       Y = [b1r(2,:)' b2r(2,:)' b3r(2,:)' b4r(2,:)' b5r(2,:)' b6r(2,:)'];
       Z = [b1r(3,:)' b2r(3,:)' b3r(3,:)' b4r(3,:)' b5r(3,:)' b6r(3,:)'];
       hold on
       bodyplot=patch(X,Y,Z,[.5 .5 .5]);

       % Translate, rotate, and plot body axes
```

```
    bodyX=0.5*RIB*[1;0;0]; bodyY=0.5*RIB*[0;1;0]; bodyZ=0.5*RIB*[0;0;1];
    hold on
    axis1 = quiver3(r(1),r(2),r(3),bodyX(1),bodyX(2),bodyX(3),'red');
    hold on
    axis2 = quiver3(r(1),r(2),r(3),bodyY(1),bodyY(2),bodyY(3),'blue');
    hold on
    axis3 = quiver3(r(1),r(2),r(3),bodyZ(1),bodyZ(2),bodyZ(3),'green');
    % Fix up plot style
    axis(S.bounds)
    xlabel('X')
    ylabel('Y')
    zlabel('Z')
    grid on

    tP=toc;
    % Pause to stay near-real-time
    pause(max(0.001,tf-tP))

    % Gif stuff
    if S.makeGifFlag
        fcounter=fcounter+1;
        frame=getframe(figureNumber);
        im=frame2im(frame);
        [imind,cm]=rgb2ind(im,256);
        if fcounter==1
            imwrite(imind,cm,S.gifFileName,'gif','Loopcount',inf,...
                'DelayTime',tf);
        else
            imwrite(imind,cm,S.gifFileName,'gif','WriteMode','append',...
                'DelayTime',tf);
        end
    end

    % Clear plot before next iteration, unless at final time step
    if i<tPlotLen
        delete(rotor1plot)
        delete(rotor2plot)
        delete(rotor3plot)
        delete(rotor4plot)
        delete(bodyplot)
        delete(axis1)
        delete(axis2)
        delete(axis3)
    end
end

P.tPlot = tPlot;
P.ePlot = ePlot;
```

```
   P.rPlot = rPlot;
end
end
```

Figure 9: Simulation visualization function

```
Unset
function [NBk] = attitudeController(R,S,P)
% attitudeController : Controls quadcopter toward a reference attitude
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%       zIstark = 3x1 desired body z-axis direction at time tk, expressed as a
%                 unit vector in the I frame.
%
%       xIstark = 3x1 desired body x-axis direction, expressed as a
%                 unit vector in the I frame.
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude
%
%                  vI = 3x1 velocity with respect to the I frame and
%                       expressed in the I frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
```

```
%
% OUTPUTS
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at time tk, in
%              N-m.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
J = P.quadParams.Jq;
RBI = S.statek.RBI;
wB = S.statek.omegaB;
% Small angle assumption
eE_dot = wB;
zIstark = R.zIstark;
xIstark = R.xIstark;
K = [1 0 0; 0 1 0; 0 0 1]; % Parameters to tune
Kd = [0.3 0 0; 0 0.3 0; 0 0 0.3];
b = cross(zIstark, xIstark)/(norm(cross(zIstark, xIstark))); % 3x1
a = cross(b,zIstark); % 3x1
RBIstark = [a, b, zIstark]'; % 3x3
RE = RBIstark*(RBI'); % Double check if element by element multiplication is
needed here
eE = [RE(2,3)-RE(3,2), RE(3,1)-RE(1,3), RE(1,2)-RE(2,1)]'; % 3x1
NBk = K*eE - Kd*eE_dot + crossProductEquivalent(wB)*J*wB;
```

Figure 10: Attitude controller function

Unset

```
function [Fk,zIstark] = trajectoryController(R,S,P)
% trajectoryController : Controls quadcopter toward a reference trajectory.
%
%
% INPUTS
%
% R ---------- Structure with the following elements:
%
%       rIstark = 3x1 vector of desired CM position at time tk in the I frame,
%                 in meters.
%
%       vIstark = 3x1 vector of desired CM velocity at time tk with respect to
%                 the I frame and expressed in the I frame, in meters/sec.
```

```
%
%       aIstark = 3x1 vector of desired CM acceleration at time tk with
%                 respect to the I frame and expressed in the I frame, in
%                 meters/sec^2.
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude
%
%                  vI = 3x1 velocity with respect to the I frame and
%                       expressed in the I frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% Fk --------- Commanded total thrust at time tk, in Newtons.
%
% zIstark ---- Desired 3x1 body z axis expressed in I frame at time tk.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+
m = P.quadParams.m;
g = P.constants.g;
rIstark = R.rIstark;
vIstark = R.vIstark;
aIstark = R.aIstark;
rI = S.statek.rI;
```

```
RBI = S.statek.RBI;
vI = S.statek.vI;
er = rIstark - rI;
er_dot = vIstark - vI;
K = [4 0 0; 0 4 0; 0 0 4]; % Parameters to tune
Kd = [1.5 0 0; 0 1.5 0; 0 0 1.5]; % Both 3x3
FIstark = K*er + Kd*er_dot + [0; 0; m*g] + m*aIstark;
zIstark = FIstark./norm(FIstark);
e3 = [0 0 1]';
zI = (RBI')*e3;
Fk = (FIstark')*zI;
```

Figure 11: Trajectory controller function

Unset

```
function [eak] = voltageConverter(Fk,NBk,P)
% voltageConverter : Generates output voltages appropriate for desired
%                    torque and thrust.
%
%
% INPUTS
%
% Fk --------- Commanded total thrust at time tk, in Newtons.
%
% NBk -------- Commanded 3x1 torque expressed in the body frame at time tk, in
%              N-m.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%
% OUTPUTS
%
% eak -------- Commanded 4x1 voltage vector to be applied at time tk, in
%              volts. eak(i) is the voltage for the ith motor.
%
%+------------------------------------------------------------------------------+
% References:
%
%
```

```
% Author:
%+==========================================================================+
locations = P.quadParams.rotor_loc;
kF = P.quadParams.kF(1);
kN = P.quadParams.kN(1);
kT = kN/kF;
Cm = P.quadParams.cm(1);
eaMax = P.quadParams.eamax;
beta = 0.9; % constant
alpha = 1; % can reduce
% Finding max force value
wmax = Cm*eaMax;
Fmax = kF*(wmax^2);
FmaxTotal = Fmax*4*beta;
extraVecOne = [FmaxTotal, Fk];
G = [1, 1, 1, 1;
    locations(2,1), locations(2,2), locations(2,3), locations(2,4);
    -locations(1,1), -locations(1,2), -locations(1,3), -locations(1,4);
    -kT, kT, -kT, kT];
% Expressed by min force and torque vector
extraVec = [min(extraVecOne), alpha*NBk(1), alpha*NBk(2), alpha*NBk(3)]';
Fvec = (G^(-1))*extraVec;
% changing alpha to satisfy Fi <= Fmax
while max(all(Fvec > Fmax)) % if any element is over Fmax return a 1 to indicate
true, if all are false, max is a 0 or false.
   alpha = alpha - 0.05;
   extraVec = [min(extraVecOne), alpha*NBk(1), alpha*NBk(2), alpha*NBk(3)]';
   Fvec = (G^(-1))*extraVec; % recalculate Fvec to check again, if passes, this is
new value
end
% Check if motor force values are below zero and fix
for F = 1:4
   if Fvec(F) < 0
       Fvec(F) = 0;
   end
end
omegaVec = ((1/kF)*Fvec).^(0.5);
eak = (1/Cm)*omegaVec;
```

Figure 12: Controller to voltage converter function

```
Unset
function [rpGtilde,rbGtilde] = gnssMeasSimulator(S,P)
% gnssMeasSimulator : Simulates GNSS measurements for quad.
%
```

```
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
%
% OUTPUTS
%
% rpGtilde --- 3x1 GNSS-measured position of the quad's primary GNSS antenna,
%              in ECEF coordinates relative to the reference antenna, in
%              meters.
%
% rbGtilde --- 3x1 GNSS-measured position of secondary GNSS antenna, in ECEF
%              coordinates relative to the primary antenna, in meters.
%              rbGtilde is constrained to satisfy norm(rbGtilde) = b, where b
%              is the known baseline distance between the two antennas.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

rI = S.statek.rI;
RBI = S.statek.RBI;

