Chandra Library User's Guide

Martin Jay McKee

March 17, 2021

Contents

1 Overview 5
1.1 Description . . . 5
1.2 Dependencies . . . 5
1.3 Platform Indepedence . . . 5
1.4 User's Guide Organization . . . 6

I Library Introduction 7

2 Getting Started 9
2.1 An Embedded Hello World . . . 9

3 Library Architecture 11
3.1 Overview . . . 11
3.2 Core . . . 11
Math . . . 11
Units . . . 11
3.3 HAL . . . 11
Chandra::Chrono . . . 11
General Purpose Input/Output . . . 11
Analog Input/Output . . . 11
Communications . . . 12
3.4 Interface . . . 12
Drivers . . . 12
3.5 Control . . . 12
Kalman Filtering . . . 12
PID . . . 12
Static IMU Calibration . . . 12
Builtin Estimators . . . 14
3.6 Aero . . . 14

II Library Deep Dive 15

4 API 17
4.1 Math . . . 17

Linear Algebra . . . 17
Quaternions . . . 17
Rotations . . . 17
Math Utilities . . . 17
4.2 Units . . . 17
4.3 Chrono . . . 17
System Frequencies . . . 17
Timestamp Clock . . . 17
Conversions between std::chrono and chandra::units . . . 17
4.4 General Purpose Input/Output . . . 18
4.5 Analog Input/Output . . . 18
4.6 Communications . . . 18
I^2C . . . 18
SPI . . . 18
USART . . . 18
4.7 External Analog Input/Output Drivers . . . 18
4.8 Temperature Sensor Drivers . . . 18
4.9 Pressure Sensor Drivers . . . 18
4.10 Inertial Motion Drivers . . . 18
4.11 Recursive Filters . . . 18
4.12 Alpha-Beta-(Gamma)-(Eta) Filtering . . . 18
4.13 Kalman Filtering . . . 18
4.14 PID Control . . . 18
4.15 Static IMU Calibration . . . 18

5 Chandra Cookbook 19
5.1 Clock and Timing . . . 19
Performance Timers . . . 19
5.2 Digital I/O . . . 19
Reading a digital pin . . . 19
Writing to a digital pin . . . 19
Reading a mechanical switch/button . . . 19
Manual control of an LED . . . 19
Automatic PWM of an LED . . . 19
Automatic PWM of an LED with asynchronous update . . . 19
Event on a pin falling (rising) edge . . . 19
Reading a Quadrature Encoder . . . 19
5.3 Analog I/O . . . 19
Raw reading of a single ADC channel . . . 19
Scaled reading of a single ADC channel . . . 19
Reading multiple ADC channels . . . 19
5.4 Math . . . 19
Construct a matrix . . . 19
Solve a system of equations . . . 19
Invert a matrix . . . 19
5.5 Control . . . 19
Kalman . . . 19
PID . . . 19
IMU Calibration . . . 19

IIIAppendicies 21

A Setting up MCUXpresso for LPC Series Microcontrollers 23
A.1 Installation . . . 23
A.2 Optimization and Standards . . . 23
A.3 Include Paths . . . 23

A.4 Preprocessor Definitions . . . 23

B Porting Guide 25
B.1 Low-Level Register Access . . . 25
B.2 Clock and Frequencies . . . 25
B.3 GPIO . . . 25
B.4 ADC . . . 25
B.5 Timers . . . 25
B.6 I^2C . . . 25
B.7 SPI . . . 25
B.8 USART . . . 25
B.9 (Optional) DAC . . . 25
B.10 (Optional) Special Peripherals . . . 25

C Notes on IMU Calibration and Errors 27
C.1 Static Calibration . . . 27
C.2 Dynamic Errors . . . 27
C.3 Modified Calibration Forms . . . 28
Three-Parameter (bias only) or Six-Parameter Calibration . . . 28
Hardware Sensor Bias . . . 29

D References 31

1 Overview

Chandra is a collection of almost entirely header-only libraries that are designed to support the development of real-time embedded control systems on microcontroller systems. While Chandra supports running inside an RTOS, it does not require it nor is it specifically designed for such usage as the libraries are designed such that functionality can easily accessed from an explicit event loop. There is a single "core" .cpp file that defines the whole Chandra runtime[1]. Chandra is best understood as not, strictly, a single library, but as a combination of libraries described in the following. The goal of Chandra is to provide powerful and safe abstractions without using excessive resources either in storage or computation.

