ASE 479W Laboratory 4 Report
Find the Balloon

Aaron Pandian

April 7, 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 said 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 provides the ability to conduct environmental assumptions. In integrating computer vision, a quadrotor can use these assumptions to control itself accordingly.

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 control 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. The error between this estimated state and the desired trajectory state is minimized using the developed feedback loop.

This architecture is used to develop and test a method of computer vision for a quadrotor. The goal of which is twofold: first, detect if a blue or red balloon is present, and second, estimate the 3D coordinates of the balloon from the detecting image.

The theory, application, and test results of this capability 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 Finding Location of 3D Feature Point from Camera Image with Known Pose

In trying to estimate 3D information from a project image of the feature in question, understanding how this can be possible is the first step. Similar to the problem depicted in Fig. 1, to conduct this process, it is imperative that more than two camera measurements are obtained, otherwise the method of triangulation estimation will not work.

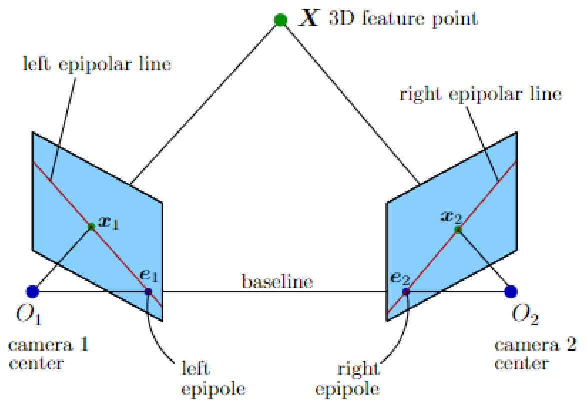

Figure 1: Simplified triangulation camera visual of feature point

Given at least two of the noisy measurements shown in Fig. 1, the following information from Fig. 1 is obtained.

$$\tilde{\boldsymbol{x}}_1 = \begin{bmatrix} p_s \tilde{\boldsymbol{x}}_{c1} \\ 1 \end{bmatrix}, \quad \tilde{\boldsymbol{x}}_2 = \begin{bmatrix} p_s \tilde{\boldsymbol{x}}_{c2} \\ 1 \end{bmatrix} \tag{1}$$

Assuming the pose and attitude of the camera at ***Oi*** are known, we are presented with enough data to solve the triangulation problem above. A simplified approach to solving this is a linear least squares test, which, as tested in the following sections, presents a passing accuracy result. This simplified approach works under the assumption that the camera measurement from (1) is noise free. As a result, the equation relating the projected coordinates ***x*** to the 3D coordinates ***X*** using the projection matrix ***P*** is as follows.

$$\boldsymbol{x} \times P\boldsymbol{X} = \boldsymbol{0} \tag{2}$$

In the above equation, using the definition of ***P*** above and ***x*** = [x, y, 1]^T, we section out its three rows as ***P***^{iT} to format the partial system of equations in (3).

$$\begin{aligned} x(\boldsymbol{p}^{3\mathsf{T}}\boldsymbol{X}) - (\boldsymbol{p}^{1\mathsf{T}}\boldsymbol{X}) = 0 \\ y(\boldsymbol{p}^{3\mathsf{T}}\boldsymbol{X}) - (\boldsymbol{p}^{2\mathsf{T}}\boldsymbol{X}) = 0 \end{aligned} \tag{3}$$

With standard factorization, ***X*** can be separated to show the two equations above are linearly independent.

$$\underbrace{\begin{bmatrix} \tilde{x}_1 \boldsymbol{p}_1^{3\mathsf{T}} - \boldsymbol{p}_1^{1\mathsf{T}} \\ \tilde{y}_1 \boldsymbol{p}_1^{3\mathsf{T}} - \boldsymbol{p}_1^{2\mathsf{T}} \\ \tilde{x}_2 \boldsymbol{p}_2^{3\mathsf{T}} - \boldsymbol{p}_2^{1\mathsf{T}} \\ \tilde{y}_2 \boldsymbol{p}_2^{3\mathsf{T}} - \boldsymbol{p}_2^{2\mathsf{T}} \end{bmatrix}}_{H} \boldsymbol{X} + \underbrace{\begin{bmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{bmatrix}}_{\boldsymbol{w}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4}$$