RpL = P.sensorParams.RpL;
sigmab = P.sensorParams.sigmab;
r0G = P.sensorParams.r0G;
ra1B = P.sensorParams.raB(:,1); % primary antenna 3x1 position in B
ra2B = P.sensorParams.raB(:,2); % secondary antenna 3x1 position in B

%% Primary
```

```
% Transform the inertial ECEF frame to the ENU frame, where vEnu = R*vEcef
RLG = Recef2enu(r0G);

% Find RpG
RpG = inv(RLG)*RpL*inv(RLG');

% Finding coordinates of primary antenna in I
rpI = rI + (RBI')*ra1B;

% Finding coordinates of primary antenna in G
rpG = (RLG')*rpI;

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RpG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rpGtilde
rpGtilde = rpG + w;

%% Baseline

% Finding coordinates of secondary antenna in I
rbI = rI + (RBI')*ra2B;

% Finding coordinates of secondary antenna in G
rbG = (RLG')*rbI;

% Find RbG
epsilon = 10^(-9);
rubG = rbG./norm(rbG);
RbG = ((norm(rbG)^2)*(sigmab^2)).*(eye(3) - (rubG*(rubG'))) + epsilon.*eye(3);

% Simulate noise vector
k = 1;
j = 1;
covarMatrix = RbG*eq(k,j);
w = mvnrnd(zeros(3,1), covarMatrix)';

% Finding rbGtilde with constraint norm(rbGtilde) = norm(rbG)
rbGtildeNoMag = rbG + w;
rbGtildeUnit = rbGtildeNoMag./norm(rbGtildeNoMag);
rbGtilde = norm(rbG).*rbGtildeUnit;
```

Figure 13: GNSS sensor model

```
Unset
function [ftildeB,omegaBtilde] = imuSimulator(S,P)
% imuSimulator : Simulates IMU measurements of specific force and
%                body-referenced angular rate.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                  vI = 3x1 velocity of CM with respect to the I frame and
%                       expressed in the I frame, in meters per second.
%
%                  aI = 3x1 acceleration of CM with respect to the I frame and
%                       expressed in the I frame, in meters per second^2.
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%           omegaBdot = 3x1 time derivative of omegaB, in radians per
%                       second^2.
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
% OUTPUTS
%
% ftildeB ---- 3x1 specific force measured by the IMU's 3-axis accelerometer
%
% omegaBtilde  3x1 angular rate measured by the IMU's 3-axis rate gyro
%
%+-------------------------------------------------------------------------+
% References:
%
%
% Author:
%+=========================================================================+
```

```
% Initializing input vectors
rI = S.statek.rI;
vI = S.statek.vI;
aI = S.statek.aI;
RBI = S.statek.RBI;
omegaB = S.statek.omegaB;
omegaBdot = S.statek.omegaBdot;

% Initializing sensor parameters
lB = P.sensorParams.lB;
Sa = P.sensorParams.Sa;
Qa = P.sensorParams.Qa;
sigmaa = P.sensorParams.sigmaa;
alphaa = P.sensorParams.alphaa;
Qa2 = P.sensorParams.Qa2;
Sg = P.sensorParams.Sg;
Qg = P.sensorParams.Qg;
sigmag = P.sensorParams.sigmag;
alphag = P.sensorParams.alphag;
Qg2 = P.sensorParams.Qg2;

% Initializing constants
g = P.constants.g;

%% Acceleration

% Find RI double dot
RIdd = aI;

% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qa*eq(k,j);
covarMatrix2 = Qa2*eq(k,j);
va = mvnrnd(zeros(3,1), covarMatrix1)';
va2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find accelerometer bias, update upon each call to the model
persistent ba;

if (isempty(ba))
    % Set ba's initial value
    QbaSteadyState = Qa2/(1 - alphaa^2);
    ba0 = mvnrnd(zeros(3,1), QbaSteadyState)';
    ba = [ba, ba0];
else
    bak1 = alphaa.*ba(:,end) + va2;
```

```
    ba = [ba, bak1];
end

% Find fBtilde
e3 = [0 0 1]';
ftildeB = RBI*(RIdd + g.*e3) + ba(:,end) + va;

%% Angular Rates

% Find noise vector
k = 1;
j = 1;
covarMatrix1 = Qg*eq(k,j);
covarMatrix2 = Qg2*eq(k,j);
vg = mvnrnd(zeros(3,1), covarMatrix1)';
vg2 = mvnrnd(zeros(3,1), covarMatrix2)';

% Find gyroscope bias, update upon each call to the model
persistent bg;

if(isempty(bg))
    % Set bg's initial value
    QbgSteadyState = Qg2/(1 - alphag^2);
    bg0 = mvnrnd(zeros(3,1), QbgSteadyState)';
    bg = [ba, bg0];
else
    bgk1 = alphag.*bg(:,end) + vg2;
    bg = [bg, bgk1];
end

% Find omegaBtilde
omegaBtilde = omegaB + bg(:,end) + vg;
```

Figure 14: IMU sensor model

```
Unset
function [rx] = hdCameraSimulator(rXI,S,P)
% hdCameraSimulator : Simulates feature location measurements from the
%                     quad's high-definition camera.
%
%
% INPUTS
%
% rXI -------- 3x1 location of a feature point expressed in I in meters.
%
% S ---------- Structure with the following elements:
```

```
%
%        statek = State of the quad at tk, expressed as a structure with the
%                 following elements:
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude of B frame wrt I frame
%
% P ---------- Structure with the following elements:
%
%  sensorParams = Structure containing all relevant parameters for the
%                 quad's sensors, as defined in sensorParamsScript.m
%
% OUTPUTS
%
% rx --------- 2x1 measured position of the feature point projection on the
%              camera's image plane, in pixels.  If the feature point is not
%              visible to the camera (the ray from the feature to the camera
%              center never intersects the image plane, or the feature is
%              behind the camera), then rx is an empty matrix.
%
%+-------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+===============================================================================+

rI = S.statek.rI;
RBI = S.statek.RBI;

rocB = P.sensorParams.rocB;
RCB = P.sensorParams.RCB;
Rc = P.sensorParams.Rc;
pixelSize = P.sensorParams.pixelSize;
K = P.sensorParams.K;
imagePlaneSize = P.sensorParams.imagePlaneSize;

X = [rXI;1];

% Solving for RCI
RCI = RCB * RBI;

% Solving for t
t = -RCI*(rI + ((RBI')*rocB));

% Solving for projection matrix PMat
```

```
zeroMatrix = [0 0 0];
K = [K,zeroMatrix'];
PMat = K*[RCI t; zeroMatrix 1];

% Solving for x using PX
xh = PMat*X;

% If feature is not in camera image plane, rx = []
rx = [];
if xh(3) <= 0
    return % If x(3) is negative, the feature is behind the camera (z-plane), end
script here
end

% Finding xc
x = xh(1)/xh(3);
y = xh(2)/xh(3);
xc = [x y]';

% Finding noise vector wc
k = 1;
j = 1;
covarMatrix = Rc*eq(k,j);
w = mvnrnd(zeros(2,1), covarMatrix)';

% Finding xctilde
xctilde = (1/pixelSize).*xc + w;

% Check if xctilde is inside the camera detection plane
imagePlaneX = imagePlaneSize(1)/2;
imagePlaneY = imagePlaneSize(2)/2;
featureCameraX = abs(xctilde(1));
featureCameraY = abs(xctilde(2));

if featureCameraX <= imagePlaneX && featureCameraY <= imagePlaneY
    rx = xctilde;
else
    rx = [];
end
```

Figure 15: Camera sensor model

```
Unset
function [zk] = h_meas(xk,wk,RBIBark,rXIMat,mcVeck,P)
% h_meas : Measurement model for quadcopter.
%
```