1.1 Description

As they stand, the Chandra libraries consist of five sub-libraries: Chandra-Core, Chandra-HAL, Chandra-Interface, Chandra-Control, and Chandra-Aero. The Chandra Core libraries are functionality that is required for any of the other libraries but which is - otherwise - stand alone. The core libraries could be used completely by themselves, though they do not provide enough functionality to implement a complete application because they do not contain features such as input and output. The HAL libraries provide a hardware abstraction layer for the platform with a consistent API. This is the only sub-library that requires explicit porting to new processor platforms[2]. The HAL includes things such as a system clock implementation, timing functionality, GPIO, serial peripherals, etc. Some of the functionality in the HAL is platform independent as it is based directly on lower level functionality. The porting documentation later in this guide describes what types and functions need to be implemented. The third sub-library is Chandra-Interface which includes drivers for external components such as GPS, IMU, Baro, ADC/DAC, pin expansion, etc. The drivers are platform independent and depend upon Chandra-HAL. Forth thre is Chandra-Control which provides observers, filters and control loop implementations. Finally, Chandra-Aero provides special-purpose functions for aerodynamics related programs. This includes functionality such as an atmosphere (altitude, pressure, temperature, density) model, basic air properties, lift and drag calculation, etc.

1.2 Dependencies

The Chandra libraries are written against the C++14 standard and require a compiler which is compatible with the standard, including language features such as: ranged for loops, generalized lambda expressions, constexpr expressions, and variadic templates. Any C++14 compliant compiler should have no issues with the code. Only minimal standard library support is needed but includes: chrono, type_traits, number_traits, and std::common_type.

To use the code generators bundled with Chandra, it is also necessary to have a Python 3 interpreter installed along with several libraries. To run the generator for Chandra Units, the Mako Templating engine is required. For the Kalman Filter Optimizer, Mako and SymPy are required.

The documentation is built in LaTex and requires a number of packages including pygments (which - in return - requires Python) and graphicx. In general, users should not need to rebuild the documentation as Chandra is distributed with the final built PDF of the documentation.

1.3 Platform Indepedence

The bulk of the Chandra libraries are platform independent. The only portion of the libraries which are platform dependent are those within Chandra-HAL – the *H*ardware *A*bstraction *L*ayer. Porting of the HAL functionality

[1] Currently there is only a single variable definition which could also be defined directly in the application code. In future, however, additional values could be defined and, as such, it is cleaner to place them in the .cpp file.

[2] In future, it may make sense to optimized some of the core functionality based on the platform. This might include choosing loop variable sizes or other micro-optimizations. The core functionality should work without requiring any porting, however.

to new platforms is required but the remainder of the libraries can be used with no changes so long as a C++14 compiler and standard library are available. If some standard library features are unavailable on the target platform, they can also be included externally.

1.4 User's Guide Organization

This guide is arranged in five major sections. The first, this overview, is simply a high-level overview which should be sufficient for a new user to understand what the Chandra library is - in a basic way - but ability to use the library should not be expected by this point. The following four sections are intended to provide a deeper introduction to the components in the library, as well as rationale, usage information, and general reference. The second section of the guide is a getting started chapter which includes ?? complete applications (targeted at an NXP LPCXpresso845MAX development board) to demonstrate basic use of the library, features, code structure, and compilation. The third section of this guide covers the overall library architecture. It introduces all the major components of the library and the rationale for including them. Following the architecture documentation, the is an in depth API (*A*pplication *P*rogrammer *I*nterface) reference which describes the complete interface for every type and function. The final section of the guide is a code "cookbook" which includes code fragments to solve specific problems[3]. This final section is intended to simplify the learning of the library as well as demonstrating some of the less obvious features.

Beyond the primary descriptive sections there are appendicies which include generally useful information that does not fit in the aforementioned sections. These subjects include how to set up the default development environment for the NXP processors, a guide on the process of porting the platform specific portions (Chandra-HAL) of the library, as well as general notes on functionality that go beyond what is required for a general understanding of the library. At the end there is a list of the references mentioned throughout the guide.

[3] Unless otherwise specified, the cookbook code is platform independent.