Then using the two equations from (3) for each camera *N* presented in the solution, where in Fig. 1 there is 2, the H matrix is populated to size (2*N*, 4). This enables the rearrangement of the equation into a standard linear measurement model, evident in (6).

$$\underbrace{\begin{bmatrix} H_r & -z \end{bmatrix}}_{H} \underbrace{\begin{bmatrix} \boldsymbol{X}_r \\ 1 \end{bmatrix}}_{\boldsymbol{X}} + w = \boldsymbol{0} \tag{5}$$

$$z = H_r \boldsymbol{X}_r + w \tag{6}$$

In reconstructing the ***H*** matrix from (4) into the separated matrices $\boldsymbol{H}_r$ and ***z***, and $\boldsymbol{X}_r$ from the initial ***X*** matrix shown in (5), the standard linear measurement model is derived. The format of this equation allows for the isolation of the $\boldsymbol{X}_r$ term to solve using the least squares method. Assuming that the noise $\boldsymbol{\omega}$ has a mean E[$\boldsymbol{\omega}$] = 0 and covariance matrix R = E[$\boldsymbol{\omega}\boldsymbol{\omega}^T$] $\in$ $R^{2N \times 2N}$, the optimal solution for $\boldsymbol{X}_r$ is shown below.

$$\hat{\boldsymbol{X}}_r = \left(H_r^T R^{-1} H_r\right)^{-1} H_r^T R^{-1} z \tag{7}$$

$$P_x = \mathbb{E}\left[\left(\hat{\boldsymbol{X}}_r - \boldsymbol{X}_r\right)\left(\hat{\boldsymbol{X}}_r - \boldsymbol{X}_r\right)^T\right] = \left(H_r^T R^{-1} H_r\right)^{-1} \tag{8}$$

Using (8) and (9), the 3D feature coordinate estimation and the corresponding error covariance matrix can be calculated respectively. The *R* matrix above can be calculated using the covariance of the pixel-level measurement R_c and the pixel size p_s presented in (9).

$$R = p_s^2 \begin{bmatrix} R_c & & 0 \\ & \ddots & \\ 0 & & R_c \end{bmatrix} \tag{9}$$

The complete method expressed above is assessed in both the quadrotor flight simulation and the computer vision schematic using functions shown and discussed in the subsequent sections.

3 Implementation

To implement camera feature detection and pose estimation, a piecewise approach was conducted. The relevant code used is highlighted and discussed below. Furthermore, this section explores the relationship between the pieces of the quadrotor computer vision architecture.

A method of simulating captured images- detailing where a feature is to be detected within the camera frame and information on the state of the camera at the time the image was captured- is developed first to conduct any image processing.

Unset

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

rx = [];
% Construct transformation matrix P = K*[RCI,t]
RCI = P.sensorParams.RCB * S.statek.RBI;
t = -RCI*(S.statek.rI + S.statek.RBI'*P.sensorParams.rocB);
Pk = P.sensorParams.K*[RCI t];

% Generate projection
x = Pk*[rXI;1];
xc = x(1:2)/x(3);

% Check if point is within image plane
if(x(3) > 0 && abs(xc(1)) < P.sensorParams.imagePlaneSize(1)/2 && ...
  abs(xc(2)) < P.sensorParams.imagePlaneSize(2)/2)

