MAAV Vehicle Documentation

MAAV

Contents

1 Introduction

MAAV's vehicle is complicated. This document aims to be both an overview of the vehicle as well as a "deep dive" into the design, math, and algorithms of the vehicle. Conventions used in this document are given in Table 1.

Table 1: Document Conventions

x	arena frame x position
y	arena frame y position
z	arena frame z position
$\dot{x}$	arena frame x velocity
$\dot{y}$	arena frame y velocity
$\dot{z}$	arena frame z velocity
$\ddot{x}$	arena frame x acceleration
$\ddot{y}$	arena frame y acceleration
$\ddot{z}$	arena frame z acceleration
x_B	vehicle frame x position
y_B	vehicle frame y position
z_B	vehicle frame z position
$\dot{x}_B$	vehicle frame x velocity
$\dot{y}_B$	vehicle frame y velocity
$\dot{z}_B$	vehicle frame z velocity
$\ddot{x}_B$	vehicle frame x acceleration
$\ddot{y}_B$	vehicle frame y acceleration
$\ddot{z}_B$	vehicle frame z acceleration
ϕ	roll
θ	pitch
ψ	yaw

2 Reference Frames

2.1 Arena Reference Frame

Looking at the arena from above, view the green goal line on the bottom, red line on the top and white out-of-bounds lines on either side. The global x-axis points up along the white lines from the goal line to the red line. The global y-axis points right, and the global z-axis points down into the ground. The origin of the world is the bottom-left corner (i.e. the intersection of the goal line with the left boundary line). This follows the right-hand-rule used to derive the control and state estimation algorithms. The global reference frame is given in Figure 1.

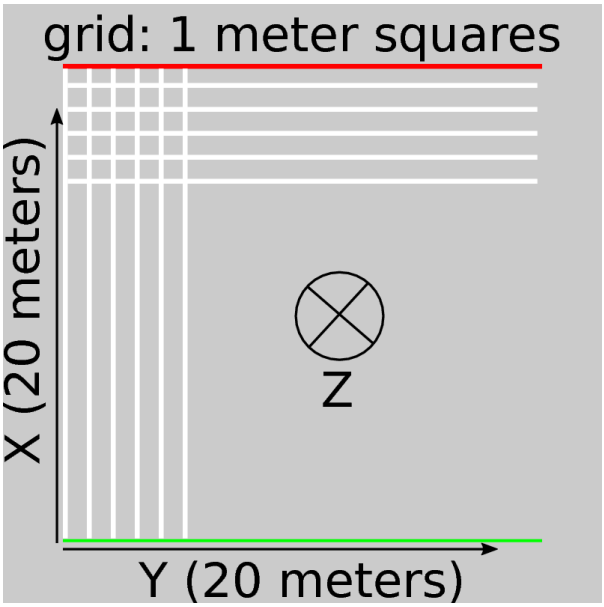

Figure 1: Global/Arena Reference Frame

All control algorithms are derived in this world frame (e.g. waypoints are given w.r.t. the world and any necessary frame changes for sensor values are performed to give feedback to the controller in the world frame).

2.2 Euler Angles

We now define ϕ as the rotation about the x-axis, θ as the rotation about the y-axis, and ψ as the rotation about the z-axis. Positive values are defined by the right-hand-rule – stick your thumb along the positive axis and the fingers curl in the positive direction of rotation.

2.3 Vehicle Reference Frame

The vehicle's x-axis points out of its front. It's y-axis points out the right side. The z-axis points down out of the vehicle. This is common for many control algorithms in aerospace.

The global and body reference frames were chosen such that at a roll, pitch, and yaw of 0 radians, the frames converge. Also, the IMU on the vehicle is physically mounted such that its reference frame is exactly the same as the vehicle's local frame. Thus, the only frame changes that are necessary are between the local vehicle frame and the global frame. These frame changes can be easily determined using the rotation matrix obtained from the Microstrain IMU when it is polled for a sensor reading which is created from filtered measurements.

2.4 Using a Rotation Matrix

Let $\boldsymbol{R}$ be a 3x3 matrix describing the rotation from the vehicle to the arena frame that is parameterized by the 3 Euler angles of ϕ, θ, and ψ.

Given a 3D vehicle frame vector $\boldsymbol{v}_B = [x_B\, y_B\, z_B]^T$ (or their derivatives), we get the 3D arena frame vector $\boldsymbol{v} = [x\, y\, z]^T$ by the equation:

$$\boldsymbol{v} = \boldsymbol{R}\boldsymbol{v}_B \tag{1}$$

Given $\boldsymbol{v}$, we can get $\mathbf{v_B}$ as follows:

$$\boldsymbol{v}_B = \boldsymbol{R}^{-1}\boldsymbol{v} = \boldsymbol{R}^T\boldsymbol{v} \tag{2}$$

Since rotation matrices are orthogonal, their inverse is their transpose, so we can computationally simplify Equation 2 by using $\boldsymbol{R}^T$.

3 Sensor Suite

The sensors used on the vehicle are described in the following subsections. Each subsection describes the data given by the sensor as well as the experimentally observed noise.

3.1 LiDAR Lite

3.1.1 Purpose and Measurements

The LiDAR Lite is a laser distance measurement sensor used for measuring the altitude of the vehicle.

It senses z_B and thus needs a frame rotation to transform it into z as shown as follows:

$$z = z_B \cos\phi \cos\theta \tag{3}$$

where ϕ and θ are extracted from IMU measurements obtained at the same timestep as the LiDAR measurement. The current sample rate from the LiDAR is 50 Hz and is communicated to from the Tiva MCU via I2C.