Part I

Library Introduction

2 Getting Started

The Chandra libraries were originally written for personal projects targeting the NXP LPC series of ARM Cortex-M microcontrollers. Despite this, however, the Chandra libraries were written from the very beginning to expose an interface that can be ported to multiple architectures without loss of functionality. The Chandra libraries are designed to be extremely powerful without being too excessively difficult to use. Furthermore, they are designed with safety in mind. While they may not be as simple to use as an Arduino compatible platform, the libraries are designed in such a way as to provide more direct access to the hardware and so that they can be optimized[1] to a much higher level than some of the core Arduino functionality. Having said that, Chandra is not a competitor to Arduino. It is aimed at implementing control systems. Specifically, it is designed for things like rovers and flight computers. Additionally, while it could certainly be used for high-level systems, it is mostly intended for deeply embedded, real-time systems.

This section will introduce some of the central concepts and functions in the Chandra library in the context of example applications.

2.1 An Embedded Hello World

As is typical of embedded (microcontroller) platforms, the first introduction to code will blink a single LED (*L*ight *E*mitting *D*iode). For those used to the basic blink of Arduino, however, the code may look unusual due to the fact that Chandra is intended to be used in an event-driven manner. The whole blink code is as follows.

```
1  // Standard Includes
2  #include <chrono> // std::chrono
3  using namespace std::literals::chrono_literals; // Use chrono UDL suffixes
4
5  // Chandra Includes
6  #include<chrono.h> // Chandra-Core clock functions
7  #include<leds.h> // LED object definitions
8
9  chandra::io::LED led(0, 0, true); // Construct the LED object
10
11 int main(void){
12         chandra::chrono::timestamp_clock::init(); // Initialze the processor clock
13
14         led.breathe(1s); // Configure the LED to breathe over 1 second
15
16         while(true) {
17                 led.update(); // Update the LED as needed
18         }
19
20         return 0;
21 }
```

There are four lines in the code that are implementing the functionality. On line 9 the LED object (*led*) is created – passing in the port and bit number of the GPIO pin. This line is platform dependent as some ports (such as

[1]Compilation of an application built on the Chandra Libraries without compiler optimization *WILL* lead to large slow programs. Chandra is implemented using templates and other C++ features that cause bloated unoptimized code. Nevertheless, when compiler optimization is enabled, much of the functionality is optimized to very nearly the hand-written equivalent. Do not compile Chandra programs without optimization and expect svelte code. You won't get it. With optimization, however, you will.

the Arduino port) may use a different method of identifying the pin to use. On the default platform [2], the green LED is connected on port 0, bit 0. The final parameter (*true*), specifies that the LED is connected in an active-low configuration. Regardless of the configuration, the LED is initialized in the off state. The second functional line is line 12 which calls the built-in processor clock initialization function. This function hides all the platform specific code for generating a continuous timestamp clock. It is not only platform dependent but may also function differently based on preprocessor directives. These directives are outlined in the library deep-dive portion of this manual. In any case, this line should be the first line of any Chandra based program to ensure that the clock is running and initialized. In so doing, timed functions (such as the LED blink here) will work correctly, but other portions of the library depend upon the clock rate having been calculated.

Line 14 tells the LED object that the function it is to perform is to breathe (increase and decrease in brightness) with a period of one second. Chandra uses the C++ standard library std::chrono to define time periods and here the user defined literal from chrono *1s*, to cleanly specify 1 second. At this first call to the *breathe()* function, the LED will update its state when its *update()* method is called. This happens in an infinite loop on line 17, leading to an LED that breathes indefinitely.

While this example demonstrates the creation of the hardware objects and all includes in a single file, it is better form with Chandra to move such hardware into external *hw.h*/*hw.cpp* files. In this case, the modified source would look something like the following. First *hw.h* would contain,

```
#ifndef HW_H
#define HW_H
#include<leds.h> // LED object definitions

extern chandra::io::LED led; // Define the LED object
#endif
```

the *hw.cpp* file,

```
#include "hw.h"
chandra::io::LED led(0, 0, true); // Construct the LED object
```

and the *main.cpp* file,

```
// Standard Includes
#include <chrono> // std::chrono
using namespace std::literals::chrono_literals; // Use chrono UDL suffixes

// Chandra Includes
#include<chrono.h> // Chandra-Core clock functions

#include "hw.h"

int main(void){
        chandra::chrono::timestamp_clock::init(); // Initialze the processor clock

        led.breathe(1s); // Configure the LED to breathe over 1 second

        while(true) {
                led.update(); // Update the LED as needed
        }

        return 0;
}
```

Due to its simplicity, there is little savings in the application file in this case. In more complicated programs, however, the hardware files can absorb quite a bit of variable creation boilerplate and this general structure leads to clearer application code.