  Rac = chol(P.sensorParams.Rc);
  rx = (1/P.sensorParams.pixelSize)*xc + Rac'*randn(2,1);
end
```

Figure 2: Noisy camera measurement model

As shown above, the camera model 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 (10).

$$\underbrace{\begin{bmatrix} fX_c \\ fY_c \\ Z_c \end{bmatrix}}_{\boldsymbol{x}_{ci}} = \left[\begin{array}{c|c} \underbrace{\begin{matrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{matrix}}_{K} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \end{array}\right] \begin{bmatrix} R_{\mathrm{CI}} & \boldsymbol{t}_{\mathrm{C}} \\ 0_{1\times 3} & 1 \end{bmatrix} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\boldsymbol{X}_{i\mathrm{I}}} \tag{10}$$

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. 2. 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.

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} = \frac{1}{p_s} \begin{bmatrix} x \\ y \end{bmatrix} + \underbrace{\begin{bmatrix} w_x \\ w_y \end{bmatrix}}_{\boldsymbol{w}_c} \tag{11}$$

The pixel size p_s is used to scale the true projection and a generated noise vector $\boldsymbol{w}_c$, derived with a predefined covariance matrix, is used to generate the noise distortion.

The 3D feature location estimation of the camera measurements is done by solving the linear least squares problem discussed in section 2.1. The implementation of the theory can be seen below.

```
Unset

function [rXIHat,Px] = estimate3dFeatureLocation(M,P)
% Begin for loop through cameras to create H and R
for i = 1:length(rx)
   % Rotation from Oc to Oi
   RCIk = RCI{i};

   % Translation from Oi to Oc in C
   tIk = -tI{i};
   tCk = RCIk*tIk;
   % Projection matrix for ith camera
   Pk = K*[RCIk tCk];
   % Initialize projection point in meters
   xk = pixelSize*rx{i};
   % Update H
   Pkr1 = Pk(1,:);
   Pkr2 = Pk(2,:);
   Pkr3 = Pk(3,:);
   Hk = [xk(1)*Pkr3 - Pkr1; xk(2)*Pkr3 - Pkr2];
   H = [H; Hk];
   % Update RcPrime array
   RcPrime = [RcPrime, Rc]; % Creating array of length 2N
end
% Create covariance matrix R
RcPrime = diag(RcPrime);
R = (pixelSize^2)*RcPrime;
% Create Hr and z matrices
Hr = H(:,1:end-1);
z = -1*H(:,end);
% Solve for error covarience matrix
Px = (Hr'*(R^(-1))*Hr)^(-1);
```

```
% Rearrange to solve for Xr, location of feature point
rXIHat = Px*Hr'*(R^(-1))*z;
```

Figure 3: 3D feature location estimation function

The two highlighted functions above are integrated, amongst the other modeling and control systems, into a simulation script. This simulation function propagates the true state of the quadrotor, then implements the sensor models to mimic noisy measurements. These measurements are fed into the controllers within the feedback architecture to identify the necessary changes to the control variables needed to best match the input reference trajectory. The function from Fig. 3 operates within this script isolated from the state propagation.

```
Unset
function [Q, Ms] = simulateQuadrotorEstimationAndControl(R,S,P)
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
  % Building Ms -------------------
  if (kk/imageTimeDelta==imagesTaken)
      if (max(size(Ms.rxArray)) < 10)
      % Simulate image and populate measurement arrays
      [rxp] = hdCameraSimulator(rXI,Sm,P);

      if (length(rxp)==2)
          Ms.rxArray{imagesSaved} = rxp;
          %Ms.RCIArray{imagesSaved} = P.sensorParams.RCB * Sm.statek.RBI;
          Ms.RCIArray{imagesSaved} = P.sensorParams.RCB * E.statek.RBI;
          %Ms.rcArray{imagesSaved} = Sm.statek.rI +
Sm.statek.RBI'*P.sensorParams.rocB;
          Ms.rcArray{imagesSaved} = E.statek.rI +
E.statek.RBI'*P.sensorParams.rocB;
          imagesSaved = imagesSaved + 1;
      end
      imagesTaken = imagesTaken + 1;
      end
  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
```

Figure 4: Simulate quadrotor estimation and control function

To conduct feature estimation tests, images are created or “taken,” using the function from Fig. 2, at constant time intervals across the simulation. In the case no feature is detected at that time, the image is discarded; this process ends when ten images detecting the feature are amassed. These ten images output, from the simulation function to the function in Fig. 3, the pixel coordinates and camera state in a Matlab structure. The operation of the feature location estimation is housed in the top level script running the simulation- indicated in Fig. 4.