3.1.2 Noise Characterization

This section provides the LiDAR noise characterization at three different distances. For each distance, 10000 data points were measured and the mean, stadard deviation, and variance were calculated. The results are provided in Table 2.

Table 2: LiDAR Lite Data Characteristics

Actual Distance (cm)	Mean Measurement (cm)	Std. Dev. (cm)	Variance (cm^2)
52.00	51.507	0.61711	0.38083
95.00	95.931	0.66139	0.43743
139.00	140.76	0.68728	0.47236

The errors between the actual distance and the mean measurement recorded by the LiDAR Lite are within the +/-2.5cm error specified in the product documentation[4].

3.2 Logitech c920

3.3 Microstrain 3DM-GX3-25 IMU

3.3.1 Purpose and Measurements

The IMU incorporates a 3degrees-of-freedom (DOF) accelerometer, 3DOF gyroscope, and a 3DOF magnetometer (compass) along with a microprocessor into one unit.

It is used to provide measurements for $\ddot{x}_B, \ddot{y}_B, \ddot{z}_B, \omega_x, \omega_y, \omega_z, \phi, \theta$, and ψ, where ω_x, ω_y, and ω_z are the angular rates about the body frame axes of the vehicle (note that these are *not* $\dot{\phi}, \dot{\theta}$, and $\dot{\psi}$).

We communicate with the IMU via a serial (UART-RS232) connection and obtain raw values for vehicle-frame accelerations and angular rates as well as a filtered/processed rotation matrix from the previous measurement frame to the current measurement frame. From this matrix of values, filtered by the IMU's onboard processor doing Kalman Filtering, we extract the actual ϕ, θ, and ψ of the vehicle which are then used to rotate the body-frame sensor values into the arena frame in which the control algorithm operates in.

3.3.2 Noise Characterization

Data was collected from the IMU while resting on a level table. Table 3 shows the mean, standard deviation, and variance for each type of measurement for 12000 data points.

Table 3: IMU Data Characteristics

Raw Accelerometer Measurments			
Type	Mean (g)	Std. Dev. (g)	Variance (g^2)
$\ddot{x}_B$	0.003367	7.0405e-04	4.49569e-07
$\ddot{y}_B$	-0.012307	9.3020e-04	8.6528e-07
$\ddot{z}_B$	1.001700	0.0031851	1.0145e-05
Raw Gyroscope Measurements			
Type	Mean (rad/s)	Std. Dev. (rad/s)	Variance (rad^2/s^2)
ω_x	-4.3443e-04	0.0026131	6.8284e-06
ω_y	0.0031502	0.0025510	6.5076e-06
ω_z	-0.0016228	0.0025015	6.2576e-06
Filtered Euler Angle Measurements			
Type	Mean (rad)	Std. Dev. (rad)	Variance (rad^2)
ϕ	-0.0060886	7.5494e-04	5.6994e-07
θ	3.1279	4.8600e-04	2.3620e-07
ψ	1.7800	0.011527	1.3288e-04

Table 4: Hokuyo URG-04LX-UG01 Spec's

Measuring Area	20 to 5600mm, 240^o
Step Angle	0.36^o
Accuracy for:	
60 to 1000mm	+/- 30mm
1000 to 4095mm	+/- 3% of measurement
Scanning Time	100ms

3.4 Pixhawk Px4Flow

3.4.1 Purpose and Measurements

3.4.2 Noise Characterization

3.5 Hokuyo URG-04LX-UG01

3.5.1 Purpose and Measurements

3.5.2 Noise Characterization

4 Microcontrollers and Processors

4.1 Atmel ATMEGA xx

This 8-bit microcontroller (MCU) runs the killswitch code. It reads in two pulse width modulation (PWM) signals; one from the throttle of our RC controller, and one from a shoulder button on the controller. The signal coming from the throttle tells the vehicle whether to enable or disable the power to the motors. The signal from the shoulder button toggles the vehicle between a completely autonomous mode and an assisted autonomous mode. The ATMEGA toggles a general purpose input output (GPIO) which drives four P-Channel MOSFET (PFET)s, which are responsible for putting the vehicle in a "killed" state.

4.2 Axiomtek PICO831-N2800

4.2.1 Physical Characteristics

This system on chip (SoC)[2] runs the navigation code. It has 2 cores running at 1.8GHz, 4GiB random access memory (RAM), and a 64GiB solid state drive (SSD). A list of model numbers for the hardware is given in Table 5.

Table 5: AxiomTek PICO831 Hardware Model Numbers

Part	**Model Number**
RAM	G.Skill F3-8500CL7S-4GBSQ
SSD	Crucial CT064M4SSD3
WiFi	Intel 633ANHMW

The PICO831 requires heatsinking, although the stock heatsink shipped by Axiomtek is overkill. The heatsink designed by Michigan Autonomous Aerial Vehicles (MAAV) reduces mass by around 60%, yet maintains similar thermal properties due to increased surface area and the plentiful flow of room temperature air in the system's operational environment.

4.2.2 System Administration

The SoC[2] runs the Ubuntu 16.04 operating system. This is mainly for access to the network stack implementation and to run the navigation software. Since Ubuntu 16.04 uses systemd for booting and managing system services, MAAV uses systemd to manage the starting, stopping, and logging for the navigation software. Instructions for deploying the navigation software are contained in the navigation source code repository, but common administration operations are given here. Instructions for using SSH to obtain a shell on the Operating System (OS) running on the PICO831 are also given in the navigation source code repository. In order to view the current status of the navigation software, start the navigation software, or stop the navigation software, use 'systemctl' as shown in Figure 4.2.2.

```
# Check the status (i.e. running or stopped) of the software
sudo systemctl status maav-nav.service

# Stop the software
sudo systemctl stop maav-nav.service

# Start the software
sudo systemctl start maav-nav.service

# Restart the software
sudo systemctl restart maav-nav.service
```

In order to view the current log output of the navigation software, use 'journalctl', as shown in Figure 4.2.2.

