

ASE 389 Laboratory 5 Report

Find the Balloon

Justin Hart

April 10, 2023

1 Introduction

In previous labs, feature locations were input into the quad simulator in order to find the position and orientation of the quad. In a reverse sense, if the position and orientation of multiple cameras are known (or the same camera in multiple locations), this information can be used to locate objects in 3D space from the camera images. This is known as structure computation and is described in [2].

In this laboratory assignment, an estimator will be built that estimates the 3D location of objects from 2D feature locations which are extracted from a set of images taken by cameras with known position and orientation. Then, OpenCV will be implemented in C++ along with a C++ version of the feature location estimator in order to locate the center of a red and blue balloon in 3D space from a set of camera images.

2 Theoretical Analysis

There is no theoretical analysis requirements in this lab.

3 Implementation

The implementation of the code for this lab is done first using MATLAB (to estimate the 3D location of an object from 2D feature locations). Then, the same method is implemented within C++ in the Aerial Robotics virtual machine (VM). Lastly, a balloon finder is developed within C++ using OpenCV tools and a given set of images with balloons.

3.1 MATLAB Structure Computation

The structure computation solution method (D) from [1] is translated into MATLAB code within the `estimate3dFeatureLocation.m` function (see Appendix A for all code). This script takes in an array of measured positions of the feature point projection onto the camera's image plane in pixels, an array of I to camera frame (C) attitude matrices for each measurement, and an array of camera center positions, as well as general sensor parameters. The estimated location of the feature point expressed in I , along with the 3×3 error covariance matrix for the estimated location are the outputs. This is done by finding the projection matrix for each camera, then forming a measurement model of the form given in (8.5) in [1]. This model is broken down into separate components and these components are combined with an approximation for the covariance matrix of the pixel level measurement to get a least squares optimal

solution for the feature location as well as the solution's error covariance matrix. These equations are shown in (1) and (2) respectively:

$$\hat{X}_r = (H_r^T R^{-1} H_r)^{-1} H_r^T R^{-1} z \quad (1)$$

$$P_x = \mathbb{E} \left[\left(\hat{X}_r - X_r \right) \left(\hat{X}_r - X_r \right)^T \right] = (H_r^T R^{-1} H_r)^{-1} \quad (2)$$

3.2 C++ Structure Computation

The same method used in section 3.1 is implemented into C++ within a function called `computeStructure()` which is a method within the `StructureComputer` class (see Appendix A). This function takes in sensor parameters, and camera bundle objects which each possess the 2x1 coordinates of the feature point as projected onto that camera's image plane in pixels, the 3x3 attitude matrix relating the C and I frames at the instant the image was taken, as well as the 3x1 position of the camera center the instant the image was taken in the I frame. Using the same method (D) in [1] to solve the structure computation problem, the estimated 3D feature location and estimated location's error covariance matrix are output into a `Point` object, `point`.

3.3 C++ Balloon Finder

A function to find balloons of a specified color within an image set is developed within C++ within the `findBalloonsOfSpecifiedColor` function, which is a method within the `BalloonFinder` class. This method takes in a pointer to an image and a specified balloon color and returns true if a balloon or multiple balloons of that color are found within the given image (utilizing OpenCV tools). The balloon center locations in pixels are returned in a vector of vectors, `rxVec`. The I to C attitude matrix for each image and the camera center position for each image are also input into the function for debugging purposes.

First, the input image is smoothed using a Gaussian blur function and output into a processed image, `framep`. Then the color space is changed to hue, saturation, value (HSV) so the colors in the image are stored in one value: Hue. Based on the input color, a switch block is implemented, designating appropriate ranges for red and blue HSV values. The values used are shown below:

```

switch(color){
case BalloonColor::RED:{
    Scalar colorLower_l(0, 120, 100), colorLower_h(10,255,255);
    Scalar colorUpper_l(170, 120, 100), colorUpper_h(180,255,255);
    Mat mLower, mUpper; //image can be thought of as matrix of pixels
    //mLower and mUpper will store 1 for every pixel in their respective in ranges
    inRange(framep,colorLower_l,colorLower_h,mLower);
    inRange(framep,colorUpper_l,colorUpper_h,mUpper);
    framep = mLower | mUpper; //bit wise 'or' both matrices
    balloonColor = Scalar(0,0,255);
    balloon_color = "Red";
    break;
}
case BalloonColor::BLUE:{
    Scalar lower_blue(90,60,100), upper_blue(110,255,255);
    Mat mOut;
    inRange(framep,lower_blue,upper_blue,mOut);
    framep = mOut;
    balloonColor = Scalar(255,0,0);
    balloon_color = "Blue";
    break;
}
default:{
    throw std::runtime_error("Color value is not handled in this program.");
}
}