```
Unset
clear all; clc;
%rng(1234);
rng('shuffle');
% 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 = [0,0,0.7];
%S.rXIMat = [];
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;
[Q, Ms] = simulateQuadrotorEstimationAndControl(R,S,P);
% Estimate 3D feature location
[rXIHat,Px] = estimate3dFeatureLocation(Ms,P);
% Compare rXI and rXIHat
fprintf('rXI compared with rXIHat (should be within a few cm): ');
[S.rXIMat' rXIHat]
% Examine diagonal elements of Px
fprintf('Sqrt of diagonal elements of Px');
sqrt(diag(Px))
```

Figure 5: Top level simulation script

In integrating these functions, the top level script is employed to visualize and assess the control application for a generated circular trajectory. Furthermore, in receiving the structure with image information, the location estimation function solves for the 3D feature coordinates in the inertial frame. This is then compared with the true feature location, as shown in Fig. 4.

In addition to the tests set up in Matlab, the function from Fig. 3 was translated to C++ to test the computer vision schematic on a set of pictures. The implementation of the estimation can be depicted in the balloon finder script developed to isolate and retrieve the image plane coordinate centers of balloons of specified color within the provided image.

Unset
```
#include "balloonfinder.h"

#include <Eigen/Dense>
#include <cassert>
```

```
#include <opencv2/core/eigen.hpp>

#include "navtoolbox.h"

bool BalloonFinder::findBalloonsOfSpecifiedColor(
    const cv::Mat* image, const Eigen::Matrix3d RCI, const Eigen::Vector3d rc_I,
    const BalloonFinder::BalloonColor color,
    std::vector<Eigen::Vector2d>* rxVec) {

  bool returnValue = false;
  rxVec->clear();
  cv::Mat original;
  if (debuggingEnabled_) original = image->clone();
  const size_t nCols_m1 = image->cols - 1;
  const size_t nRows_m1 = image->rows - 1;
  // Blur the image to reduce small-scale noise
  cv::Mat framep;
  cv::GaussianBlur(*image, framep, cv::Size(21, 21), 0, 0);
  cv::cvtColor(framep, framep, cv::COLOR_BGR2HSV);

  // Finding red balloon
  if (color == BalloonColor::RED) {
    cv::Scalar colorLower_l(0, 80, 100), colorLower_h(10, 255, 255);
    cv::Scalar colorUpper_l(170, 80, 100), colorUpper_h(180, 255, 255);
    cv::Mat mLower, mUpper;
    cv::inRange(framep, colorLower_l, colorLower_h, mLower);
    cv::inRange(framep, colorUpper_l, colorUpper_h, mUpper);
    framep = mLower | mUpper;
    // Erode image to eliminate stray wisps of red
    constexpr int iterations = 5;
    cv::erode(framep, framep, cv::Mat(), cv::Point(-1, -1), iterations);
    // Dilate image to restore red square to original size
    cv::dilate(framep, framep, cv::Mat(), cv::Point(-1, -1), iterations);
    // Find contours
    std::vector<std::vector<cv::Point>> contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(framep, contours, hierarchy, cv::RETR_EXTERNAL,
cv::CHAIN_APPROX_SIMPLE);
    cv::RNG rng(12345);
    cv::Point2f center;
    float radius;
    constexpr float maxAspectRatio = 1.475;
    constexpr float minAspectRatio = 1.20;
    constexpr float minRadius = 50;
    constexpr float maxRadius = 250;
    constexpr int minPointsFor_fitEllipse = 5;
    for (size_t ii = 0; ii < contours.size(); ii++) {
      const cv::Scalar color = cv::Scalar(rng.uniform(0, 256), rng.uniform(0,
256), rng.uniform(0, 256));
      cv::minEnclosingCircle(contours[ii], center, radius);
      float aspectRatio = maxAspectRatio;