Developers should utilize the systemd documentation[5] for more information on using systemd to manage the vehicle properly. In addition,

```
# Open up the log output, going directly to the end
sudo journalctl -e -u maav-nav.service
```

utilize the Debian Handbook[1] for information on properly managing a Debian or Debian-derived system like Ubuntu.

4.3 Texas Instruments TM123xx

This MCU is an ARM Cortex-M4F, which was chosen because of its integrated floating point unit (FPU) for flight control. It is responsible for taking in all sensor data inputs, processes them, sends commands to the DJI attitude controller, and relay back relevant information about the quadrotor state to the Atom. The data in question may change with each iteration of the vehicle. The Tiva is responsible for all control and state estimation algorithms.

4.3.1 Pinout for MAAV Spine Circuit Board

Table 6 shows the pinout of the Tiva and components connected to it for the 2016 verison of the circuit board on the vehicle. Table 7 shows the pinout for the summer 2015 version of the circuit board which is still in use for flight testing.

5 Electrical System

5.1 Power System

The vehicle is powered by a Thunder Power TP6600-4SP+25 lithium polymer (LiPo) battery.

6 Control Software

This section describes the flight modes and the two main algorithms implemented in the controller that make the vehicle fly–the PID controller, a feedback controller that commands the various inputs to the DJI Naza-M Lite attitude controller (which in turn commadns the motors), and the Kalman filter which filters and fuses the various sensor readings to estimate the final state of the quadrotor. This final

Table 6: 2016 Tiva-Spine Pinout

Pin	Label	Function	Component
47	PB2	I2C0 SCL	Px4, Lidar
48	PB3	I2C0 SDA	
61	PD0	SSI1 CLK	SD Card
62	PD1	SSI1 FSS (CS)	
63	PD2	SSI1 RX (MISO)	
64	PD3	SSI1 TX (MOSI)	
58	PB4	SSI2 CLK	Light Board
57	PB5	SSI2 FSS (CS)	
1	PB6	SSI2 RX (MISO)	
4	PB7	SSI2 TX (MOSI)	
17	PA0	UART0 RX	Atom
18	PA1	UART0 TX	
16	PC4	UART1 RX	IMU
15	PC5	UART1 TX	
49	PC3	TDO	JTAG
50	PC2	TDI	
51	PC1	TMS	
52	PC0	TCK	
9	PE0	ADC AIN3	Battery
59	PE4	DJI PPM (M0 PWM4)	DJI
19	PA2	RC 1 (Pitch)	Pilot RC
20	PA3	RC 2 (Roll)	
21	PA4	RC 3 (Thrust)	
22	PA5	RC 4 (Yaw Rate)	
23	PA6	RC 5 (Manual Button)	
24	PA7	RC 6	
14	PC6	KILL 3 (Kill Signal)	Kill RC
13	PC7	KILL 5 (Mode)	
10	PD7	SOFT KILL	Kill Switch
43	PD4	OPT SW1	Switches
44	PD5	OPT SW2	
53	PD6	OPT SW3	
29	PF1	RED LED	LED
30	PF2	BLUE LED	
31	PF3	GREEN LED	

state is the feedback sent to the controller and to the navigation software to be used as the actual state of the vehicle in various algorithms and decision-making logic.

Table 7: 2015 Tiva-Spine Pinout

Pin	Label	Function	Component
47	PB2	I2C0 SCL	Px4, Lidar
48	PB3	I2C0 SDA	
61	PD0	SSI1 CLK	SD Card & Light Board
62	PD1	SSI1 FSS (Test Point)	
63	PD2	SSI1 RX (MISO)	
64	PD3	SSI1 TX (MOSI)	
43	PD4	SD Card CS	
10	PD7	Light Board CCS	
17	PA0	UART0 RX	Atom
18	PA1	UART0 TX	
16	PC4	UART1 RX	IMU
15	PC5	UART1 TX	
49	PC3	TDO	JTAG
50	PC2	TDI	
51	PC1	TMS	
52	PC0	TCK	
58	PB4	ADC0 CH10	Battery
1	PB6	DJI PPM (M0 PWM4)	DJI
19	PA2	RC 1 (Pitch)	Pilot RC
20	PA3	RC 2 (Roll)	
21	PA4	RC 3 (Thrust)	
22	PA5	RC 4 (Yaw Rate)	
23	PA6	RC 5 (Manual Button)	
24	PA7	RC 6	
59	PE4	KILL 1	Kill RC
60	PE5	KILL 2	
6	PE3	KILL 3 (KILL SIG)	
7	PE2	KILL 4	
8	PE1	KILL 5 (Mode)	
9	PE0	KILL 6	
5	PF4	OPT SW1	Switches
45	PB0	OPT SW2	
46	PB1	OPT SW3	
29	PF1	RED LED	LED
30	PF2	BLUE LED	
31	PF3	GREEN LED	
8	PE1	DJI Motor 1	Output from DJI
7	PE2	DJI Motor 2	
6	PE3	DJI Motor 3	
60	PE5	DJI Motor 4	

6.1 Flight Modes

The controller supports three different flight modes, Manual, Assisted, and Autonomous. These modes can be toggled using the shoulder button and central knob on the Pilot RC controller. The Pilot RC controller shoulder button toggles between Manual and Assisted/Autonomous, while the Pilot RC controller center knob toggles between Assisted and Autonomous when the shoulder button is in Assisted/Autonomous Mode.