```

Next, the image is opened by using 10 iterations of erosion followed by 10 iterations of dilation. This removes color noise within the image. The contours for the blocks of remaining color are found. Using designated radius and aspect ratio ranges shown below, only the contours within these ranges are considered balloons.

```

constexpr float maxAspectRatio = 1.45;
constexpr float minAspectRatio = 1.18;
constexpr float maxRadius = 300;
constexpr float minRadius = 50;

```

The center and radius of each contour is found using the minEnclosingCircle OpenCV function. The aspect ratio is found by creating a bounding rectangle for each contour with the fitEllipse OpenCV function. The maximum dimension is then divided by the minimum dimension of this bounding rectangle. If the contour is considered a balloon, it's center is converted into the image plane coordinate system in pixels using the code below and pushed into the rxVec vector. A value of true is returned if a balloon is found and false is returned otherwise.

```

rx(0) = nCols_m1 - center.x;
rx(1) = nRows_m1 - center.y;
rxVec->push_back(rx);
returnValue = true;

```

If debugging is enabled for this function, a copy of the original image is created and the contours found are plotted to this copy. If a balloon is found, a circle is plotted around it using it's center and radius value, and the estimated center of the balloon is plotted. The true center value is plotted as well. The aspect ratio and radius of each contour is output to the terminal, along with the estimated vs. true value of the balloon center coordinates.

4 Results and Analysis

4.1 Estimating Feature Location from Camera Measurements MATLAB

The estimate3dFeatureLocation.m function was testing using the provided camTest.m script. Three different results are shown below:

```
rXI compared with rXIHat (should be within a few cm):  
ans =
```

```
      0      0.0179  
      0     -0.0097  
0.5000      0.4985
```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

```
ans =
```

```
0.0205  
0.0121  
0.0144
```

```
rXI compared with rXIHat (should be within a few cm):  
ans =
```

```
      0     -0.0013  
      0     -0.0053  
0.5000      0.5102
```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

```
ans =
```

```
0.0209  
0.0120  
0.0141
```

```
rXI compared with rXIHat (should be within a few cm):  
ans =
```

```
      0      0.0202  
      0      0.0126  
0.5000      0.5089
```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

```
ans =
```

```
0.0204  
0.0118  
0.0141
```

On average, the estimated 3D feature point location is within 2cm of the true location and is within the designated error values from the error covariance matrix.

Next, the `estimate3dFeatureLocation.m` function was tested using the MATLAB quad simulator flying in a circular trajectory as dictated in Lab assignment 3. The `simulateQuadrotorEstimationAndControl.m` script was modified to also output a structure, `Ms`, with an `rxArray`, `RCIArray`, and `rcArray`. These are feature point projection positions on the camera's image plane in pixels, I to C attitude matrices, and camera center positions respectively. These were found using the `hdCameraSimulator` function, sensor parameters, and the current state of the quad as shown below. The image feature location used was `[0;0;0.7]`.

```
image_span = (N-500)/10;

kk = 1;
for ii=500:image_span:N-1
    rx = hdCameraSimulator(rXI,SCMat(ii),P);
    if isempty(rx)
        rxArray{kk} = [NaN;NaN];
    else
        rxArray{kk} = rx;
    end

    RCI = P.sensorParams.RCB*SCMat(ii).statek.RBI;
    RCIArray{kk} = RCI;

    rc = SCMat(ii).statek.rI+SCMat(ii).statek.RBI'*P.sensorParams.rocB;
    rcArray{kk} = rc;

    kk = kk+1;
end

Ms.rxArray = rxArray;
Ms.RCIArray = RCIArray;
Ms.rcArray = rcArray;
```

The `estimate3dFeatureLocation.m` function was applied to the `Ms` structure within `topSimulateControl.m` and compared with the true value of the image feature location:

```
% Estimate 3D feature location
[rXIHat,Px] = estimate3dFeatureLocation(Ms,P);

% Compare rXI and rXIHat
disp('rXI compared with rXIHat (should be within a few cm): ');
[S.rXIMat(2,:) ' rXIHat]

% Examine diagonal elements of Px
disp(['Sqrt of diagonal elements of Px (errors in rXIHat should not be much ' ...
    'bigger than these numbers): ']);
sqrt(diag(Px))
```