```
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to those
%              defined for E.statek in stateEstimatorUKF.m and where e is the
%              3x1 error Euler angle vector defined such that for an estimate
%              RBIBar of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIBar, where dRBI(e) is the DCM formed from the error
%              Euler angle vector e.
%
% wk --------- nz-by-1 measurement noise vector at time tk, defined as
%
%              wk = [wpIk', wbIk', w1C', w2C', ..., wNfkC']'
%
%              where nz = 6 + Nfk*3, with Nfk being the number of features
%              measured by the camera at time tk, and where all 3x1 noise
%              vectors represent additive noise on the corresponding
%              measurements.
%
% RBIBark ---- 3x3 attitude matrix estimate at time tk.
%
% rXIMat ----- Nf-by-3 matrix of coordinates of visual features in the
%              simulation environment, expressed in meters in the I
%              frame. rXIMat(i,:)' is the 3x1 vector of coordinates of the ith
%              feature.
%
% mcVeck ----- Nf-by-1 vector indicating whether the corresponding feature in
%              rXIMat is sensed by the camera: If mcVeck(i) is true (nonzero),
%              then a measurement of the visual feature with coordinates
%              rXIMat(i,:)' is assumed to be made by the camera.  mcVeck
%              should have Nfk nonzero values.
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%  sensorParams = Structure containing sensor parameters, as defined in
%                 sensorParamsScript.m
%
%
% OUTPUTS
```

```
%
% zk --------- nz-by-1 measurement vector at time tk, defined as
%
%              zk = [rpItilde', rbItildeu', v1Ctildeu', ..., vNfkCtildeu']'
%
%              where rpItilde is the 3x1 measured position of the primary
%              antenna in the I frame, rbItildeu is the 3x1 measured unit
%              vector pointing from the primary to the secondary antenna,
%              expressed in the I frame, and viCtildeu is the 3x1 unit vector,
%              expressed in the camera frame, pointing toward the ith 3D
%              feature, which has coordinates rXIMat(i,:)'.
%
%+---------------------------------------------------------------------------+
% References:
%
%
% Author:
%+===========================================================================+

% Unpack state vector
rI = xk(1:3);
vI = xk(4:6);
er = xk(7:9);
ba = xk(10:12);
bg = xk(13:15);

% Initialize needed parameters
rocB = P.sensorParams.rocB;
ra1B = P.sensorParams.raB(:,1);
ra2B = P.sensorParams.raB(:,2);
RCB = P.sensorParams.RCB;

% Find predicted RBI matrix
RBI = euler2dcm(er)*RBIBark;

% Set location of C frame origin in I
rcI = rI + (RBI')*rocB;

% Solve for vector pointing from primary to secondary
rbB = ra2B - ra1B;
rubB = rbB./norm(rbB);

% Solve for first two vectors of z(k)
zk = [rI + (RBI')*ra1B; (RBI')*rubB] + wk(1:6);

% Initializing indexes for noise to parse
startIndex = 6;
endIndex = 8;
```

```
% Solve for the feature vectors in z(k)
for i = 1:length(mcVeck)
    % If detected (nonzero), assess
    if mcVeck(i) ~= 0
        % Find vector pointing from camera center to the ith 3D feature in I
        viI = rXIMat(i,:)' - rcI;

        % Normalize for attitude calculation
        vuiI = viI./norm(viI);

        % Finding 3x1 noise vector from wk
        wki = wk(startIndex:endIndex);

        % Solve for vector in C frame
        vuiC = RCB*RBI*vuiI + wki;

        % Update zk for features
        zk = [zk;vuiC];
    end
    % Update noise array indexes
    startIndex = startIndex + 3;
    endIndex = endIndex + 3;
end
```

Figure 16: Sensor measurement model

```
Unset
function [xkp1] = f_dynamics(xk,uk,vk,delt,RBIHatk,P)
% f_dynamics : Discrete-time dynamics model for quadcopter.
%
% INPUTS
%
% xk --------- 15x1 state vector at time tk, defined as
%
%              xk = [rI', vI', e', ba', bg']'
%
%              where all corresponding quantities are identical to those
%              defined for E.statek in stateEstimatorUKF.m and where e is the
%              3x1 error Euler angle vector defined such that for an estimate
%              RBIHat of the attitude, the true attitude is RBI =
%              dRBI(e)*RBIHat, where dRBI(e) is the DCM formed from the error
%              Euler angle vector e.
%
% uk --------- 6x1 IMU measurement input vector at time tk, defined as
%
```

```
%              uk = [omegaBtilde', fBtilde']'
%
%              where all corresponding quantities are identical to those
%              defined for M in stateEstimatorUKF.m.
%
% vk --------- 12x1 process noise vector at time tk, defined as
%
%              vk = [vg', vg2', va', va2']'
%
%              where vg, vg2, va, and va2 are all 3x1 mutually-independent
%              samples from discrete-time zero-mean Gaussian noise processes.
%              These represent, respectively, the gyro white noise (rad/s),
%              the gyro bias driving noise (rad/s), the accelerometer white
%              noise (m/s^2), and the accelerometer bias driving noise
%              (m/s^2).
%
% delt ------- Propagation interval, in seconds.
%
% RBIHatk ---- 3x3 attitude matrix estimate at time tk.
%
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%  sensorParams = Structure containing sensor parameters, as defined in
%                 sensorParamsScript.m
%
%
% OUTPUTS
%
% xkp1 ------- 15x1 state vector propagated to time tkp1
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author:
%+==============================================================================+

if(abs(delt - P.sensorParams.IMUdelt) > 1e-9)
  error('Propagation time must be same as IMU measurement time');
end
```

```
% Initializing input vectors
rIk = xk(1:3); % 3x1
vIk = xk(4:6);
ek = xk(7:9);
bak = xk(10:12);
bgk = xk(13:15);
omegaBtildek = uk(1:3);
fBtildek = uk(4:6);
vgk = vk(1:3);
vg2k = vk(4:6);
vak = vk(7:9);
va2k = vk(10:12);
RBIk = euler2dcm(ek)*RBIHatk;
phik = ek(1);
thetak = ek(2);
psik = ek(3);

% Initializing sensor parameters
deltat = P.sensorParams.IMUdelt;
alphaa = P.sensorParams.alphaa;
alphag = P.sensorParams.alphag;
g = P.constants.g;

% Find aIk
ge3 = [0 0 g]';
aIk = ((RBIk')*(fBtildek - bak - vak)) - ge3;

% Find edotk
omegabk = omegaBtildek - bgk - vgk;
S = (1/cos(phik)).*[cos(phik)*cos(thetak), 0, cos(phik)*sin(thetak);
                    sin(phik)*sin(thetak), cos(phik), -cos(thetak)*sin(phik);
                    -sin(thetak), 0, cos(thetak)];
edotk = S*omegabk;

% Put together output vectors
rIkp1 = rIk + deltat.*vIk + 0.5*(deltat^2)*aIk;
vIkp1 = vIk + deltat.*aIk;
ekp1 = ek + deltat.*edotk;
bakp1 = alphaa*bak + va2k;
bgkp1 = alphag*bgk + vg2k;

% Output
xkp1 = [rIkp1;vIkp1;ekp1;bakp1;bgkp1];
```

Figure 17: Measurement-based dynamics model

Unset

```
function [E] = stateEstimatorUKF(S,M,P)
% stateEstimatorUKF : Unscented-Kalman-filter-based estimation of the state of
%                     quadcopter from sensor measurements.
%
% INPUTS
%
% S ---------- Structure with the following elements:
%
%        rXIMat = Nf-by-3 matrix of coordinates of visual features in the
%                 simulation environment, expressed in meters in the I
%                 frame. rXIMat(i,:)' is the 3x1 vector of coordinates of
%                 the ith feature.
%
%          delt = Measurement update interval, in seconds.
%
% M ---------- Structure with the following elements:
%
%            tk = Time at which all measurements apply, in seconds.
%
%      rpGtilde = 3x1 GNSS-measured position of the quad's primary GNSS
%                 antenna, in ECEF coordinates relative to the reference
%                 antenna, in meters.
%
%      rbGtilde = 3x1 GNSS-measured position of secondary GNSS antenna, in
%                 ECEF coordinates relative to the primary antenna, in meters.
%                 rbGtilde is constrained to satisfy norm(rbGtilde) = b, where b
%                 is the known baseline distance between the two antennas.
%
%         rxMat = Nf-by-2 matrix of measured positions of feature point
%                 projections on the camera's image plane, in
%                 pixels. rxMat(i,:)' is the 2x1 image position measurement of
%                 the 3D feature in rXIMat(i,:)'.  If the ith feature point is
%                 not visible to the camera (the ray from the feature to the
%                 camera center never intersects the image plane), then
%                 rxMat(i,:) = [NaN, NaN].
%
%       ftildeB = 3x1 specific force measured by the IMU's 3-axis
%                 accelerometer, in meters/sec^2
%
%   omegaBtilde = 3x1 angular rate measured by the IMU's 3-axis rate gyro,
%                 in rad/sec.
%
%
% P ---------- Structure with the following elements:
%
%    quadParams = Structure containing all relevant parameters for the
%                 quad, as defined in quadParamsScript.m
%
```