6.1.1 Manual Mode

In Manual mode, the RC input is passed straight through to the DJI. The LED will be solid green in Manual mode.

6.1.2 Assisted Mode

In Manual mode, the RC input for Z is treated as a desired height, the RC input for Roll and Pitch are treated as a desired X and Y rates, and the RC input for Yaw is passed straight through to the DJI. The LED will be solid red in Assisted Mode. Using the onboard DIP switches, an alternate Assisted Mode can be activate in which Z is treated as desired height and Roll and Pitch are passed straight through to the DJI.

6.1.3 Autonomous Mode

In Autonomous mode, the RC input is ignored in favor of commands from the Navigation board. The LED will be solid blue in Autonomous mode.

6.2 PID Control

PID stands for Proportional-Integral-Derivative control. The generic PID algorithm is laid out as follows. Given a desired target state for the system, called a *setpoint*, x_s and the current state of the system x, the PID controller minimizes the error e between the setpoint and current state and produces a control output u to enact upon the system at time t as follows:

$$e(t) = x_s(t) - x(t) \tag{4}$$

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \dot{e}(t) \tag{5}$$

where K_P, K_I, and K_D are the proportional (P), integral (I), and derivative (D) gains, respectively. These are tunable hyperparameters for the algorithm that are set during an empirical tuning process. There are different versions of this control law that have been used in practice. One such version involves the substitution of the time derivative of the state $\dot{x}$ instead of using $\dot{e}$ to reduce the noise in the error derivative that occurs near zero-crossings. This is a strategy that is used in MAAV's control software, and results in the following modification to equation 5:

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \dot{x}(t) \tag{6}$$

This algorithm is used in many instances to control the full quadrotor–one for each value and rate within the x, y, z DOFs as well as for the state of the ψ DOF. Within each DOF, the PID instances are cascaded such that the value PID output becomes the setpoint of the rate PID, and the rate PID's output (which is a force in that DOF) is used to calculate the final commands to the DJI.

6.3 Kalman Filtering

7 Navigation Software

MAAV's navigation software controls the vehicle at a high level, planning the mission, observing the state of the arena, and accepting and applying user control. The material in this section covers the theoretical parts of the software. If you are trying to deploy and run software on the vehicle, see 4.2.

7.1 Field of View Transformation

This section describes the mathematics behind the transformation between the different fields of views for the camera. The math is quite simple and only requires high school trigonometry.

7.1.1 Camera Fields of View

Let us first define what we mean by fields of view (FOV). The word "field" is somewhat confusing, as we will see, since these quantities are actually angles, not lengths.

There are three FOVs that we are concerned with: the horizontal, vertical, and diagonal FOVs. We will define the horizontal FOVs in detail here; the definition for the rest are analogous. We also refer to the horizontal FOV as the x FOV, and the vertical FOV as the y FOV.

Consider Figure 2. Point P is the location of the camera. Let point O be a point along the center of the camera's view. Plane $ABCD$ contains point O and is perpendicular to OP. Also, we let plane $ABCD$ be the largest plane that fits in the camera's view. In other words, if we were to let the camera take an image right now, AB, BC, CD, DA will be the right, bottom, left, and top edge of the image, respectively.

Let FH be a line containing O that is parallel to the horizontal plane. Let EG be a line containing O that is parallel to the vertical plane. From simple high school geometry, we know that E, F, G, H are midpoints of AD, AB, BC, CD, respectively. We then define the horizontal FOV of the camera, f_x, be the angle $\angle HPF$.

Similarly, we define the vertical FOV f_y as the angle $\angle EPG$ (Figure 3) and the diagonal FOV f_d as the angle $\angle DPB$ (Figure 4).

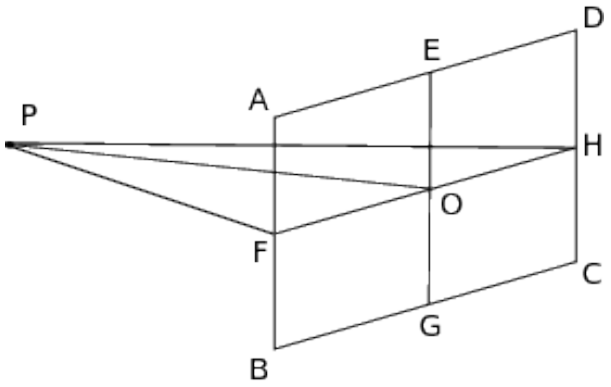

Figure 2: Camera x FOV

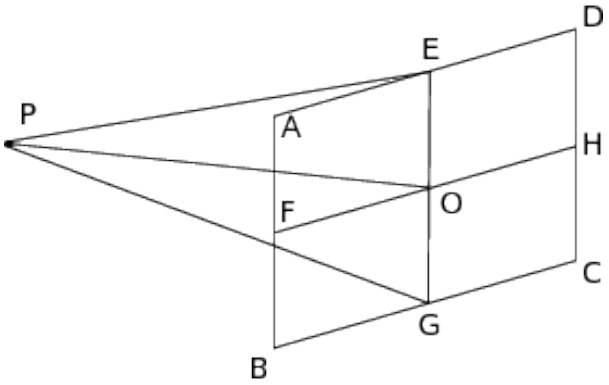

Figure 3: Camera y FOV

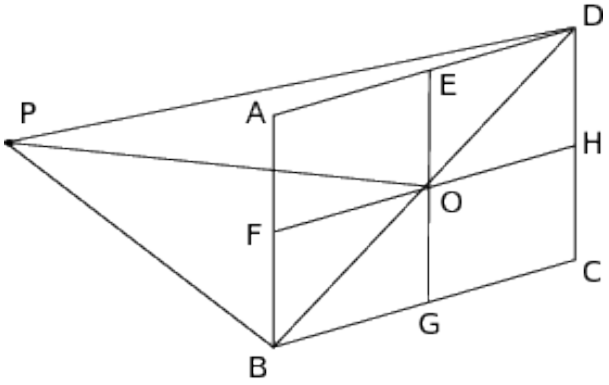

Figure 4: Camera Diagonal FOV

7.1.2 Transformation Algorithm

Once the definitions are clear, it should be easy to see that we can relate f_x, f_y, and f_d by the following equation:

$$f_d = 2 \arctan\left(\sqrt{\tan^2\left(\frac{1}{2}f_x\right) + \tan^2\left(\frac{1}{2}f_y\right)}\right) \quad (7)$$

Also, if we know the length w of AD, the length h of AB, as well as f_d, we can find f_x and f_y using the

following equations:

$$f_x = 2\arctan\left(\frac{w\tan\left(\frac{1}{2}f_d\right)}{\sqrt{w^2+h^2}}\right) \tag{8}$$

$$f_y = 2\arctan\left(\frac{h\tan\left(\frac{1}{2}f_d\right)}{\sqrt{w^2+h^2}}\right) \tag{9}$$

7.2 Frame of Reference Mapping

7.2.1 Introduction

This section describes the algorithm that the navigation software uses to transform two dimensional camera coordinates to three dimensional coordinates in the vehicle's reference frame. First, the algorithm corrects the radial distortion from the camera. Second, the algorithm generates a unit vector in the x direction of the vehicle reference frame. Third, the camera rotates this vector down to the coordinate of interest in the image, assuming the image is centered on the vehicle reference frame x-axis. Next, the vector is rotated according to the vehicle's rotation matrix, and then again according to the camera's vehicle-relative rotation matrix. Finally, the position vector of the vehicle and of the camera relative to the vehicle are added. The result of this series of computations is the global coordinate of the object of interest in the given image from the camera.

7.2.2 Details

In general, the camera reference frame has $(0, 0)$ in the top left corner of the image, with x increasing to the right and y increasing down. The camera reference frame is shown in Figure 5.

The camera reference frame contains no height information at all. To deal with this lack of data, we make three assumptions. The first assumption is that the cameras mostly capture the ground, so $z = 0$. The second assumption is that every object of interest, for which the coordinates in vehicle reference frame are desired, is on the ground ($z = 0$). The third and final assumption is that the camera locations in the vehicle reference frame are known with a negligible error.

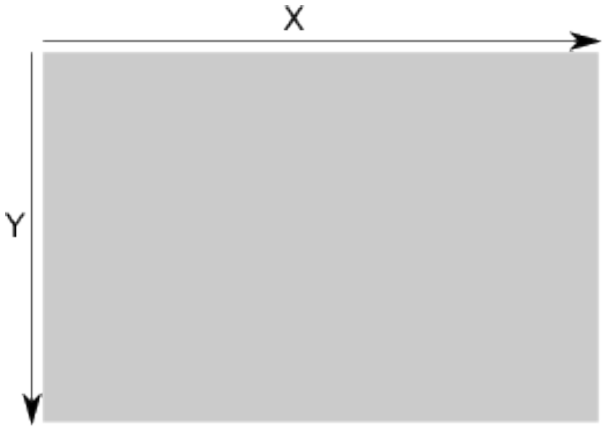

Figure 5: Camera Reference Frame

To find the coordinates in the vehicle reference frame, the navigation software creates a ray originating from the camera and going to the object of interest in the image.

Let the camera field of view, in radians, be f, with f_x the field of view along the x-axis, and f_y the same for the y-axis; the experimentally determined radial correction factors be given by c_1, c_2, c_3; the camera coordinates of the object of interest be x_c and y_c; the arena coordinates of the object of interest be x_I and y_I. The radial correction factors are computed using an OpenCV utility[3].

It is useful to normalize the image coordinates and re-map them to be offsets x_o and y_o from the center of the image:

$$x_o = \frac{x_c}{f_x} - \frac{1}{2} \qquad y_o = \frac{y_c}{f_y} - \frac{1}{2} \tag{10}$$

Using the distance of this re-mapped coordinate from the center of the image r from Equation 11, the radial correction factor can be applied:

$$r = \sqrt{\left(x_o - \frac{1}{2}\right)^2 + \left(y_o - \frac{1}{2}\right)^2} \tag{11}$$

$$c = 1 + c_1 \cdot r^2 + c_2 \cdot r^4 + c_3 \cdot r^6 \tag{12}$$

The new, useful camera coordinates x_u and y_u after this correction is applied are:

$$x_u = x_o \cdot c \qquad y_u = y_o \cdot c \tag{13}$$

With these corrected coordinates, real position may now be computed. The real position is computed by rotating a ray to point from the center of the vehicle to (x_I, y_I), projecting the ray until its z-component is zero, and then adding the position vectors of the camera and the vehicle. For the following transformations, the camera is in the vehicle's reference frame, and the vehicle is in the global reference frame. Assume a ray starting at the origin of the vehicle's reference frame to be pitched toward (w_I, y_I). The ray's pitch, given in 14, is computed using the camera's pitch θ_C and the vehicle's pitch θ_V:

$$\theta = \left(\frac{1}{2} - y_u\right) \cdot f_y + \theta_C + \theta_V \tag{14}$$

As a visual aid, θ_C and θ_V are shown in Figure 6, and ψ_C and ψ_V are shown in Figure 7.