For the first set of tests, the true attitude and position of the camera corresponding to each feature measurement was used. The test images used were varied in spacing around the circle for comparison and the estimator was turned on in topSimulateControl.m. Three results of 10 images spaced 150 samples apart are shown below:

rXI compared with rXIHat (should be within a few cm):

ans =

```

      0      0.0193
      0     -0.0241
0.7000      0.6984

```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

```

0.0052
0.0057
0.0034

```

rXI compared with rXIHat (should be within a few cm):

ans =

```

      0     -0.0063
      0      0.0025
0.7000      0.7007

```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

```

0.0052
0.0057
0.0034

```

rXI compared with rXIHat (should be within a few cm):

ans =

```

      0     -0.0036
      0     -0.0055
0.7000      0.6976

```

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

```

0.0052
0.0057
0.0034

```

Three results of 10 images spaced 15 samples apart are shown below:

rXI compared with rXIHat (should be within a few cm):

ans =

0	-0.0119
0	0.0528
0.7000	0.7049

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0072
0.0271
0.0051

rXI compared with rXIHat (should be within a few cm):

ans =

0	0.0012
0	0.0118
0.7000	0.6995

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0073
0.0277
0.0051

rXI compared with rXIHat (should be within a few cm):

ans =

0	-0.0083
0	0.0383
0.7000	0.7082

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0071
0.0271
0.0049

On average, the images spaced further apart seem to be slightly more accurate in predicting the 3D feature location.

Lastly, the estimate3dFeatureLocation was tested using the estimated camera position and attitude instead of the true pose. Three results for the estimated camera pose test using 10 images spaced 150 samples apart are shown below:

rXI compared with rXIHat (should be within a few cm):

ans =

0	-0.0068
0	0.0048
0.7000	0.7035

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0052
0.0057
0.0034

rXI compared with rXIHat (should be within a few cm):

ans =

0	-0.0178
0	0.0287
0.7000	0.6899

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0052
0.0057
0.0034

rXI compared with rXIHat (should be within a few cm):

ans =

0	-0.0016
0	0.0016
0.7000	0.6981

Sqrt of diagonal elements of Px (errors in rXIHat should not be much bigger than these numbers):

ans =

0.0052
0.0057
0.0034

On average, the estimated values for the 3D feature location are slightly less accurate than when using the estimated camera pose vs. using the true values. This makes sense intuitively as using the estimated camera pose would introduce additional error within the measurement and estimation.

4.2 Integrated C++ Test

In order to test the C++ `computeStructure` and `findBalloonsOfSpecifiedColor` functions, the `locateBalloon` executable was run within the Aerial Robotics VM on the easy-ones image set. The terminal output from this test is shown below:

```
Blue balloon estimated 3D location error:
-0.00867198
-0.0333635
-0.0157806
Red balloon estimated 3D location error:
-0.0213855
0.00632616
-0.00205082
Blue error covariance matrix sqrt diagonal:
0.00453293
0.0029082
0.00211216
Red error covariance matrix sqrt diagonal:
0.0064773
0.00544595
0.00371723
```

The estimated blue and red balloon location errors are less than 10cm. The largest error is in the blue balloon's Y location in the I frame and this is about 3cm.

When the `locateBalloon` executable was run in debugging mode, a copy of each image was output with drawn contours, circles drawn around balloons, balloon center estimated locations drawn, and true balloon center locations plotted for reference. Example red and blue balloon images are shown in Fig. 1a and 1b below:



Figure 1: (a) Red balloon debugging image example. (b) Blue balloon debugging image example.

An example of the terminal output in debugging mode is shown below. The color of the balloon tested in the current image, all contour aspect ratios and radii, the color of any found balloons along with its center, and the true balloon center for the image were output for reference.

```
Processing image frame00796.jpg
Red testing:
Red balloon found, cx: 831.5, cy: 597
aspectRatio: 1.34997, radius: 221.204
true cx: 789.812, true cy: 543.866
Blue testing:
aspectRatio: 2.42278, radius: 74.359
aspectRatio: 1.17176, radius: 48.9184
aspectRatio: 1.22795, radius: 19.1051
Blue balloon found, cx: 1905.5, cy: 108
aspectRatio: 1.3609, radius: 77.1509
true cx: 1892.63, true cy: 85.7627
```

5 Conclusion

A feature location estimator was implemented in MATLAB and tested using the previously designed quad simulator. This same methodology was transferred to a C++ version and combined with OpenCV tools to 3D-locate the center of a red and blue balloon from a set of camera images with known location and orientation.