```
%     constants = Structure containing constants used in simulation and
%                 control, as defined in constantsScript.m
%
%  sensorParams = Structure containing sensor parameters, as defined in
%                 sensorParamsScript.m
%
%
% OUTPUTS
%
% E ---------- Structure with the following elements:
%
%        statek = Estimated state of the quad at tk, expressed as a structure
%                 with the following elements.
%
%                  rI = 3x1 position of CM in the I frame, in meters
%
%                 RBI = 3x3 direction cosine matrix indicating the
%                       attitude
%
%                  vI = 3x1 velocity of CM with respect to the I frame and
%                       expressed in the I frame, in meters per second.
%
%              omegaB = 3x1 angular rate vector expressed in the body frame,
%                       in radians per second.
%
%                  ba = 3x1 bias of accelerometer expressed in the
%                       accelerometer frame in meter/second^2.
%
%                  bg = 3x1 bias of rate gyro expressed in the body frame in
%                       rad/sec.
%
%             Pk = nx-by-nx error covariance matrix for the estimator state
%                  vector xk that applies at time tk, which is defined as
%
%                  xk = [rI', vI', e', ba', bg']'
%
%                  where all corresponding quantities are identical to those
%                  defined for E.statek and where e is the 3x1 error Euler
%                  angle vector defined such that for an estimate RBIHat of the
%                  attitude, the true attitude is RBI = dRBI(e)*RBIHat, where
%                  dRBI(e) is the DCM formed from the error Euler angle vector
%                  e, expressed in radians.
%
%+------------------------------------------------------------------------------+
% References:
%
%
% Author: Todd Humphreys
```

```
%+=========================================================================+

%----- Setup
persistent xBark PBark RBIBark
RIG = Recef2enu(P.sensorParams.r0G);
rbB = P.sensorParams.raB(:,2) - P.sensorParams.raB(:,1); rbBu = rbB/norm(rbB);
rpB = P.sensorParams.raB(:,1);  e3 = [0;0;1];
epsilon = 1e-8; nx = 15; nv = 12;
alphaUKF = 1e-3; betaUKF = 2; kappaUKF = 0;
lambda_p = alphaUKF^2*(kappaUKF + nx + nv) - nx - nv;
c_p = sqrt(nx+nv+lambda_p);
Wm0_p = lambda_p/(nx + nv + lambda_p);
Wmi_p = 1/(2*(nx + nv + lambda_p));
Wc0_p = Wm0_p + 1 - alphaUKF^2 + betaUKF; Wci_p = Wmi_p;

%----- Convert GNSS measurements to I frame
rpItilde = RIG*M.rpGtilde;
rbItilde = RIG*M.rbGtilde; rbItildeu = rbItilde/norm(rbItilde);

%----- Initialize state estimate on first call to this function
if(isempty(xBark))
  vIMat = [rbItildeu'; e3']; vBMat = [rbBu'; e3']; aVec = ones(2,1);
  RBIBark = wahbaSolver(aVec,vIMat,vBMat);
  rIBark = rpItilde - RBIBark'*rpB;
  xBark = [rIBark; zeros(12,1)];
  QbaSteadyState = P.sensorParams.Qa2/(1 - P.sensorParams.alphaa^2);
  QbgSteadyState = P.sensorParams.Qg2/(1 - P.sensorParams.alphag^2);
  PBark = diag([2*diag(P.sensorParams.RpL);
                0.001*ones(3,1); ...
                2*P.sensorParams.sigmab^2*ones(3,1); ...
                diag(QbaSteadyState); diag(QbgSteadyState)]);
end

%----- Assemble measurements
zk = [rpItilde; rbItildeu];
RcCCellArray = {}; jj = 1; Nfk = 0;
[Nf,~] = size(M.rxMat);
mcVeck = zeros(Nf,1);
for ii=1:Nf
  if(~isnan(M.rxMat(ii,1)))
    mcVeck(ii) = 1;
    viCtilde = [P.sensorParams.pixelSize*M.rxMat(ii,:)';P.sensorParams.f];
    norm_viCtilde = norm(viCtilde);
    viCtildeu = viCtilde/norm_viCtilde;
    % viCtildeu is the measured unit vector pointing from the camera (C) frame
    % origin to the 3D feature point, expressed in C.
    zk = [zk; viCtildeu];
    % Error covariance matrix for the unit vector viCtildeu
```

```
    sigmac = sqrt(P.sensorParams.Rc(1,1))*P.sensorParams.pixelSize/norm_viCtilde;
    RcC = sigmac^2*(eye(3) - viCtildeu*viCtildeu') + epsilon*eye(3);
    RcCCellArray{jj} = RcC;
    jj = jj + 1;
    Nfk = Nfk + 1;
  end
end

%----- Perform measurement update
% Form measurement error covariance matrix Rk
RpI = P.sensorParams.RpL;
rbIu = RBIBark'*rbBu;
RbI = P.sensorParams.sigmab^2*(eye(3)-rbIu*rbIu') + epsilon*eye(3);
Rk = blkdiag(RpI,RbI);
for ii=1:Nfk
  Rk = blkdiag(Rk,RcCCellArray{ii});
end
nz = length(zk);
lambda_u = alphaUKF^2*(kappaUKF + nx + nz) - nx - nz;
c_u = sqrt(nx+nz+lambda_u);
Wm0_u = lambda_u/(nx + nz + lambda_u);
Wmi_u = 1/(2*(nx + nz + lambda_u));
Wc0_u = Wm0_u + 1 - alphaUKF^2 + betaUKF;
Wci_u = Wmi_u;
% Form augmented a priori state and error covariance matrix
xBarAugk = [xBark; zeros(nz,1)];
PBarAugk = blkdiag(PBark,Rk);
SxBar = chol(PBarAugk)';
% Assemble sigma points and push these through the measurement function
sp0 = xBarAugk;
spMat = zeros(nx+nz, 2*(nx+nz));  zpMat = zeros(nz,2*(nx+nz));
zp0 = h_meas(sp0(1:nx),sp0(nx+1:end),RBIBark,S.rXIMat,mcVeck,P);
for ii=1:2*(nx+nz)
  jj = ii; pm = 1;
  if(ii > (nx + nz)) jj = ii - nx - nz; pm = -1; end
  spMat(:,ii) = sp0 + pm*c_u*SxBar(:,jj);
  zpMat(:,ii) =
h_meas(spMat(1:nx,ii),spMat(nx+1:end,ii),RBIBark,S.rXIMat,mcVeck,P);
end
% Recombine sigma points
zBark = sum([Wm0_u*zp0, Wmi_u*zpMat],2);
Pzz = Wc0_u*(zp0 - zBark)*(zp0 - zBark)';
Pxz = Wc0_u*(sp0(1:nx) - xBark)*(zp0 - zBark)';
for ii=1:2*(nx+nz)
  Pzz = Pzz + Wci_u*(zpMat(:,ii) - zBark)*(zpMat(:,ii) - zBark)';
  Pxz = Pxz + Wci_u*(spMat(1:nx,ii) - xBark)*(zpMat(:,ii) - zBark)';
end
% Perform LMMSE measurement update
```

```
PzzInv = inv(Pzz);
xHatk = xBark + Pxz*PzzInv*(zk - zBark);
Pk = PBark - Pxz*PzzInv*Pxz';