      // Touching edge check
      if (touchesEdge(framep, contours[ii]) == true) {
        // If touching edge, skip the current contour
        continue;
      }

      // Too small check
      if (contours[ii].size() >= minPointsFor_fitEllipse) {
        cv::RotatedRect boundingRectangle = cv::fitEllipse(contours[ii]);
        const cv::Size2f rectSize = boundingRectangle.size;
        aspectRatio =
            static_cast<float>(std::max(rectSize.width, rectSize.height)) /
            std::min(rectSize.width, rectSize.height);
      }

      // Wrong aspect ratio check
      std::cout << "aspectRatio: " << aspectRatio << ", radius: " << radius <<
std::endl;
      cv::drawContours(*image, contours, ii, color, 2, cv::LINE_8, hierarchy, 0);
      if (aspectRatio > minAspectRatio && aspectRatio < maxAspectRatio && radius >
minRadius && radius < maxRadius) {
        cv::circle(*image, center, static_cast<int>(radius), color, 2);
        // Adjust center
        Eigen::Vector2d xc_pixels;
        xc_pixels(0) = center.x;
        xc_pixels(1) = center.y;
        auto rxx = nCols_m1 - xc_pixels(0);
        auto rxy = nRows_m1 - xc_pixels(1);
        // Push rx to rxVec
        Eigen::Vector2d rx;
        rx(0) = rxx;
        rx(1) = rxy;
        rxVec->push_back(rx);
        // Found red balloon
        returnValue = true;
        std::cout << "This red image passed with " << rx(0) << ", " << rx(1) <<
std::endl;
      }
    }
  } else { // Finding blue balloon
    cv::Scalar colorLower_l(85, 80, 80), colorLower_h(100, 255, 255);
    cv::Scalar colorUpper_l(100, 80, 80), colorUpper_h(110, 255, 255);
    cv::Mat mLower, mUpper;
    cv::inRange(framep, colorLower_l, colorLower_h, mLower);
    cv::inRange(framep, colorUpper_l, colorUpper_h, mUpper);
    framep = mLower | mUpper;
    // Erode image to eliminate stray wisps of blue
    constexpr int iterations = 5;
    cv::erode(framep, framep, cv::Mat(), cv::Point(-1, -1), iterations);
    // Dilate image to restore red square to original size
    cv::dilate(framep, framep, cv::Mat(), cv::Point(-1, -1), iterations);
    // Find contours
    std::vector<std::vector<cv::Point>> contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(framep, contours, hierarchy, cv::RETR_EXTERNAL,
cv::CHAIN_APPROX_SIMPLE);
    cv::RNG rng(12345);
    cv::Point2f center;
    float radius;
    constexpr float maxAspectRatio = 1.475;
    constexpr float minAspectRatio = 1.20;
    constexpr float minRadius = 70;
    constexpr float maxRadius = 250;
    constexpr int minPointsFor_fitEllipse = 5;
    for (size_t ii = 0; ii < contours.size(); ii++) {
      const cv::Scalar color = cv::Scalar(rng.uniform(0, 256), rng.uniform(0,
256), rng.uniform(0, 256));
      cv::minEnclosingCircle(contours[ii], center, radius);
      float aspectRatio = maxAspectRatio;

      // Touching edge check
      if (touchesEdge(framep, contours[ii]) == true) {
        // If touching edge, skip the current contour
        continue;
      }

      // Too small check
      if (contours[ii].size() >= minPointsFor_fitEllipse) {
        cv::RotatedRect boundingRectangle = cv::fitEllipse(contours[ii]);
        const cv::Size2f rectSize = boundingRectangle.size;
        aspectRatio =
            static_cast<float>(std::max(rectSize.width, rectSize.height)) /
            std::min(rectSize.width, rectSize.height);
      }