References

[1] Humphreys, T 2023, *Aerial Robotics Course Notes for ASE 479W/ASE 389*, lecture notes, Aerial Robotics ASE 389, University of Texas at Austin, delivered 10 January 2023.

[2] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, vol. 2. Cambridge University Press, 2000.

Appendix A: Code

A.1 balloonfinder.cc

```
#include "balloonfinder.h"

#include <Eigen/Dense>
#include <cassert>
#include <opencv2/core/eigen.hpp>

#include "navtoolbox.h"

BalloonFinder::BalloonFinder(bool debuggingEnabled, bool
calibrationEnabled,
                                const Eigen::Vector3d& blueTrue_I,
                                const Eigen::Vector3d& redTrue_I) {
    debuggingEnabled_ = debuggingEnabled;
    calibrationEnabled_ = calibrationEnabled;
    blueTrue_I_ = blueTrue_I;
    redTrue_I_ = redTrue_I;
    V_.resize(3, 0);
    W_.resize(3, 0);
}

// Returns true if the input contour touches the edge of the input
// image;
// otherwise returns false.
//
bool touchesEdge(const cv::Mat& image, const std::vector<cv::Point>&
contour) {
    const size_t borderWidth = static_cast<size_t>(0.01 * image.rows);

    for (const auto& pt : contour) {
        if (pt.x <= borderWidth || pt.x >= (image.cols - borderWidth) ||
            pt.y <= borderWidth || pt.y >= (image.rows - borderWidth))
            return true;
    }
    return false;
}

Eigen::Vector3d BalloonFinder::eCB_calibrated() const {
    using namespace Eigen;
    const SensorParams sp;
    const size_t N = V_.cols();
    if (N < 2 || !calibrationEnabled_) {
        return Vector3d::Zero();
    }
    const VectorXd aVec = VectorXd::Ones(N);
    const Matrix3d dRCB = navtbx::wahbaSolver(aVec, W_, V_);
    const Matrix3d RCB = navtbx::euler2dc(sp.eCB());
    return navtbx::dc2euler(dRCB * RCB);
}
```

```

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) {
    using namespace cv;
    bool returnValue = false;
    rxVec->clear();
    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
    Mat framep;
    GaussianBlur(*image, framep, Size(21, 21), 0, 0);

    //
    ****
    //
    // Implement the rest of the function here. Your goal is to find a
balloon
    // of the color specified by the input 'color', and find its center
in image
    // plane coordinates (see the comments below for a discussion on
image plane
    // coordinates), expressed in pixels. Suppose rx is an
Eigen::Vector2d
    // object that holds the x and y position of a balloon center in
such
    // coordinates. You can push rx onto rxVec as follows: rxVec-
>push_back(rx)
    //
    //
    ****

    cvtColor(framep, framep, COLOR_BGR2HSV);
    Scalar balloonColor;
    std::string balloon_color;
    switch(color){
        case BalloonColor::RED:{
            Scalar colorLower_l(0, 120, 100), colorLower_h(10,255,255);
            Scalar colorUpper_l(170, 120, 100), colorUpper_h(180,255,255);
            Mat mLower, mUpper; //image can be thought of as matrix of
pixels
            //mLower and mUpper will store 1 for every pixel in their
respective in ranges
            inRange(framep,colorLower_l,colorLower_h,mLower);
            inRange(framep,colorUpper_l,colorUpper_h,mUpper);
            framep = mLower | mUpper; //bit wise 'or' both matrices

```

```

        balloonColor = Scalar(0,0,255);
        balloon_color = "Red";
        break;
    }
    case BalloonColor::BLUE:{
        Scalar lower_blue(90,60,100), upper_blue(110,255,255);
        Mat mOut;
        inRange(framep,lower_blue,upper_blue,mOut);
        framep = mOut;
        balloonColor = Scalar(255,0,0);
        balloon_color = "Blue";
        break;
    }
    default:{
        throw std::runtime_error("Color value is not handled in this
program.");
    }
}

/*namedWindow("pre", WINDOW_NORMAL);
resizeWindow("pre", 1000, 1000);
imshow("pre", *image);*/

// Erode image to eliminate stray wisps of color
constexpr int iterations = 10;
erode(framep, framep, Mat(), cv::Point(-1,-1), iterations);
// Dilate image to restore balloon to original size
dilate(framep, framep, Mat(), cv::Point(-1,-1), iterations);

/*namedWindow("post", WINDOW_NORMAL);
resizeWindow("post", 1000, 1000);
imshow("post", framep);
waitKey(0);*/

// Find contours
std::vector<std::vector<cv::Point>> contours;
std::vector<Vec4i> hierarchy;
findContours(framep, contours, hierarchy, RETR_EXTERNAL,
CHAIN_APPROX_SIMPLE);
// Loop through the contours. Bound each contour by both a bounding
// (rotated) rectangle and a minimum enclosing circle. Draw the
contour on
// the original image. If the bounding rectangle has aspect ratio
close to balloon
// and if the enclosing circle is large enough, then also
// draw the minimum enclosing circle.
RNG rng(12345);
Point2f center;
float radius;
float aspectRatio;
constexpr float maxAspectRatio = 1.45;
constexpr float minAspectRatio = 1.18;