%----- Package output state
statek.rI = xHatk(1:3); statek.vI = xHatk(4:6); ek = xHatk(7:9);
statek.RBI = euler2dcm(ek)*RBIBark;
statek.ba = xHatk(10:12); statek.bg = xHatk(13:15);
statek.omegaB = M.omegaBtilde - statek.bg;
PkDiag = diag(Pk);
E.statek = statek;
E.Pk = Pk;

if(0)
% Testing section
tk = M.tk
[statek.rI - statekTrue.rI]
[statek.vI - statekTrue.vI]
dcm2euler(statekTrue.RBI' * statek.RBI)*180/pi
sqrt(PkDiag(7:9))*180/pi
pause;
clc;
end

%----- Propagate state to time tkp1
RBIHatk = statek.RBI; xHatk(7:9) = zeros(3,1);
Qk = blkdiag(P.sensorParams.Qg,P.sensorParams.Qg2,...
  P.sensorParams.Qa,P.sensorParams.Qa2);
xHatAugk = [xHatk; zeros(nv,1)];
PAugk = blkdiag(Pk,Qk);
Sx = chol(PAugk)';
% Assemble sigma points and push these through the dynamics function
sp0 = xHatAugk;
xpMat = zeros(nx,2*(nx+nv));
uk = [M.omegaBtilde;M.ftildeB];
xp0 = f_dynamics(sp0(1:nx),uk,sp0(nx+1:end),S.delt,RBIHatk,P);
for ii=1:2*(nx+nv)
  jj = ii; pm = 1;
  if(ii > (nx + nv)) jj = ii - nx - nv; pm = -1; end
  spii = sp0 + pm*c_p*Sx(:,jj);
  xpMat(:,ii) = f_dynamics(spii(1:nx),uk,spii(nx+1:end),S.delt,RBIHatk,P);
end
% Recombine sigma points
xBarkp1 = sum([Wm0_p*xp0, Wmi_p*xpMat],2);
PBarkp1 = Wc0_p*(xp0 - xBarkp1)*(xp0 - xBarkp1)';
for ii=1:2*(nx+nv)
  PBarkp1 = PBarkp1 + ...
            Wci_p*(xpMat(:,ii) - xBarkp1)*(xpMat(:,ii) - xBarkp1)';
```

```
end
ekp1 = xBarkp1(7:9);
RBIBarkp1 = euler2dcm(ekp1)*RBIHatk;
xBarkp1(7:9) = zeros(3,1);
% Set k = kp1 in preparation for next iteration
RBIBark = RBIBarkp1; xBark = xBarkp1; PBark = PBarkp1;
```

Figure 18: Unscented Kalman filter state estimator function

Unset

```
function [RBI] = wahbaSolver(aVec,vIMat,vBMat)
% wahbaSolver : Solves Wahba's problem via SVD.  In other words, this
%               function finds the rotation matrix RBI that minimizes the
%               cost Jw:
%
%                     N
%    Jw(RBI) = (1/2) sum ai*||viB - RBI*viI||^2
%                    i=1
%
%
% INPUTS
%
% aVec ------- Nx1 vector of least-squares weights.  aVec(i) is the weight
%              corresponding to the ith pair of vectors
%
% vIMat ------ Nx3 matrix of 3x1 unit vectors expressed in the I frame.
%              vIMat(i,:)' is the ith 3x1 vector.
%
% vBMat ------ Nx3 matrix of 3x1 unit vectors expressed in the B
%              frame. vBMat(i,:)' is the ith 3x1 vector, which corresponds to
%              vIMat(i,:)';
%
% OUTPUTS
%
% RBI -------- 3x3 direction cosine matrix indicating the attitude of the
%              B frame relative to the I frame.
%
%+-------------------------------------------------------------------------+
% References:
%
%
% Author:
%+=========================================================================+

% Find B
B = zeros(3);
```

```
for i=1:length(aVec)
    a = aVec(i);
    V = vIMat(i,:)';
    U = vBMat(i,:)';
    B = B + a.*U*V';
end

% Find U and V from B = USV' given B
[U,S,V] = svd(B);

% Initilize M
M = [1 0 0; 0 1 0; 0 0 det(U)*det(V)];

% Solve for
RBI = U*M*V';
```

Figure 19: Wahba's Problem solver function

Unset

```
% Create a random euler angle vector
e = [0, pi/2, 0]';

% Create the resulting RBI
RBI = euler2dcm(e);

% Initializing weights
aVec = ones(3);

%% No Noise Test
% Create random body frame unit vectors for function input
VB1 = rand(3,1); % 3X1
VB1 = VB1/norm(VB1);
VB2 = rand(3,1);
VB2 = VB2/norm(VB2);
VB3 = rand(3,1);
VB3 = VB3/norm(VB3);

% Calculate I frame vectors from body frame rotation using RBI
VI1 = RBI'*VB1; % 3x1
VI2 = RBI'*VB2;
VI3 = RBI'*VB3;

% Using predefined vectors to create VIMat and VBMat
VBMat = [VB1'; VB2'; VB3'];
VIMat = [VI1'; VI2'; VI3'];
```

```
% Attempt Wahba solver to get original RBI matrix
RBItest = wahbaSolver(aVec, VIMat, VBMat);

% disp(VBMat) % To show that the matrix values are changing
disp(RBItest)
disp(RBI)

%% Noise Test
% Create random body frame unit vectors for function input
VB1 = rand(3,1); % 3X1
VB1 = VB1/norm(VB1);
VB2 = rand(3,1);
VB2 = VB2/norm(VB2);
VB3 = rand(3,1);
VB3 = VB3/norm(VB3);

% Calculate I frame vectors from body frame rotation using RBI
VI1 = RBI'*VB1; % 3x1
VI2 = RBI'*VB2;
VI3 = RBI'*VB3;

% Add noise to the inertial frame vectors, after transformation
noiseVector1 = 0 + (0.5-0).*rand(3,1);
noiseVector2 = 0 + (0.5-0).*rand(3,1);
noiseVector3 = 0 + (0.5-0).*rand(3,1);
VI1 = VI1 + noiseVector1;
VI2 = VI2 + noiseVector2;
VI3 = VI3 + noiseVector3;

% Using predefined vectors to create VIMat and VBMat
VBMat = [VB1'; VB2'; VB3']; % NX3
VIMat = [VI1'; VI2'; VI3'];

% Attempt Wahba solver to get original RBI matrix
RBItest = wahbaSolver(aVec, VIMat, VBMat);

disp(RBItest)
disp(RBI)
```

Figure 20: Wahba's Problem solver testing script

```
Unset
#include <stack>
#include <iostream>
#include <algorithm>
```

```
#include "depth_first_search2d.h"

namespace game_engine {
  // Anonymous namespace. Put any file-local functions or variables in here
  namespace {
    // Helper struct that functions as a linked list with data. The linked
    // list represents a path. Data members are a node and a cost to reach
    // that node.
    struct NodeWrapper {
      std::shared_ptr<struct NodeWrapper> parent;
      std::shared_ptr<Node2D> node_ptr;
      double cost;

      // Equality operator
      bool operator==(const NodeWrapper& other) const {
        return *(this->node_ptr) == *(other.node_ptr);
      }
    };

    using NodeWrapperPtr =std::shared_ptr<NodeWrapper>;
    bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec) {
      for (auto n: nwPtrVec) {
       if (*n == *nwPtr) {
         return true;
       }
      }
      return false;
    }
  }

  PathInfo DepthFirstSearch2D::Run(
      const Graph2D& graph,
      const std::shared_ptr<Node2D> start_ptr,
      const std::shared_ptr<Node2D> end_ptr) {
    using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

    // SETUP
    Timer timer;
    timer.Start();

    // Use these data structures
    std::stack<NodeWrapperPtr> to_explore; // nodes to explore
    std::vector<NodeWrapperPtr> explored; // expored nodes
    PathInfo path_info; // Output structure

    // Create a NodeWrapperPtr
    NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
```

```
nw_ptr->parent = nullptr;
nw_ptr->node_ptr = start_ptr;
nw_ptr->cost = 0;
to_explore.push(nw_ptr);

/////////////// Edits Begin ////////////////////

// Check if the exploration pointer is empty
while (!to_explore.empty()) {

  // Find the item to explore using first value of to_explore and remove
  NodeWrapperPtr item_to_explore = to_explore.top();
  to_explore.pop();

  // Check if item_to_explore has already been explored
  if (is_present(item_to_explore, explored)) {
   continue;
  }

  // Add node to explored vector
  explored.push_back(item_to_explore);