[2] NXP LPCxpresso845Max development board

3 Library Architecture

3.1 Overview

The following sections provide an outline of the library facilities. Each feature will be explained and a very basic example of use provided. Complete description of the API is saved for the API chapter which is arranged in a manner that mirrors this chapter. Essentially, this chapter is an introduction to functionality and the rationale behind it while the API reference gets into the nuts and bolts of each function, class, and datatype provided by the library.

3.2 Core

Math

Matrix Types

The ADD SYMMETRIC AND SKEW-SYMMETRIC MATRICIES. THESE CAN HAVE OPTIMIED STORAGE AND MAY BE POSSIBLE TO OPTIMIZE OPERATIONS FOR THESE SPECIFIC FORMS (I.E. ADDITION CAN BE OPTIMIZED, MULTIPLICATION THAT CREATES A SKEW-SYMMETRIC MATRIC MAY BE OPTIMIZABLE, ETC.

Matrix Operations

IT MAY MAKE SENSE TO USE EXPRESSION TEMPLATES TO SELECT THE IMPLEMENTATION OF OPERATIONS.

Units

3.3 HAL

Chandra::Chrono

System Clock

To configure the timestamp clock, a preprocessor directive describing the onboard peripheral that will be used as the source is used.

Preprocessor Symbol	Definition
SCT_HARDWARE_TIMESTAMP_MODE	Use an SCT clock as the timestamp generator
SYSTICK_SOFTWARE_TIMESTAMP_MODE	Use the SysTick clock as the timestamp generator

System Frequencies

Performance Timers

General Purpose Input/Output

Analog Input/Output

CURRENTLY THERE IS NO API SPECIFIED FOR DAC, THERE SHOULD BE.

Communications

I^2C

SPI

USART

3.4 Interface

Drivers

Analog Input/Output

THESE ARE DRIVERS FOR EXTERNAL ADCS AND DACS. THE API IS THE SAME AS THE INTERNAL ADC/DAC INTERFACE BUT THESE DRIVERS ARE PLATFORM INDEPENDENT BECAUSE THEY DEPEND UPON THE COMMUNICATION SUPPORT IN THE HAL.

GPS

NEED TO COME UP WITH AN API AND A COMPUTATIONAL ENGINE TO BASE ALL OF THE DRIVERS ON.

Inertia Measurement Sensors

Servo Drivers

RF Transceivers

CURRENTLY THIS ONLY HAS THE RFM9X

3.5 Control

Kalman Filtering

The Kalman filtering components of the Chandra library are designed to support multiple approaches to constructing a kalman filter. Explicit implementation of a Kalman filter directly in source code by configuring the various matricies is simple. This also has the advantage that it allows for dynamic modification of the matricies during runtime. A second method for constructing a kalman filter for Chandra projects is to use the included code generator. This generator takes as input a description file which includes descriptions of variables and the state as well as the state transition and measurement equations; and it constructs the required code C++ for the Kalman filter. This constructed filter provides the exact same API as the built-in explicit kalman filter. Additionally, it will do simplification of the resultant equations to minimize the computational load of the final filter. Currently, the code generator only works