```

```

constexpr float maxRadius = 300;
constexpr float minRadius = 50;
constexpr int minPointsFor_fitEllipse = 5;

if (debuggingEnabled_){
    std::cout << balloon_color << " testing:" << std::endl;
}

for (size_t ii = 0; ii < contours.size(); ii++) {
    if (touchesEdge(framep, contours[ii])){
        continue;
    }
    const Scalar color =
        Scalar(rng.uniform(0, 256), rng.uniform(0, 256),
rng.uniform(0, 256));
    minEnclosingCircle(contours[ii], center, radius);
    aspectRatio = maxAspectRatio;
    if (contours[ii].size() >= minPointsFor_fitEllipse) {
        RotatedRect boundingRectangle = fitEllipse(contours[ii]);
        const Size2f rectSize = boundingRectangle.size;
        aspectRatio =
            static_cast<float>(std::max(rectSize.width,
rectSize.height)) /
            std::min(rectSize.width, rectSize.height);
    }

    if (aspectRatio < maxAspectRatio && aspectRatio > minAspectRatio
        && radius > minRadius && radius < maxRadius) {

        if (debuggingEnabled_){
            circle(original, center, static_cast<int>(radius),
balloonColor, 8);
            circle(original, center, static_cast<int>(radius/6),
balloonColor, 8);
            std::cout << balloon_color << " balloon found" << ", cx: " <<
center.x << ", cy: " << center.y << std::endl;
        }

        Eigen::Vector2d rx;
        rx(0) = nCols_m1 - center.x;
        rx(1) = nRows_m1 - center.y;
        rxVec->push_back(rx);
        returnValue = true;
    }

    if (debuggingEnabled_){
        drawContours(original, contours, ii, color, 8, LINE_8,
hierarchy, 0);
        std::cout << "aspectRatio: " << aspectRatio << ", radius: " <<
radius << std::endl;
    }
}

```

```

    }

    // The debugging section below plots the back-projection of true
    balloon 3d
    // location on the original image. The balloon centers you find
    should be
    // close to the back-projected coordinates in xc_pixels. Feel free
    to alter
    // the debugging section below, or add other such sections, so you
    can see
    // how your found centers compare with the back-projected centers.
    if (debuggingEnabled_) {
        // Clone the original image for debugging purposes
        //original = image->clone();
        Eigen::Vector2d xc_pixels;
        Scalar trueProjectionColor;
        if (color == BalloonColor::BLUE) {
            xc_pixels = backProject(RCI, rc_I, blueTrue_I_);
            trueProjectionColor = Scalar(255, 0, 0);
        } else {
            xc_pixels = backProject(RCI, rc_I, redTrue_I_);
            trueProjectionColor = Scalar(0, 0, 255);
        }

        Point2f true_center;
        // The image plane coordinate system, in which xc_pixels is
        expressed, has
        // its origin at the lower-right of the image, x axis pointing
        left and y
        // axis pointing up, whereas the variable 'center' below, used by
        OpenCV
        // for plotting on the image, is referenced to the image's top
        left corner
        // and has the opposite x and y directions. The measurements
        returned in
        // rxVec should be given in the image plane coordinate system like
        // xc_pixels. Hence, once you've found a balloon center from your
        image
        // processing techniques, you'll need to convert it to the image
        plane
        // coordinate system using an inverse of the mapping below.
        true_center.x = nCols_m1 - xc_pixels(0);
        true_center.y = nRows_m1 - xc_pixels(1);
        circle(original, true_center, 10, trueProjectionColor, FILLED);
        namedWindow("Display", WINDOW_NORMAL);
        resizeWindow("Display", 1000, 1000);
        imshow("Display", original);

        //namedWindow("Display1", WINDOW_NORMAL);
        //resizeWindow("Display1", 1000, 1000);
        //imshow("Display1", framep);
    }
}