  // Check if agent reached the end node
  if (*item_to_explore->node_ptr == *end_ptr) {

   // Stop timer
   path_info.details.run_time = timer.Stop();

   // Set up PathInfo
   path_info.path = {};
   path_info.details.path_cost = item_to_explore->cost;
   path_info.details.num_nodes_explored = explored.size();
   path_info.path.push_back(item_to_explore->node_ptr);
   path_info.details.path_length = 1;

   // Creating output path list
   while ((*item_to_explore->node_ptr != *start_ptr)) {
     // Push current node then update to parent
     item_to_explore = item_to_explore->parent;
     path_info.path.push_back(item_to_explore->node_ptr);
     path_info.details.path_length += 1;
   }

   // You must return a reversed PathInfo
   std::reverse(path_info.path.begin(), path_info.path.end());
   path_info.details.path_length = path_info.path.size();

   // Exit while loop since path is found
```

```
      break;
      }

      // If not the end then find next path

      // Use graph to find the neighbor nodes of the node_to_explore
      auto edges = graph.Edges(item_to_explore->node_ptr);

      // Push all neighbors nodes to to_explore
      for(auto edge : edges) {

       // Create a NodeWrapperPtr for each neighbor node
       NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();

       // Update exploring node to parent node
       neighbor_ptr->parent = item_to_explore;

       // Set neighbor node pointer coordinates
       neighbor_ptr->node_ptr = edge.Sink();

       // Set cost of neighbor exploration
       neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

       // Push neighbor instance to the to_explore stack for next iteration
       to_explore.push(neighbor_ptr);
      }
    }

    return path_info;
    //////////////Edits End///////////////

  }
}
```

Figure 21: Depth First Search algorithm

```
Unset
#include <queue>
#include <iostream>
#include <algorithm>

#include "dijkstra2d.h"

namespace game_engine {
  // Anonymous namespace. Put any file-local functions or variables in here
  namespace {
```

```
  // Helper struct that functions as a linked list with data. The linked
  // list represents a path. Data members are a node and a cost to reach
  // that node.
  struct NodeWrapper {
    std::shared_ptr<struct NodeWrapper> parent;
    std::shared_ptr<Node2D> node_ptr;
    double cost;

    // Equality operator
    bool operator==(const NodeWrapper& other) const {
      return *(this->node_ptr) == *(other.node_ptr);
    }
  };

  using NodeWrapperPtr =std::shared_ptr<NodeWrapper>;
  bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec) {
    for (auto n: nwPtrVec) {
     if (*n == *nwPtr) {
       return true;
     }
    }
    return false;
  }

  // Helper function. Compares the values of two NodeWrapper pointers.
  // Necessary for the priority queue.
  bool NodeWrapperPtrCompare(
      const std::shared_ptr<NodeWrapper>& lhs,
      const std::shared_ptr<NodeWrapper>& rhs) {
    return lhs->cost > rhs->cost;
  }
}

PathInfo Dijkstra2D::Run(
    const Graph2D& graph,
    const std::shared_ptr<Node2D> start_ptr,
    const std::shared_ptr<Node2D> end_ptr) {
  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

  // SETUP
  Timer timer;
  timer.Start();

  // Use these data structures
  std::priority_queue<
    NodeWrapperPtr,
    std::vector<NodeWrapperPtr>,
```

```
  std::function<bool(
      const NodeWrapperPtr&,
      const NodeWrapperPtr& )>>
  to_explore(NodeWrapperPtrCompare); // To explore priority queue
std::vector<NodeWrapperPtr> explored; // Explored vector
PathInfo path_info; // Output structure

// Create a NodeWrapperPtr
NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
nw_ptr->parent = nullptr;
nw_ptr->node_ptr = start_ptr;
nw_ptr->cost = 0;
to_explore.push(nw_ptr);

/////////////// Edits Begin ////////////////////

// Check if the exploration pointer is empty
while (!to_explore.empty()) {

  // Find the item to explore using first value of to_explore and remove
  NodeWrapperPtr item_to_explore = to_explore.top();
  to_explore.pop();

  // Check if item_to_explore has already been explored
  if (is_present(item_to_explore, explored)) {
   continue;
  }

  // Add node to explored vector
  explored.push_back(item_to_explore);

  // Check if agent reached the end node
  if (*item_to_explore->node_ptr == *end_ptr) {

   // Stop timer
   path_info.details.run_time = timer.Stop();

   // Set up PathInfo
   path_info.path = {};
   path_info.details.path_cost = item_to_explore->cost;
   path_info.details.num_nodes_explored = explored.size();
   path_info.path.push_back(item_to_explore->node_ptr);
   path_info.details.path_length = 1;

   // Creating output path list
   while ((*item_to_explore->node_ptr != *start_ptr)) {
     // Push current node then update to parent
     item_to_explore = item_to_explore->parent;
```

```
        path_info.path.push_back(item_to_explore->node_ptr);
        path_info.details.path_length += 1;
      }

      // You must return a reversed PathInfo
      std::reverse(path_info.path.begin(), path_info.path.end());
      path_info.details.path_length = path_info.path.size();

      // Exit while loop since path is found
      break;
     }

     // If not the end then find next path

     // Use graph to find the neighbor nodes of the node_to_explore
     auto edges = graph.Edges(item_to_explore->node_ptr);

     // Push all neighbors nodes to to_explore
     for(auto edge : edges) {

      // Create a NodeWrapperPtr for each neighbor node
      NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();

      // Update parent node to exploring node
      neighbor_ptr->parent = item_to_explore;

      // Set neighbor node pointer coordinates
      neighbor_ptr->node_ptr = edge.Sink();

      // Set cost of neighbor exploration
      neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

      // Push neighbor instance to the to_explore stack for next iteration
      to_explore.push(neighbor_ptr);
     }
    }

    return path_info;
    ///////////////Edits End///////////////

  }

}
```

Figure 22: Dijkstra's algorithm

Unset

```
#include <queue>
#include <iostream>
#include <algorithm>
#include <math.h>
#include <cstdlib>

#include "a_star2d.h"

namespace game_engine {
  // Anonymous namespace.
  namespace {
    // Helper struct that functions as a linked list with data. The linked
    // list represents a path. Data members are a node, a cost to reach that
    // node, and a heuristic cost from the current node to the destination.
    struct NodeWrapper {
      std::shared_ptr<struct NodeWrapper> parent;
      std::shared_ptr<Node2D> node_ptr;

      // True cost to this node
      double cost;

      // Heuristic to end node
      double heuristic;

      // Equality operator
      bool operator==(const NodeWrapper& other) const {
        return *(this->node_ptr) == *(other.node_ptr);
      }
    };

    using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;
    bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec) {
      for (auto n: nwPtrVec) {
       if (*n == *nwPtr) {
         return true;
       }
      }
      return false;
    }

    // Helper function. Compares the values of two NodeWrapper pointers.
    // Necessary for the priority queue.
    bool NodeWrapperPtrCompare(
        const std::shared_ptr<NodeWrapper>& lhs,
        const std::shared_ptr<NodeWrapper>& rhs) {
      return lhs->cost + lhs->heuristic > rhs->cost + rhs->heuristic;
    }
```

```
// HEURISTIC FUNCTION
double Heuristic(
    const std::shared_ptr<Node2D>& current_ptr,
    const std::shared_ptr<Node2D>& end_ptr) {
  double deltax = end_ptr->Data().x() - current_ptr->Data().x();
  double deltay = end_ptr->Data().y() - current_ptr->Data().y();
  float euclidean = pow(((deltax * deltax) + (deltay * deltay)), 0.5);
  return euclidean;
}

// OVERESTIMATE
/*
double Heuristic(
              const std::shared_ptr<Node2D>& current_ptr,
              const std::shared_ptr<Node2D>& end_ptr) {
  double deltax = end_ptr->Data().x() - current_ptr->Data().x();
  double deltay = end_ptr->Data().y() - current_ptr->Data().y();
  float manhattan = abs(deltax) + abs(deltay);
  return manhattan;
}
*/