Code Generator

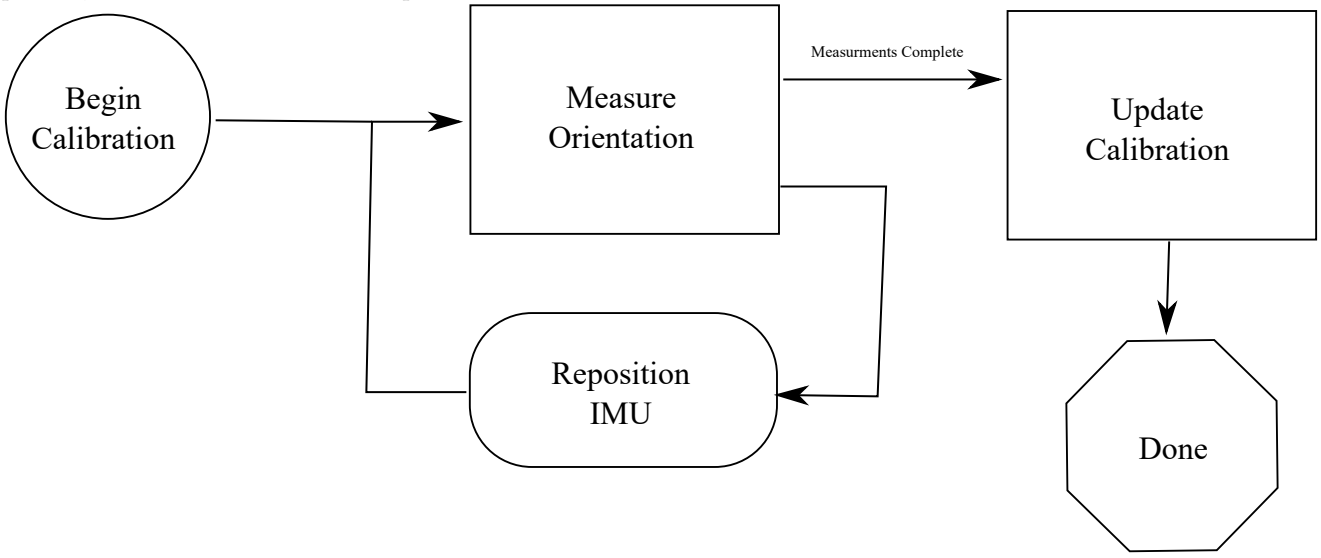
The code generator is written in the Python programming language and uses the SymPy library to handle symbolic representations of the Kalman equations. This is how it is able to do operation reduction (subexpression folding) as well as pre-calcuation of constants and multiplication by unity and zero. The code generator currently only works with the basic Kalman Filter formulation but is intended to be expanded to also work with – at least – the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF). In so doing, the code generator should speed the implementation and testing of various algorithms in Chandra applications.

PID

Static IMU Calibration

The basic IMU calibration API is based on the structure found in NXP application note AN4399. This white paper – which discusses high-precision, full 12 parameter calibration of accelerometers – looks at various optimal measurement orientations for calibration of accelerometer offsets and gains in addition to formulating a least means

squared approach to calculating the calibration parameters. The API[1] provides calibration objects for accelerometer only, gyroscope only, magnetometer only, accelerometer/gyroscope, accelerometer/magnetometer, and 9-axis sensors. The calibration completed is limited to a static calibration and, as such, it does not provide the information required to calibrate gyroscope gains. It does, however, provide sufficient information for an initial gyro bias calibration. All of these objects follow the same general API where a calibration is started and for each step the required orientation of the sensor is returned. This allows for a manual repositioning of the IMU prior to the next measurement. When the required number of steps have been completed, the calibration values are calculated, the IMU object – optionally – updated, and the calibration is complete.



The default measurements provided by the IMU calibration as defined in the aforementioned application are – generally – insufficient to do a complete calibration of a magnetometer and thus the built-in static calibration objects do not include a full calibration of the magnetometer. Basically, the most that can be done with the current API is a 12-parameter calibration of the accelerometers, a 3-parameter (bias-only) calibration of the gyroscopes, and a low-resolution 6-parameter calibration of the magnetometers (axis scale and bias). Dynamic calibration is required to calibrate the gain parameters of the gyroscopes while a greater number of samples are required to calculate an accurate ellipsoidal fit for the magnetometer calibration.

Given an IMU object named *imu*, the following code demonstrates the calibration procedure.

```
// Create the calibrator object with default parameters (orthagonal cubic calibration)
auto imu_calibrator = StaticIMUCalibrator<>(imu);
auto cal_start_result = imu_calibrator.init();
// ... reposition to cal_start_result.next_orientation
bool done = false;
while(!done) {
        // Take a measurement step at the default orientation
        auto cal_step_result = imu_calibrator.measure();
        done = cal_step_result.complete;
        // ... reposition to cal_step_result.next_orientation
}

// Calculate the calibration and apply to the IMU
auto cal_final_result = imu_calibrator.calibrate();
```

While the simplest possible version, this code does – at least – introduce the general usage of the static calibrator objects. There are three functional calls: *init()*, *measure()*, and *calibrate()*. Each of these return an object which contains data useful for the tracking and control of the calibration process.