```

```

        if (returnValue){
            std::cout << "true cx: " << true_center.x << ", true cy: " <<
true_center.y << std::endl;
        }

        waitKey(0);
    }

    return returnValue;
}

void BalloonFinder::findBalloons(
    const cv::Mat* image, const Eigen::Matrix3d RCI, const
Eigen::Vector3d rc_I,
    std::vector<std::shared_ptr<const CameraBundle>>* bundles,
    std::vector<BalloonColor>* colors) {
    // Crop image to 4k size. This removes the bottom 16 rows of the
image,
    // which are an artifact of the camera API.
    const cv::Rect croppedRegion(0, 0, sensorParams_.imageWidthPixels(),
                                sensorParams_.imageHeightPixels());
    cv::Mat croppedImage = (*image)(croppedRegion);
    // Convert camera instrinsic matrix K and distortion parameters to
OpenCV
    // format
    cv::Mat K, distortionCoeffs, undistortedImage;
    Eigen::Matrix3d Kpixels = sensorParams_.K() /
sensorParams_.pixelSize();
    Kpixels(2, 2) = 1;
    cv::eigen2cv(Kpixels, K);
    cv::eigen2cv(sensorParams_.distortionCoeffs(), distortionCoeffs);
    // Undistort image
    cv::undistort(croppedImage, undistortedImage, K, distortionCoeffs);

    // Find balloons of specified color
    std::vector<BalloonColor> candidateColors = {BalloonColor::RED,
                                                BalloonColor::BLUE};

    for (auto color : candidateColors) {
        std::vector<Eigen::Vector2d> rxVec;
        if (findBalloonsOfSpecifiedColor(&undistortedImage, RCI, rc_I,
color,
                                &rxVec)) {
            for (const auto& rx : rxVec) {
                std::shared_ptr<CameraBundle> cb =
std::make_shared<CameraBundle>();
                cb->RCI = RCI;
                cb->rc_I = rc_I;
                cb->rx = rx;
                bundles->push_back(cb);
                colors->push_back(color);
            }
        }
    }
}

```

```

    }
}
}

```

A.2 estimate3dFeatureLocation.m

```

function [rXIHat,Px] = estimate3dFeatureLocation(M,P)
% estimate3dFeatureLocation : Estimate the 3D coordinates of a feature
point
%                               seen by two or more cameras with known
pose.
%
%
% INPUTS
%
% M ----- Structure with the following elements:
%
%     rxArray = 1xN cell array of measured positions of the feature
point
%               projection on the camera's image plane, in pixels.
%               rxArray{i} is the 2x1 vector of coordinates of the
feature
%               point as measured by the ith camera. To ensure the
%               estimation problem is observable, N must satisfy N
%               >= 2 and
%               at least two cameras must be non-collinear.
%
%     RCIArray = 1xN cell array of I-to-camera-frame attitude
matrices.
%               RCIArray{i} is the 3x3 attitude matrix corresponding
to the
%               measurement rxArray{i}.
%
%     rcArray = 1xN cell array of camera center positions.
rcArray{i} is
%               the 3x1 position of the camera center corresponding
to the
%               measurement rxArray{i}, expressed in the I frame in
meters.
%
% P ----- Structure with the following elements:
%
%     sensorParams = Structure containing all relevant parameters for the
quad's
%                   sensors, as defined in sensorParamsScript.m
rocB = P.sensorParams.rocB;
RCB = P.sensorParams.RCB;
Rc = P.sensorParams.Rc;
f = P.sensorParams.f;
pixelSize = P.sensorParams.pixelSize;
K = P.sensorParams.K;
imagePlaneSize = P.sensorParams.imagePlaneSize;
%

```

```

% OUTPUTS
%
%
N = length(M.rxArray);
if N<2
    error('Estimation Problem is not observable');
else
    H = [];
    RC = [];
    for ii = 1:N
        xtilde_i = [pixelSize*M.rxArray{ii};1];

        RCIi = M.RCIArray{ii};
        ti = -RCIi*M.rcArray{ii};
        I2CMati = [RCIi,ti;0 0 0 1];
        Pi = [K,[0;0;0]]*I2CMati;

        H_i = [xtilde_i(1)*Pi(3,:) - Pi(1,:);
                xtilde_i(2)*Pi(3,:) - Pi(2,:)];
        H = [H;H_i];

        RC = blkdiag(RC,Rc);

    end

    Hr = H(:,1:3);
    z = -1*H(:,4);
    R = pixelSize^2*RC;

% rXIHat ----- 3x1 estimated location of the feature point
% expressed in I
%               in meters.
    rXIHat = inv(Hr'*inv(R)*Hr)*Hr'*inv(R)*z;
%
% Px ----- 3x3 error covariance matrix for the estimate rxIHat.

    Px = inv(Hr'*inv(R)*Hr);

end