      // Wrong aspect ratio check
      std::cout << "aspectRatio: " << aspectRatio << ", radius: " << radius <<
std::endl;
      cv::drawContours(*image, contours, ii, color, 2, cv::LINE_8, hierarchy, 0);
      if (aspectRatio > minAspectRatio && aspectRatio < maxAspectRatio && radius >
minRadius && radius < maxRadius) {
        cv::circle(*image, center, static_cast<int>(radius), color, 2);
```

```
          // Adjust center
          Eigen::Vector2d xc_pixels;
          xc_pixels(0) = center.x;
          xc_pixels(1) = center.y;
          auto rxx = nCols_m1 - xc_pixels(0);
          auto rxy = nRows_m1 - xc_pixels(1);
          // Push rx to rxVec
          Eigen::Vector2d rx;
          rx(0) = rxx;
          rx(1) = rxy;
          rxVec->push_back(rx);
          // Found blue balloon
          returnValue = true;
          std::cout << "This blue image passed with " << rx(0) << ", " << rx(1) <<
  std::endl;
        }
      }
    }
```

Figure 6: Balloon finder function

These functions set up the foundation for conducting in-depth dynamical analysis on a quadcopter and the efficacy of the developed computer vision program.

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 Isolated Feature Location Estimation Test

Using a provided test case, the results from the 3D location estimation function fall within a 10 cm error. This test case was set to estimate the true feature point at an Inertial frame location of [0 0 0.5]. For the estimation function to operate, at least two measurements or images of the feature must be captured- falling in line with the requirements of triangulation described in section 2.1. As a result, two images were generated. To do this, the Fig. 2 function was called, calculating the feature location on the image plane using example vector positions and attitude matrices of the quadrotor center of mass. Creating a structure of image information, as described before, the results of the developed location estimation function are below.

	Estimated ***rXI*** Vector	Error Covariance Matrix
Trial 1	$[0.0206\ \ 0.0492\ \ 0.7088]^T$	$[0.0077\ \ 0.0193\ \ 0.0040]^T$

Trial 2	[-0.0679 -0.0130 0.4890]^T	[0.0224 0.0122 0.0145]^T
Trial 3	[0.0082 -0.0008 0.4906]^T	[0.0207 0.0120 0.0145]^T
Trial 4	[0.0123 0.0148 0.4910]^T	[0.0206 0.0117 0.0145]^T
Trial 5	[-0.0233 -0.0048 0.4949]^T	[0.0214 0.0120 0.0144]^T
Average	[-0.0100 0.0091 0.5348]^T	[0.0186 0.0134 0.0124]^T

Table 1: Results of the feature location estimation function across repeated tests

The constant conditions previously described were used to evaluate the feature location five times. The results demonstrate the feature location in the Inertial frame in meters and the error covariance matrix, or a representation of precision. Given the average estimated coordinate value, the absolute error between it and the known location is [0.0100 0.0091 0.0348]^T in meters. This means the developed method is capable of estimating the feature location in 3D space within centimeters of actual location.

4.2 Integrated Feature Location Estimation Test

Now applying the estimation function to the quadrotor simulation, Fig. 14 and 19 allow the observation of how the method works with ten images generated in constant intervals.

	Average ***rXI*** Vector Estimate	Absolute Error Matrix	Average Error Covariance Matrix
Δt = 20s	[0.0016 -0.0141 0.7032]^T	[0.0016 0.0141 0.0032]^T	[0.0075 0.0129 0.0043]^T
Δt = 50s	[-0.0020 0.0011 0.6944]^T	[0.0020 0.0011 0.0056]^T	[0.0044 0.0088 0.0036]^T
Δt = 80s	[0.0039 -0.0014 0.7030]^T	[0.0039 0.0014 0.0030]^T	[0.0053 0.0058 0.0034]^T

Table 2: Results of feature location estimation with varying image frequency using true state; 5 trial average

From the Δt = 80s case in Table 2, the absolute error matrix displays a significant improvement compared to the case observed in the previous section. In fact, in almost every case the estimated feature location, in each axis, is closer to the true coordinate point of [0 0 0.7] meters than the average estimation with two images calculated in the previous section.