// DISTINCT FUNCTION
/*
double Heuristic(
              const std::shared_ptr<Node2D>& current_ptr,
              const std::shared_ptr<Node2D>& end_ptr) {
  double deltax = end_ptr->Data().x() - current_ptr->Data().x();
  double deltay = end_ptr->Data().y() - current_ptr->Data().y();
  float euclidean = pow(((deltax * deltax) + (deltay * deltay)), 0.5);
  float distinct = euclidean/2;
  return distinct;
}
*/

// DISTINCT FUNCTION 2
/*
double Heuristic(
              const std::shared_ptr<Node2D>& current_ptr,
              const std::shared_ptr<Node2D>& end_ptr) {
  double deltax = end_ptr->Data().x() - current_ptr->Data().x();
  double deltay = end_ptr->Data().y() - current_ptr->Data().y();
  float manhattan = abs(deltax) + abs(deltay);
  float distinct = manhattan*2;
  return distinct;
}
*/
```

```
  // ZERO FUNCTION
  /*
  double Heuristic(
                const std::shared_ptr<Node2D>& current_ptr,
                const std::shared_ptr<Node2D>& end_ptr) {
    return 0;
  }
  */
}

PathInfo AStar2D::Run(
    const Graph2D& graph,
    const std::shared_ptr<Node2D> start_ptr,
    const std::shared_ptr<Node2D> end_ptr) {
  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

  // SETUP
  Timer timer;
  timer.Start();

  // Use these data structures
  std::priority_queue<
    NodeWrapperPtr,
    std::vector<NodeWrapperPtr>,
    std::function<bool(
        const NodeWrapperPtr&,
        const NodeWrapperPtr& )>>
    to_explore(NodeWrapperPtrCompare); // Priority queue
  std::vector<NodeWrapperPtr> explored; // Explored vector
  PathInfo path_info; // Output structure

  // Create a NodeWrapperPtr
  NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
  nw_ptr->parent = nullptr;
  nw_ptr->node_ptr = start_ptr;
  nw_ptr->cost = 0;
  nw_ptr->heuristic = Heuristic(start_ptr, end_ptr);
  to_explore.push(nw_ptr);

  /////////////// Edits Begin ////////////////////

  // Check if the exploration pointer is empty
  while (!to_explore.empty()) {

    // Find the item to explore using first value of to_explore and remove
    NodeWrapperPtr item_to_explore = to_explore.top();
    to_explore.pop();
```

```
// Check if item_to_explore has already been explored
if (is_present(item_to_explore, explored)) {
 continue;
}

// Add node to explored vector
explored.push_back(item_to_explore);

// Check if agent reached the end node
if (*item_to_explore->node_ptr == *end_ptr) {

 // Stop timer
 path_info.details.run_time = timer.Stop();

 // Set up PathInfo
 path_info.path = {};
 path_info.details.path_cost = item_to_explore->cost;
 path_info.details.num_nodes_explored = explored.size();
 path_info.path.push_back(item_to_explore->node_ptr);
 path_info.details.path_length = 1;

 // Creating output path list
 while ((*item_to_explore->node_ptr != *start_ptr)) {
   // Push current node then update to parent
   item_to_explore = item_to_explore->parent;
   path_info.path.push_back(item_to_explore->node_ptr);
   path_info.details.path_length += 1;
 }

 // You must return a reversed PathInfo
 std::reverse(path_info.path.begin(), path_info.path.end());
 path_info.details.path_length = path_info.path.size();

 // Exit while loop since path is found
 break;
}

// If not the end then find next path

// Use graph to find the neighbor nodes of the node_to_explore
auto edges = graph.Edges(item_to_explore->node_ptr);

// Push all neighbors nodes to to_explore
for(auto edge : edges) {

 // Create a NodeWrapperPtr for each neighbor node
 NodeWrapperPtr neighbor_ptr = std::make_shared<NodeWrapper>();
```

```
      // Update parent node to exploring node
      neighbor_ptr->parent = item_to_explore;

      // Set neighbor node pointer coordinates
      neighbor_ptr->node_ptr = edge.Sink();

      // Update heuristic value
      neighbor_ptr->heuristic = Heuristic(neighbor_ptr->node_ptr, end_ptr);

      // Set cost of neighbor exploration
      neighbor_ptr->cost = item_to_explore->cost + edge.Cost();

      // Push neighbor instance to the to_explore stack for next iteration
      to_explore.push(neighbor_ptr);
     }
    }

    return path_info;
    ///////////////Edits End///////////////

  }
}
```

Figure 23: A* algorithm

Unset

```
#include <cstdlib>
#include <vector>
#include <cmath>

#include "polynomial_solver.h"
#include "polynomial_sampler.h"
#include "gnuplot-iostream.h"

////////////////////////////////////////////////////////////////////
// FUNCTION PROTOTYPES
// DO NOT MODIFY
////////////////////////////////////////////////////////////////////
// void Example();
// void DerivativeExperiments();
// void ArrivalTimeExperiments();
void NumWaypointExperiments();

////////////////////////////////////////////////////////////////////
// MAIN FUNCTION
```

```
// TODO: UNCOMMENT THE FUNCTIONS YOU WANT TO RUN
////////////////////////////////////////////////////////////////
int main(int argc, char** argv) {
  // Example();
  // DerivativeExperiments();
  // ArrivalTimeExperiments();
  NumWaypointExperiments();

  return EXIT_SUCCESS;
}

// Example function that demonstrates how to use the polynomial solver. This
// example creates waypoints in a triangle: (0,0) -- (1,0) -- (1,1) -- (0,0)
void Example() {
  // Time in seconds
  const std::vector<double> times = {0,1,2,3};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constrains position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constrains position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The fourth node constrains position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 2;   /////////// CHANGE  Minimize the 2nd order
```

```
  osqp_set_default_settings(&solver_options.osqp_settings);
  solver_options.osqp_settings.polish = true;       // Polish the solution,
getting the best answer possible
  solver_options.osqp_settings.verbose = true;     // Suppress the printout

  // Use p4::PolynomialSolver object to solve for polynomial trajectories
  p4::PolynomialSolver solver(solver_options);
  const p4::PolynomialSolver::Solution path
    = solver.Run(
        times,
        node_equality_bounds,
        {},
        {});

  // Sampling and Plotting
  { // Plot 2D position
    // Options to configure the polynomial sampler with
    p4::PolynomialSampler::Options sampler_options;
    sampler_options.frequency = 100;             // Number of samples per second
    sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

    // Use this object to sample a trajectory
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
```

```
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void DerivativeExperiments() {
  // Time in seconds
  const std::vector<double> times = {0,1,2,3,4};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ////////////////////////////////////////////////////////////////////
    // TODO: CREATE A SQUARE TRAJECTORY
    ////////////////////////////////////////////////////////////////////

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constrains position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constrains position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The third node constrains position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,1),
```

```
    // The fifth node constrains position
    p4::NodeEqualityBound(0,4,0,0),
    p4::NodeEqualityBound(1,4,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 4;   // TODO: VARY THE DERIVATIVE ORDER

  osqp_set_default_settings(&solver_options.osqp_settings);
  solver_options.osqp_settings.polish = true;       // Polish the solution,
getting the best answer possible
  solver_options.osqp_settings.verbose = false;     // Suppress the printout

  // Use p4::PolynomialSolver object to solve for polynomial trajectories
  p4::PolynomialSolver solver(solver_options);
  const p4::PolynomialSolver::Solution path
    = solver.Run(
        times,
        node_equality_bounds,
        {},
        {});

  // Sampling and Plotting
  { // Plot 2D position
    // Options to configure the polynomial sampler with
    p4::PolynomialSampler::Options sampler_options;
    sampler_options.frequency = 100;             // Number of samples per second
    sampler_options.derivative_order = 0;        // IMPORTANT: Derivative to
sample

    // Use this object to sample a trajectory
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
```

```
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void ArrivalTimeExperiments() {
  // Time in seconds
  const std::vector<double> times = {0,1,2,3,4};
  // const std::vector<double> times = {0,0.02,0.04,0.06,0.08};
  // const std::vector<double> times = {0,10,20,30,40};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ////////////////////////////////////////////////////////////////////
    // TODO: CREATE A SQUARE TRAJECTORY
    ////////////////////////////////////////////////////////////////////

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
```

```
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constrains position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constrains position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The third node constrains position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,1),