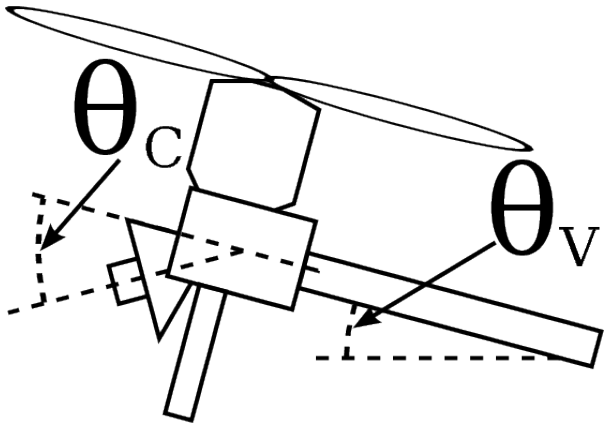

Figure 6: The camera and vehicle pitch angles

Similarly, the ray's yaw is given in Equation 15.

$$\psi = \left(x_u - \frac{1}{2}\right) \cdot f_x + \psi_C + \psi_V \tag{15}$$

In addition, the roll of this ray is $\phi_C + \phi_V$. The ray's roll has no component from the image, as this can't be extracted without known points of reference in the image. Using this ray, a rotation matrix $\mathbf{R}$ is constructed. A unit vector in the x-direction is multiplied by R to produce a unit vector $\vec{a}$ directed

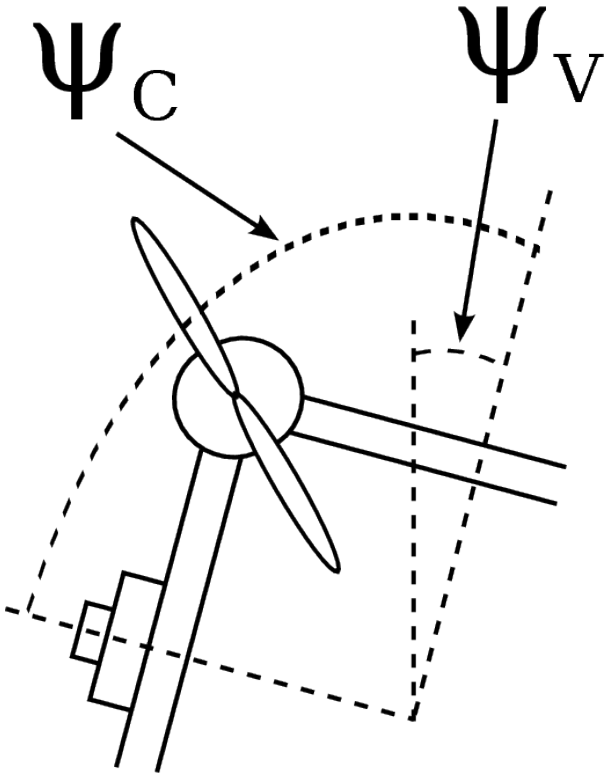

Figure 7: The camera and vehicle yaw angles

from the camera to $(x_I, y_I, 0)$ in the image:

$$\vec{a} =< 1, 0, 0 > \times \mathrm{R} \tag{16}$$

$\vec{a}$ is then scaled until its z is equal to the sum of the camera's height in the vehicle reference frame and the vehicle's height in the global frame $z_C + z_V$, producing a vector $\vec{b}$:

$$\vec{b} = \vec{a} \cdot \frac{z_C + z_V}{\vec{a}_z} \tag{17}$$

$\vec{b}$ is now the camera-relative position vector for the object of interest. Adding the position vectors of the camera and vehicle finally yields the global position vector for the object of interest.

7.2.3 Testing Procedure

Experimental testing is critical to ensure the above math works correctly in all situations. To gather ex-

perimental data, mount a camera at a known roll, pitch, and yaw. Place a small dot somewhere in the camera frame. Record both the read-world location of the small dot relative to the camera and the position of the dot in camera coordinates. Create a test case with all of this information.

7.3 Ground Robot Detection

7.4 Ground Robot Time To Edge

This section describes the algorithm used to calculate the minimum time a Ground Robot will take to reach an out of bounds edge of the arena. The algorithm takes into account the randomized movement of the Ground Robots and simplifies their trajectory using a normal Gaussian distribution.

7.4.1 Ground Robot Movement

Ground Robots follow two rules for movement based on an internal clock. Every five seconds a Ground Robot modifies its orientation by random angle from -20 to 20 degrees. Every 20 seconds a Ground Robot modifies its orientation by 180 degrees, also known as turning around. Starting from time zero, a Ground Robot can modify its orientation three times before turning around. This gives a span from -60 to 60 degrees from its original orientation. The randomized angles lead to a normal Gaussian distribution of possible trajectories. The obvious implication is that Ground Robots most often maintain the orientation they started with.

7.4.2 Trajectory Prediction

- Uses two vectors at -45 degrees and 45 degrees
- Handles cases where shortest distance is along vectors or along current trajectory
- Graphics of arena, ground robot, and prediction vectors
- Confidence interval of falling within prediction vectors

7.5 Localization

7.5.1 Introduction

The localization algorithm is used to correct the vehicle's idea of its position by looking at the arena grid below it. This is achieved by extracting features (points along gridlines) from the image of the grid, then comparing these with the features the vehicle would expect to see based on its current (and outdated) idea of its position.

7.5.2 Extracting Image Features

The first step is to find points along the gridlines in the image taken of the arena grid. To do this we use OpenCV to find contours in the image, then choose points along these contours as the features.