```

A.3 simulateQuadrotorEstimationAndControl.m

```

function [Q,Ms] = 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).
%
%
% Ms ----- Structure with the following elements:
%
%           rxArray = 1xN cell array of measured positions of the feature
point
%           projection on the camera's image plane, in pixels.
%           rxArray{i} is the 2x1 vector of coordinates of the
feature
%           point as measured by the ith camera. To ensure the
%           estimation problem is observable, N must satisfy N
>= 2 and
%           at least two cameras must be non-collinear.
%
%           RCIArray = 1xN cell array of I-to-camera-frame attitude
matrices.
%           RCIArray{i} is the 3x3 attitude matrix corresponding
to the
%           measurement rxArray{i}.
%
%           rcArray = 1xN cell array of camera center positions.
rcArray{i} is
%           the 3x1 position of the camera center corresponding
to the
%           measurement rxArray{i}, expressed in the I frame in
meters.
%
%+-----+
-----+
% References:
%
%
% Author: Justin Hart
%+=====+
=====+

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 = [];
EMat = []; TMat = []; %estimated and true states respectively
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;
    TMat = [TMat;Sm];
    % 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);
    EMat = [EMat;E];
    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)]';
[tVec,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
Q.state.RBIMat = XMat(:,7:15);

%Structure Computation
SCMat = TMat; %TMat for true state, EMat for estimated state
rXI = [0;0;0.7];

% Images close together
% image_span = (N-500)/100;
% N=(N-500)/10+500;
% Images far apart
image_span = (N-500)/10;

```

```

kk = 1;
for ii=500:image_span:N-1
    rx = hdCameraSimulator(rXI,SCMat(ii),P);
    if isempty(rx)
        rxArray{kk} = [NaN;NaN];
    else
        rxArray{kk} = rx;
    end

    RCI = P.sensorParams.RCB*SCMat(ii).statek.RBI;
    RCIArray{kk} = RCI;

    rc = SCSMat(ii).statek.rI+SCMat(ii).statek.RBI'*P.sensorParams.roCB;
    rcArray{kk} = rc;

    kk = kk+1;
end

Ms.rxArray = rxArray;
Ms.RCIArray = RCIArray;
Ms.rcArray = rcArray;

```

A.4 structurecomputer.cc

```

#include "structurecomputer.h"

#include <Eigen/LU>

void pr(Eigen::MatrixXd m) { std::cout << m << std::endl; }
void pr(Eigen::VectorXd m) { std::cout << m << std::endl; }
void pr(Eigen::Matrix3d m) { std::cout << m << std::endl; }
void pr(Eigen::Vector3d m) { std::cout << m << std::endl; }
void pr(Eigen::Vector2d m) { std::cout << m << std::endl; }

Eigen::Vector2d backProject(const Eigen::Matrix3d& RCI,
                           const Eigen::Vector3d& rc_I,
                           const Eigen::Vector3d& X3d) {
    using namespace Eigen;
    Vector3d t = -RCI * rc_I;
    MatrixXd Pextrinsic(3, 4);
    Pextrinsic << RCI, t;
    SensorParams sp;
    MatrixXd Pc = sp.K() * Pextrinsic;
    VectorXd X(4, 1);
    X.head(3) = X3d;
    X(3) = 1;
    Vector3d x = Pc * X;
    Vector2d xc_pixels = (x.head(2) / x(2)) / sp.pixelSize();
    return xc_pixels;
}

```

```

Eigen::Vector3d pixelsToUnitVector_C(const Eigen::Vector2d& rPixels) {
    using namespace Eigen;
    SensorParams sp;
    // Convert input vector to meters
    Vector2d rMeters = rPixels * sp.pixelSize();
    // Write as a homogeneous vector, with a 1 in 3rd element
    Vector3d rHomogeneous;
    rHomogeneous.head(2) = rMeters;
    rHomogeneous(2) = 1;
    // Invert the projection operation through the camera intrinsic
    matrix K to
    // yield a vector rC in the camera coordinate frame that has a Z
    value of 1
    Vector3d rC = sp.K().lu().solve(rHomogeneous);
    // Normalize rC so that output is a unit vector
    return rC.normalized();
}

void StructureComputer::clear() {
    // Zero out contents of point_
    point_.rXIHat.fill(0);
    point_.Px.fill(0);
    // Clear bundleVec_
    bundleVec_.clear();
}

void StructureComputer::push(std::shared_ptr<const CameraBundle>
bundle) {
    bundleVec_.push_back(bundle);
}

// This function is where the computation is performed to estimate the
// contents of point_. The function returns a copy of point_.
//
Point StructureComputer::computeStructure() {
    // Throw an error if there are fewer than 2 CameraBundles in
    bundleVec_,
    // since in this case structure computation is not possible.
    if (bundleVec_.size() < 2) {
        throw std::runtime_error(
            "At least 2 CameraBundle objects are "
            "needed for structure computation.");
    }

    //
    *****
    // Fill in here the required steps to calculate the 3D position of
    the
    // feature point and its covariance. Put these respectively in
    // point_.rXIHat and point_.Px