    // The fifth node constrains position
    p4::NodeEqualityBound(0,4,0,0),
    p4::NodeEqualityBound(1,4,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 2;   // IMPORTANT

  osqp_set_default_settings(&solver_options.osqp_settings);
  solver_options.osqp_settings.polish = true;       // Polish the solution,
getting the best answer possible
  solver_options.osqp_settings.verbose = false;     // Suppress the printout

  // Use p4::PolynomialSolver object to solve for polynomial trajectories
  p4::PolynomialSolver solver(solver_options);
  const p4::PolynomialSolver::Solution path
    = solver.Run(
        times,
        node_equality_bounds,
        {},
        {});

  // Sampling and Plotting
  { // Plot 2D position
    // Options to configure the polynomial sampler with
    p4::PolynomialSampler::Options sampler_options;
    sampler_options.frequency = 100;             // Number of samples per second
    sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)
```

```
    // Use this object to sample a trajectory
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void NumWaypointExperiments() {
  // Independant variables
  double N = 100;
  int r = 1;
```

```
// Time in seconds
std::vector<double> times = {};
for (double i = 0; i <= N; i++) {
  times.push_back(i);
}
// std::vector<double> times = {0,1,2,3,4};
// std::vector<double> times = {0,1,2,3,4,5,6,7,8};
// std::vector<double> times = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

// Initialize the vector
std::vector<p4::NodeEqualityBound> node_equality_bounds = {

  // The first node must constrain position, velocity, and acceleration at (1,0)
  p4::NodeEqualityBound(0,0,0,r),
  p4::NodeEqualityBound(1,0,0,0),
  p4::NodeEqualityBound(0,0,1,r),
  p4::NodeEqualityBound(1,0,1,0),
  p4::NodeEqualityBound(0,0,2,r),
  p4::NodeEqualityBound(1,0,2,0),

};

// Add the other nodes

// Set PI
const double PI = M_PI;

// Set the angle variation
double angle_shift = (2*PI)/N;
double curr_angle = angle_shift;

// Create a for loop that vaires with N waypoints
for (double i = 1; i <= N; i++) {

  // Find the next point using the angle variation
  double x = r*cos(curr_angle);
  double y = r*sin(curr_angle);

  // Create waypoint node coordinates constrining position
  node_equality_bounds.push_back(p4::NodeEqualityBound(0,i,0,x));
  node_equality_bounds.push_back(p4::NodeEqualityBound(1,i,0,y));

  // Update curr_angle
  curr_angle += angle_shift;
}
```

```
  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 4;   // Minimize snap

  osqp_set_default_settings(&solver_options.osqp_settings);
  solver_options.osqp_settings.polish = true;       // Polish the solution,
getting the best answer possible
  solver_options.osqp_settings.verbose = false;     // Suppress the printout

  // Use p4::PolynomialSolver object to solve for polynomial trajectories
  p4::PolynomialSolver solver(solver_options);
  const p4::PolynomialSolver::Solution path
    = solver.Run(
        times,
        node_equality_bounds,
        {},
        {});

  // Sampling and Plotting
  { // Plot 2D position
    // Options to configure the polynomial sampler with
    p4::PolynomialSampler::Options sampler_options;
    sampler_options.frequency = 100;             // Number of samples per second
    sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

    // Use this object to sample a trajectory
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
```

```
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}
```

Figure 24: Polynomial planning test script

```
Unset
#include <cstdlib>
#include <vector>
#include <fstream>
#include <string.h>

#include "a_star2d.h"
#include "occupancy_grid2d.h"
#include "path_info.h"
#include "polynomial_solver.h"
#include "polynomial_sampler.h"
#include "gnuplot-iostream.h"
#include "gui2d.h"

using namespace game_engine;

// PROTOTYPE
PathInfo RunAStar(
    const Graph2D& graph,
    const OccupancyGrid2D* occupancy_grid,
```

```
    const std::shared_ptr<Node2D>& start_node,
    const std::shared_ptr<Node2D>& end_node);

int main(int argc, char** argv) {
  if(argc != 6) {
    std::cerr << "Usage: ./full_stack_planning occupancy_grid_file row1 col1 row2
col2" << std::endl;
    return EXIT_FAILURE;
  }

  // Parsing input
  const std::string occupancy_grid_file = argv[1];
  const std::shared_ptr<Node2D> start_node = std::make_shared<Node2D>(
      Eigen::Vector2d(std::stoi(argv[2]),std::stoi(argv[3])));
  const std::shared_ptr<Node2D> end_node = std::make_shared<Node2D>(
      Eigen::Vector2d(std::stoi(argv[4]),std::stoi(argv[5])));

  // Load an occupancy grid from a file
  OccupancyGrid2D occupancy_grid;
  occupancy_grid.LoadFromFile(occupancy_grid_file);

  // Transform an occupancy grid into a graph
  const Graph2D graph = occupancy_grid.AsGraph();

  auto path_info_ret = RunAStar(graph, &occupancy_grid, start_node, end_node);
  auto node_path = path_info_ret.path;

  // Find the number of waypoints
  int N = node_path.size();

  // Time in seconds, a second per waypoint
  std::vector<double> times = {};
  for (double i = 0; i < N; i++) {
    times.push_back(i);
  }

  // Find the first node coordinates
  auto starting_node = node_path[0];
  auto x1 = starting_node->Data().x();
  auto y1 = starting_node->Data().y();

  // Initialize the waypoint vector
  std::vector<p4::NodeEqualityBound> node_equality_bounds = {

    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,x1),
    p4::NodeEqualityBound(1,0,0,y1),
    p4::NodeEqualityBound(0,0,1,x1),
```

```
  p4::NodeEqualityBound(1,0,1,y1),
  p4::NodeEqualityBound(0,0,2,x1),
  p4::NodeEqualityBound(1,0,2,y1),

};

// Add the other nodes
for (double i = 1; i < N; i++) {

  // Find node coordinates
  auto current_node = node_path[i];
  auto x = current_node->Data().x();
  auto y = current_node->Data().y();

  // Create waypoint node coordinates constrining position
  node_equality_bounds.push_back(p4::NodeEqualityBound(0,i,0,x));
  node_equality_bounds.push_back(p4::NodeEqualityBound(1,i,0,y));
}

// Options to configure the polynomial solver with
p4::PolynomialSolver::Options solver_options;
solver_options.num_dimensions = 2;     // 2D
solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
solver_options.continuity_order = 4;   // Require continuity to the 4th order
solver_options.derivative_order = 2;   /////////// CHANGE  Minimize the 2nd order

osqp_set_default_settings(&solver_options.osqp_settings);
solver_options.osqp_settings.polish = true;
solver_options.osqp_settings.verbose = true;

// Use p4::PolynomialSolver object to solve for polynomial trajectories
p4::PolynomialSolver solver(solver_options);
const p4::PolynomialSolver::Solution path
  = solver.Run(
      times,
      node_equality_bounds,
      {},
      {});

// Sampling the desired derivative
p4::PolynomialSampler::Options sampler_options;
sampler_options.frequency = 200;             // Number of samples per second
sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

// Use this object to sample a trajectory
p4::PolynomialSampler sampler(sampler_options);
Eigen::MatrixXd samples = sampler.Run(times, path);
```

```
  // Plotting tool requires vectors
  std::vector<std::string> t_hist, x_hist, y_hist;
  for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
    t_hist.push_back(std::__cxx11::to_string(samples(0,time_idx)));
    x_hist.push_back(std::__cxx11::to_string(samples(1,time_idx)));
    y_hist.push_back(std::__cxx11::to_string(samples(2,time_idx)));
  }

  // Download the time history of the desired derivative
  std::ofstream MyFile("./data.txt");

  // Create easiest format file
  for (int i = 0; i < t_hist.size(); i++) {
    MyFile<<y_hist[i]<<", "<<x_hist[i]<<", 0"<<std::endl;
  }

  return EXIT_SUCCESS;
}

// HELPER FUNCTIONS
PathInfo RunAStar(
    const Graph2D& graph,
    const OccupancyGrid2D* occupancy_grid,
    const std::shared_ptr<Node2D>& start_node,
    const std::shared_ptr<Node2D>& end_node) {

  // Run A*
  AStar2D a_star;
  PathInfo path_info = a_star.Run(graph, start_node, end_node);
  return path_info;
}
```

Figure 25: Full stack planning test script