Additionally, in comparing the Δt = 20s case to the Δt = 80s case, improved accuracy is clearly evident. In almost every parameter the 80 second spacing displays greater precision. It can be noted that the lower the error covariance values, the more precise that dimensional evaluation can be assumed. An explanation for this is that with greater time intervals, the image taken is more likely to be vastly different, in terms of point of view to the feature, than the image prior. This, in turn, leads to better triangulation accuracy. In the context of two points, this is because, when noise is introduced, the epipolar line of points closer together can offset the estimation or intersection point more than if the points were spaced slightly further apart.

When the number of points increases, as observed when more images are provided for estimation, the problem is mitigated. However, the resulting precision of the estimation is still for low image capture frequencies. The Δt = 20s case has the highest error covariance matrix average, and, in practice, the highest computational requirements.

It is also counterintuitive to take images at a very low frequency because then the risk of missing capturing the feature increases. Thus, a medium of imaging frequency where results are relatively similar can be derived. As evident from Table 2, the Δt = 50s and Δt = 80s cases performed similarly.

	Average ***rXI*** Vector Estimate	Absolute Error Matrix	Average Error Covariance Matrix
Δt = 80s	$[0.0090\ 0.0054\ 0.6989]^T$	$[0.0090\ 0.0054\ 0.0011]^T$	$[0.0053\ 0.0058\ 0.0035]^T$

Table 3: Results of feature location estimation using estimated state; 5 trial average

After changing the camera pose to the projected state instead of its true location, the average performance of the developed estimation algorithm across five trials is depicted in Table 3. The error covariance matrix and absolute error highlight effective precision given the realistic condition. In using the estimated state from the developed sensor models, the resulting feature estimation shows resilience in its invariance. The Δt = 80s case from Table 2 shows this to be the case as the average location estimate changes by millimeters or less in each Inertial axis value.

4.3 Developed Ballon Finder Evaluation

Utilizing the updated method found in Fig. 6, the computer vision capability was tested on a set of provided images containing red and blue objects- of which, the function was to isolate and predict the 3D location of the red and blue balloons. The test results can be found in Table 4.

	Average ***rXI*** Vector Estimate Error	Average Error Covariance Matrix Square Root Diagonal
Blue Balloon	[0.00405113 -0.0204551 -0.0163551]	[2538.37 1659.93 1130.85]
Red Balloon	[-0.0351931 -0.0201116 0.00302867]	[3629.91 1725.77 1304.92]

Table 4: Results of feature location estimation for known balloon location; 5 trial average

The ***rXI*** Vector estimate, measured in meters, from Table 4, can be seen to the millimeter in both Balloon cases. This was done by using an aspect ratio range of 1.2 - 1.475 and a radius range of 50 - 250. With these parameters, approximately 23 images were used to inform the estimation algorithm with projected balloon center coordinates and camera state information. As a result, a highly effective method was developed, capable of successfully estimating feature position in 3D space well within 10 cm.

5 Conclusion

A computer vision architecture was developed to detect a red or blue balloon and estimate its location in 3D space based off of the captured image. The estimation method aims to solve a linear least squares problem using the current camera state and the location of the image on the image plane (in pixels). It was found that the estimation function works best with more images at a higher imaging frequency.

Furthermore, image processing is conducted to strip the image of distracting features. The image is converted from RGB to HSV color schemes to highlight red and blue hues, which are identified using specified value ranges. To ensure a colored object is a balloon, aspect ratio and radius tests are employed. Cases at the edge of the image plane are also ignored, to an extent. The processing is complete with erosion and dilation. The resulting method is able to *successfully* isolate a balloon within an image, identify its color, and estimate its location in 3D space to 10 cm accuracy.

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.