7.5.3 Calculating Expected Features

The next step is to calculate a set of expected features based on the vehicle's current idea of its position and orientation, and camera's resolution and field of view. To make the math work out a little bit better (and by historical accident) a different coordinate system in which the x axis points to the right, the y axis forward, and z up is used for this step. The camera's position in this coordinate system is therefore given by $(x', y', z') = (y, x, -z)$, and this position will be the one referred to from here on when generating expected features. Also, for the purposes of feature generation the camera is assumed to be pointing straight downward with the image plane parallel to the ground, and so the only rotational information used is the yaw angle ψ.

The first step in generating features is to compile a transformation that can be used to convert coordinates on the ground to coordinates in the image. This transformation is comparable to the inverse of the method discussed earlier for frame of reference mapping, but is somewhat simpler because of the assumption about the camera being oriented with the ground. Specifically, this transformation is the composition of the following five transformations:

1. a translation to coordinates relative to the point

directly beneath the camera on the ground (specifically, one by $(-x, -y)$)

2. a rotation to align the coordinates with the camera rather than the ground (one by $-\psi$)

3. a possibly-non-uniform scale to put the coordinates in image units rather than ground units (one by $(r_w/g_w, r_h/g_h)$, where (r_w, r_h) are the horizontal and vertical resolutions of the image and (g_w, g_h) are the side lengths of the rectangle covering the area of the ground that is visible to the camera, which are calculated as $(g_w, g_h) = 2z(\tan(f_w/2), \tan(f_h/2))$, where (f_w, f_y) are the horizontal and vertical fields of view).

4. a translation to move the origin from the center of the image to the top left (one by $(r_w/2, -r_h/2)$)

5. a reflection about the x axis to make the y coordinates oriented the correct way (implemented as a scaling by $(1, -1)$)

The second step in generating features is to enumerate the visible corners and to apply the transformation calculated in the previous step to each of them. In order to enumerate visible corners, the side lengths (g_w, g_h) of the visible rectangle are calculated as before and then those and the yaw angle ψ are "normalized" by setting ψ to its floating-point remainder with π (which produces an equivalent rectangle because of rotational periodicity and rectangles' rotational symmetry) and then by exchanging g_w and g_h and subtracting $\pi/2$ from ψ if $\psi \geq \pi/2$ (which also produces an equivalent rectangle because the reflection cancels the rotation). The result of this is a rectangle with side lengths (g_w, g_h) which is rotated by an angle $\psi \in [0, \pi/2)$. Besides the trivial case where $\psi = 0$ and the rectangle is perfectly aligned with the ground, there are essentially two different cases to handle now, as shown in Figure 8:

In every case, the lines $l_0, \ldots, l_3$ are labeled in counterclockwise order starting with the line that would cross the -x axis (the left side) for $\psi = 0$ and the y coordinates of the corners are labeled y_b, y_r, y_t, y_l standing for "bottom", "right", "top", and "left" respectively in counterclockwise order starting with the point in the third quadrant (bottom-left) for $\psi = 0$. The values for these y coordinates are given by:

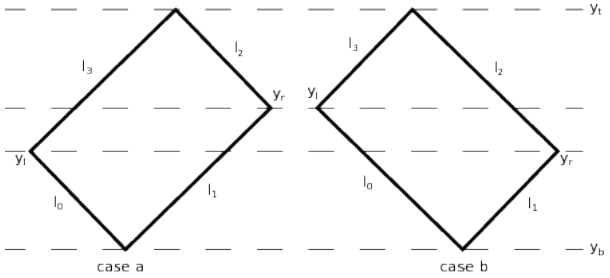

Figure 8: The two non-trivial cases for the "normalized" visible ground rectangle

$$
\begin{aligned}
y_b &= y - (g_w/2)\sin\psi - (g_h/2)\cos\psi \\
y_l &= y - (g_w/2)\sin\psi + (g_h/2)\cos\psi \\
y_t &= y + (g_w/2)\sin\psi + (g_h/2)\cos\psi \\
y_r &= y + (g_w/2)\sin\psi - (g_h/2)\cos\psi
\end{aligned}
$$

Each of the lines is represented simply using slope-intercept form $x = my + b$. Because each pair of opposite lines is parallel and their intercepts with the x axis are symmetric, the line equations can be written as:

$$
\begin{aligned}
x_{l_0}(y) &= m_1 y - b_1 \\
x_{l_1}(y) &= m_2 y + b_2 \\
x_{l_2}(y) &= m_1 y + b_1 \\
x_{l_3}(y) &= m_2 y - b_2
\end{aligned}
$$

where:

$$
\begin{aligned}
m_1 &= -\tan\psi \\
m_2 &= \cot\psi \\
b_1 &= (g_w/2)(\sin\psi\tan\psi + \cos\psi) \\
b_2 &= (g_h/2)(\cos\psi\cot\psi + \sin\psi)
\end{aligned}
$$