```

```

//
*****
using namespace Eigen;
SensorParams sp;

int n = bundleVec_.size();
MatrixXd H(2*n,4);
for(int ii = 0;ii < n; ii++){
    Vector2d xtilde = bundleVec_[ii]->rx*sp.pixelSize();
    //xtilde[3] = 1;
    Vector3d t = -bundleVec_[ii]->RCI * bundleVec_[ii]->rc_I;
    MatrixXd Pextrinsic(3, 4);
    Pextrinsic << bundleVec_[ii]->RCI, t;
    MatrixXd Pc = sp.K() * Pextrinsic;
    VectorXd P1 = Pc.row(0);
    VectorXd P2 = Pc.row(1);
    VectorXd P3 = Pc.row(2);
    H.row(ii*2) = xtilde[0]*P3 - P1;
    H.row(ii*2+1) = xtilde[1]*P3 - P2;
}

MatrixXd Hr = H.block(0,0,2*n,3);
VectorXd z = -1 * H.block(0,3,2*n,1);

MatrixXd RC = MatrixXd::Zero(2*n,2*n);
for(int jj = 0;jj < n;jj++){
    RC.block(jj*2,jj*2,2,2) = sp.Rc();
}

MatrixXd R = pow(sp.pixelSize(),2)*RC;
constexpr size_t nx = 3;
VectorXd xHat(nx);
MatrixXd Px(nx,nx);
MatrixXd Rinv = R.inverse();
xHat = (Hr.transpose()*Rinv*Hr).ldlt().solve(Hr.transpose()*Rinv*z);
Px = (Hr.transpose()*Rinv*Hr).inverse();

point_.rXIHat = xHat;
point_.Px = Px;

return point_;
}

```

A.5 topSimulateControl.m

```

% 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,Ms] = simulateQuadrotorEstimationAndControl(R,S,P);
else
    [Q,Ms] = simulateQuadrotorControl(R,S,P);
end

% Estimate 3D feature location
[rXIHat,Px] = estimate3dFeatureLocation(Ms,P);

% Compare rXI and rXIHat
disp('rXI compared with rXIHat (should be within a few cm): ');
[S.rXIMat(2,:) ' rXIHat]

% Examine diagonal elements of Px
disp(['Sqrt of diagonal elements of Px (errors in rXIHat should not be
much ' ...
    'bigger than these numbers): ']);
sqrt(diag(Px))

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);

%get vector from CM in direction of xI projected onto XY plane
lg = length(Q.state.eMat);
xIvec = [];
for mm = 1:lg
    RBIk = euler2dcm(Q.state.eMat(mm,:));
    xk = RBIk*[1;0;0];
    xIvec = [xIvec;xk];
end

figure(1);clf;
plot(Q.tVec,Q.state.rMat(:,3),'LineWidth',1.5); 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,'LineWidth',1.5);
grid on;
xlabel('Time (sec)');
ylabel('\Delta \psi (rad)');

```

```

title('Yaw angle error');

figure(2);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2), 'LineWidth',1.5);
axis equal; grid on;hold on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
quiver(Q.state.rMat(:,1), Q.state.rMat(:,2),xIvec(:,1),xIvec(:,2));

%calculate performance
xerror = Q.state.rMat(1:2:end,1)-R.rIstar(:,1);
meanxerr = round(mean(xerror));
yerror = Q.state.rMat(1:2:end,2)-R.rIstar(:,2);
meanyerr = round(mean(yerror));
zerror = Q.state.rMat(1:2:end,3)-R.rIstar(:,3);
meanzerr = round(mean(zerror));
figure(4);clf;
plot(tVec,xerror-meanxerr,tVec,yerror-meanyerr,tVec,zerror-
meanzerr, 'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Error (m)');
title('Tracking Error Relative To Desired Trajectory');
legend('x','y','z');

```