By default the call to *calibrate()* will update the calibration in the passed in *imu* object. The calibration parameters will be calculated regardless and returned as members of the return value. Passing *false* to this function will disable the update of the calibration but still return the parameters.

[1] *A*pplication *P*rogrammer *I*nterface

The StaticIMUCalibrator object has member functions which access the parameters of the calibration such as number of measurement orientations, the target orientations, the number of samples to take for each measurement, the type of calibration (3-/6-/12- parameter) of each sensor, etc. Information regarding the complete API is contained in the Library Deep Dive section while there is an appendix containing general notes on IMU calibration in general.

Builtin Estimators

Madgwick's AHRS

Alpha-Beta-(Gamma)-(Eta)

3.6 Aero

ATMOSPHERIC PROPERTIES, MODEL OF ATMOSPHERE (PRESSURE, DENSITY, TEMPERATURE, ALTITUDE, ETC.), BASIC AERODYNAMICS

3.7 Aero

Atmosphere Calculations

Gravity Calculations

http://walter.bislins.ch/bloge/index.asp?page=Earth+Gravity+Calculator

Part II

Library Deep Dive

4 API

4.1 Math

Linear Algebra

Quaternions

Rotations

Math Utilities

4.2 Units

4.3 Chrono

System Frequencies

Timestamp Clock

Conversions between std::chrono and chandra::units

NONE. THIS DOESN'T HAPPEN, AT THE MOMENT. IT REALLY SHOULD THOUGH.

4.4 General Purpose Input/Output

4.5 Analog Input/Output

4.6 Communications

I^2C

SPI

USART

4.7 External Analog Input/Output Drivers

4.8 Temperature Sensor Drivers

4.9 Pressure Sensor Drivers

4.10 Inertial Motion Drivers

4.11 Recursive Filters

4.12 Alpha-Beta-(Gamma)-(Eta) Filtering

4.13 Kalman Filtering

4.14 PID Control

4.15 Static IMU Calibration

5 Chandra Cookbook

5.1 Clock and Timing

Performance Timers

Timing a function

Conditional timing of a function

Loop frequency measurement

Loop frequency statistics

5.2 Digital I/O

Reading a digital pin

Writing to a digital pin

Reading a mechanical switch/button

Manual control of an LED

Automatic PWM of an LED

Automatic PWM of an LED with asynchronous update

Event on a pin falling (rising) edge

Reading a Quadrature Encoder

5.3 Analog I/O

Raw reading of a single ADC channel

Scaled reading of a single ADC channel

Reading multiple ADC channels

5.4 Math

Construct a matrix

Solve a system of equations

Invert a matrix

5.5 Control

Kalman

PID

IMU Calibration

Part III

Appendicies

A Setting up MCUXpresso for LPC Series Microcontrollers

A.1 Installation

A.2 Optimization and Standards

A.3 Include Paths

A.4 Preprocessor Definitions

B Porting Guide

The only section of the Chandra libraries that require porting when moving to new platforms is the Chandra-HAL components. Any other components (core, control, and aero) are platform independent.

ALL COMPONENTS THAT REQUIRE PORTING NEED TO HAVE HEADER TEMPLATES.

B.1 Low-Level Register Access

THIS IS THINGS LIKE CLOCK POWER, PERIPHERAL RESET, GPIO MAPPING, ETC.

B.2 Clock and Frequencies

B.3 GPIO

B.4 ADC

B.5 Timers

THERE IS NO TIMER API DEFINED AT THE MOMENT, BUT THERE REALLY SHOULD BE ONE...

B.6 I^2C

B.7 SPI

B.8 USART

B.9 (Optional) DAC

B.10 (Optional) Special Peripherals

C Notes on IMU Calibration and Errors

IMU sensors – including accelerometers, gyroscopes, and magnetometers – present a wide range of potential error sources. The drivers contained in Chandra include the equations for a full twelve-parameter calibration which will correct for the most basic static errors. Even so, there are a number of dynamic effects that the built-in calibration is unable to correct. Both of these effects are outlined here.