With these formulas in place, the corners can be enumerated in each of the three cases by stepping through every grid line parallel to the x axis (which

are all at integral values of y) that is covered by the viewable ground rectangle and then stepping horizontally through every point in the rectangle on each of those lines (which are at integral values of x). In the trivial case, this is accomplished by stepping from $y = \lceil y_b \rceil$ to $y = \lfloor y_t \rfloor$ and then from $x = \lceil x - g_w/2 \rceil$ to $x = \lfloor x + g_w/2 \rfloor$ on each of those lines. For non-trivial case a where $y_l < y_r$, as shown in Figure 8, this is done by first stepping from $y = \lceil y_b \rceil$ to $y = \lfloor y_l \rfloor$ and then from $x = \lceil x_{l_0}(y) \rceil$ to $x = \lfloor x_{l_1}(y) \rfloor$ along each of those lines, then stepping from $y = \lceil y_l \rceil$ to $y = \lfloor y_r \rfloor$ and from $x = \lceil x_{l_3}(y) \rceil$ to $x = \lfloor x_{l_1}(y) \rfloor$, then from $y = \lceil y_r \rceil$ to $y = \lfloor y_t \rfloor$ and from $x = \lceil x_{l_3}(y) \rceil$ to $x = \lfloor x_{l_2}(y) \rfloor$. Similarly, for non-trivial case b where $y_r < y_l$, this is done by stepping from $y = \lceil y_b \rceil$ to $y = \lfloor y_r \rfloor$ and from $x = \lceil x_{l_0}(y) \rceil$ to $x = \lfloor x_{l_1}(y) \rfloor$, then stepping from $y = \lceil y_r \rceil$ to $y = \lfloor y_l \rfloor$ and from $x = \lceil x_{l_0}(y) \rceil$ to $x = \lfloor x_{l_2}(y) \rfloor$, then from $y = \lceil y_l \rceil$ to $y = \lfloor y_t \rfloor$ and from $x = \lceil x_{l_3}(y) \rceil$ to $x = \lfloor x_{l_2}(y) \rfloor$.

7.5.4 Correcting the Position

Now we have a set of observed points and a set of expected points based on our last known position, so what we need to know now is how each point changed in the time since our last position update. To do this we need to find which of the observed points correspond to which of the expected points. We do this by calculating the closest observed point to each of the expected points, and matching these points as a corresponding pair.

We now have a set of features, and pairs of their position at the last position update and their current position. However, these position points are not in the reference frame of the arena, but the reference frame of the image from the camera at the time of the last position update (i.e. the coordinates of the points are in pixel units, not meters). The Frame of Reference mapping algorithm described above is used to get the position of these features in terms of the arena reference frame.

To get the change in position of the vehicle, we calculate an affine transformation matrix from the feature positions. The transformation matrix T will be of the form:

$$T = \begin{bmatrix} cos(\theta) & -sin(\theta) & t_x \\ sin(\theta) & cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the angle of rotation, and t_x, t_y are the translations in the x and y directions.

We now have a translation vector (t_x, t_y) from feature coordinates of our last position to feature coordinates of our current position. The translation vector is relative to the orientation of the vehicle, so it must be rotated by the vehicle's yaw angle to get an absolute, global translation that can be used to update the position. The translation is then subtracted from the position, because the translation is for the features' global coordinates, and the coordinates change in the opposite direction that the vehicle moves (i.e. if the vehicle moves left, it then sees the same coordinates more to the right in the image).

7.5.5 Generating the Transformation Matrix

Because we are able to get accurate orientation information from the IMU, the algorithm for generating the affine transformation matrix becomes a bit simpler, as it only needs to calculate the change in position.

Let n be the total number of features (matched coordinate pairs) that will be used to generate the transformation. We create a $2n \times 3$ matrix A of the form:

$$A = \begin{bmatrix} E_{0,x} \cdot cos(\psi) + E_{0,y} \cdot -sin(\psi) & 1 & 0 \\ E_{0,x} \cdot sin(\psi) + E_{0,y} \cdot cos(\psi) & 0 & 1 \\ E_{1,x} \cdot cos(\psi) + E_{1,y} \cdot -sin(\psi) & 1 & 0 \\ E_{1,x} \cdot sin(\psi) + E_{1,y} \cdot cos(\psi) & 0 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

where ψ is the vehicle's yaw, and $E_{i,[xy]}$ denotes the x or y coordinate of the ith expected feature coordinate.

In other words, this matrix is defined by the fol-

lowing:

$A_{i,1} = E_{i,x} \cdot cos(\psi) + E_{i,y} \cdot -sin(\psi)$, if i is even
$A_{i,1} = E_{i-1,x} \cdot sin(\psi) + E_{i-1,y} \cdot cos(\psi)$, if i is odd
$A_{i,2} = i \bmod 2$
$A_{i,3} = 1 - (i \bmod 2)$

Next we create a vector b of the form:

$$b = \begin{bmatrix} C_{0,x} \\ C_{0,y} \\ C_{1,x} \\ C_{1,y} \\ \vdots \end{bmatrix}$$

where $C_{i,[xy]}$ denotes the x or y coordinate of the ith camera feature coordinate.

By using the A and b matrices to solve the equation $Ax = b$, we get the 3×1 vector x, where x_2 is the translation in the x direction, and x_3 is the translation in the y direction. This is the only information we need from the x vector because we already have a reliable source for the vehicle orientation. We can create the transformation matrix T using the yaw ψ and the translation x_2, x_3:

$$T = \begin{bmatrix} cos(\psi) & -sin(\psi) & x_2 \\ sin(\psi) & cos(\psi) & x_3 \\ 0 & 0 & 1 \end{bmatrix}$$

An important note about this algorithm is that it requires at least 3 matched coordinate pairs to generate any useful results. With 2 or less we can't generate an accurate transformation from one set to the other.

8 Airframe

8.1 Propeller Guards

8.2 Landing Gear

References

[1] *The Debian Administrator's Handbook*. Freexian, 2015. https://debian-handbook.info/.

[2] Axiomtek. *PICO 831 User's Manual*.

[3] OpenCV Documentation. *Camera Calibration with OpenCV*.

[4] PulsedLight, Inc. *LIDAR-Lite Specifications*. https://cdn.sparkfun.com/datasheets/Sensors/Proximity/LIDAR-Lite-Laser-Datasheet.pdf.

[5] systemd Authors. systemd, 2016. https://www.freedesktop.org/wiki/Software/systemd/.