C.1 Static Calibration

The static calibration of an IMU sensor is defined by the following general equation.

$$\vec{y_c} = \mathbf{W}\vec{y_m} + \vec{b} \tag{C.1}$$

Here the quantity measured from the sensor is $\vec{y_m}$, $\vec{b}$ is a bias value, and $\mathbf{W}$ is a gain matrix. The calibrated value $\vec{y_c}$ is the result. This general form hides several possible sources of error which may be calibrated out. The bias is the least complicated.

Bias is a – generally – constant offset from the value the sensor should be measuring. As long as the bias is small, it may not cause massive issues for accelerometers or magnetometers when used for orientation estimation. If small, the bias only causes a small error in the calculated orientation. When accelerometers are used for inertial navigation or gyroscopes for orientation tracking, however, the bias term becomes especially important as the sensor readings are integrated. This integration causes the bias term to effect the estimation without bound. As such, for AHRS and MARG applications it is vitally important that the gyroscope bias be corrected. For INS applications, the accelerometer bias, also, needs to be corrected.

The gain matrix hides much more complexity than the bias; but it is no more difficult to understand. Simply, the gain matrix can be represented as a combination of matricies which apply different transformations to the values. These transformations may be bulk, such as physical rotation of the sensor represented by a matrix, $\mathbf{R}$, or corrective such as the axis scaling factor in a diagonal matrix $\mathbf{S}$. Additionally, there are errors that result from the sensor axes being non-orthogonal or general misalignment (either in the package or on the board) which are represented by a matrix $\mathbf{O}$. The final gain matrix then has the form,

$$\mathbf{W} = \mathbf{R}(\mathbf{S} + \mathbf{O}) \tag{C.2}$$

Given that $\mathbf{R}$ is a known quantity for a particular PCB design, it is not included in the calibration but, rather, applied separately. As such, when Chandra calculates gain calibration, it is calculating $(\mathbf{S} + \mathbf{O})$. To do so, however, a correct value of $\mathbf{R}$ must be provided. This is because the full $\mathbf{W}$ matrix is calculated and the calibration correction is then backed out by calculating,

$$(\mathbf{S} + \mathbf{O}) = \mathbf{R^{-1}W} \tag{C.3}$$

C.2 Dynamic Errors

Unfortunately, neither bias nor gain error are truly static errors. They may change with time and often change with temperature. As such, any calibration that is completed is nothing more than an estimate. Dynamic errors include things like temperature effects and noise.

Some temperature effects cause a proportional change in the gain and/or bias of a sensors, in these cases the error can be calibrated out using gain matrix and bias vector corrected by a polynomial fit with temperature as the controlling parameter. For instance, a two-temperature calibration may be corrected by using the following,

$$\vec{b}(T) = \frac{(\vec{b_2} - \vec{b_1})}{(T_2 - T_1)}(T - T_1) + \vec{b_1} \tag{C.4}$$

and,

$$\mathbf{W}(T) = \frac{(\mathbf{W_2} - \mathbf{W_1})}{(T_2 - T_1)}(T - T_1) + \mathbf{W_1} \tag{C.5}$$

Similar solutions may be found for a fit of any order N so long as at least $N+1$ calibrations are calculated at different temperatures. This temperature compensation, however, is not implemented as a core feature of the Chandra libraries due to it being highly application dependent.

Beyond the temperature effects that are possible to calibrate out, however, there are other error sources which cannot be handled so simply. For instance, they may have the effect of superimposing a random walk error on the output. This is often the case for rate gyro sensors. When this happens, there is no consistent proportionality between the state of the sensor and the error of the measurement and, as such, it cannot be removed formulaically. The only recourse is to use an adaptive filter such as a Kalman filter to both identify the error component and remove it. As before, this is not directly implemented in the Chandra libraries being even more application dependent than temperature calibration.

Finally, sensors produce random noise. This may be the result of electrical noise, and affect the magnitude of the measurement signal, or it may be a result of clock inaccuracy or clock jitter and affect the timing of a measurement. Generally speaking, the only control that might be held over the electrical noise is to control the temperature of the sensor or the cleanliness of the power rails. When running a calibration, Chandra will take multiple measurements at each orientation and generate an estimate of the standard deviation of the measurements which are an estimate of the random noise power. This may be used to construct the noise covariance matrix for a Kalman filter or for other similar uses. As with the gain and bias, the sampling clock rate may have a temperature dependence, but it is often not controllable in any meaningful way by the user. On the other hand, some more advanced IMU sensors have the ability to return a timestamp value with a measurement or even run from an external clock. To take advantage of this additional information, the Chandra drivers provide access to a measurement timestamp and sample period. For sensors that do not provide a hardware timestamp, these are generated by the main processor. If available, however, they are sourced from the sensor and corrected using the main clock as a reference. This allows for the removal of clock inaccuracy so long as the main processor clock is more accurate.

C.3 Modified Calibration Forms

There are a couple of possible modifications to the calculation of the calibration which may be used internally to an IMU driver. These are, primarily intended to reduce the computational complexity of generating the corrected value $\vec{y_c}$. They may be used individually or in combination though not all combinations are valid or implemented in any particular driver.'

Three-Parameter (bias only) or Six-Parameter Calibration

If the IMU axes are known to be physically aligned with the body frame to be sensed, the rotation matrix $\mathbf{R}$ will be the identity matrix and some reduced complexity calibration options become reasonable to consider. Firstly, the assumption that the sensor is perfectly aligned and orthogonal may be made and that reduces the calibration to six parameters. Secondly, it may be assumed that three is no gain error whatsoever which will reduce the calibration to three parameters. It should be understood, however, that due to the caching of calculating the matrix $\mathbf{W}$, these methods of calibration only reduce complexity in the case where it can be ensured that $\mathbf{R} = \mathbf{I}$. If configured for three or six parameter calibration, the IMU driver loses the ability to set a rotation matrix and will cause a compile-time error if it is attempted.

By assuming perfect sensor alignment and orthogonality, the $\mathbf{O}$ matrix becomes zero so that the calibration equations become,

$$\vec{y_c} = \mathbf{S}\vec{y_m} + \vec{b} \tag{C.6}$$

Because $\mathbf{S}$ is a diagonal matrix, this reduces the number of multiplications required substantially and speeds up the calculation of $\vec{y_c}$.

Similarly, when doing a static calibration on a rate gyro sensor, there is no information available on the sensor magnitude accuracy. As such, the only reasonable value for the scale matrix, $\mathbf{S}$ is the identity matrix and for $\mathbf{O}$ is the zero matrix. That results in the calibration equations reducing to,

$$\vec{y_c} = \vec{y_m} + \vec{b} \tag{C.7}$$

and requiring only additions.

Hardware Sensor Bias

The general form of the gain/bias calibration as described above is used for most sensors. There are, however, some sensors which include internal offset calibration registers and that, therefore, remove the necessity to calculate the gain correction in the processor code. In this case, the calibration may be calculated as,

$$\vec{y_c} = \mathbf{W}\vec{y_b} \tag{C.8}$$

where $\vec{y_b}$ is the already bias corrected measurement coming from the sensor. While $\vec{y_b}$ need not be calculated directly, this form actually represents,

$$\vec{y_c} = \mathbf{W}(\vec{y_m} + \vec{b_m}) \tag{C.9}$$

with a measurement bias term of $\vec{b_m}$. Some simple algebra allows us to solve for this term,

$$\mathbf{W}(\vec{y_m} + \vec{b_m}) = \mathbf{W}(\vec{y_m}) + \vec{b} \tag{C.10}$$

distributing (using $\mathbf{I} = \mathbf{W}\mathbf{W^{-1}}$),

$$\mathbf{W}\vec{y_m} + \mathbf{W}\vec{b_m} = \mathbf{W}(\vec{y_m}) + \mathbf{W}\mathbf{W^{-1}}\vec{b} \tag{C.11}$$

and simplifying,

$$\vec{b_m} = \mathbf{W^{-1}}\vec{b} \tag{C.12}$$

Thus, $\vec{b_m}$ can be calculated from the same gain and bias as calculated for the standard form. In the Chandra driver for an IMU this modified form is used for sensors that have offset registers available onboard. The API remains consistent regardless.

D References

NEED TO GET THIS SET UP TO HANDLE REFERENCES AND START ADDING THEM.