

Chandra Library User's Guide

Martin Jay McKee

May 20, 2021

Contents

1	Overview	5
1.1	Description	5
1.2	Dependencies	5
1.3	Platform Independence	6
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	13
3.1	Overview	13
3.2	Core	13
	Math	13
	Units	13
3.3	HAL	13
	Chandra::Chrono	13
	General Purpose Input/Output	14
	Analog Input/Output	14
	Communications	14
3.4	Interface	14
	Drivers	14
3.5	Control	14
	Kalman Filtering	14
	PID	15
	Static IMU Calibration	15
	Builtin Estimators	16
3.6	Aero	16
	Atmosphere Calculations	16
	Gravity Calculations	16

II Library API	19
4 Math	21
4.1 Linear Algebra	21
Matrices	21
Vectors	22
4.2 Quaternions	23
4.3 Rotations	23
4.4 Math Utilities	23
5 Units	25
6 Chrono	27
6.1 System Frequencies	27
6.2 Timestamp Clock	27
6.3 Conversions between <code>std::chrono</code> and <code>chandra::units</code>	27
7 General Purpose Input/Output	29
8 Analog Input/Output	31
9 Communications	33
9.1 I^2C	33
9.2 SPI	33
9.3 USART	33
10 Drivers	35
10.1 External Analog Input/Output Drivers	35
10.2 Temperature Sensor Drivers	35
10.3 Pressure Sensor Drivers	35
10.4 Inertial Motion Sensor Drivers	35
11 Control	37
11.1 Recursive Filters	37
11.2 Alpha-Beta-(Gamma)-(Eta) Filtering	37
11.3 Kalman Filtering	37
11.4 PID Control	37
11.5 Static IMU Calibration	37
12 Aero	39
12.1 Atmospheric Calculations – <code>class Atmosphere<></code>	39
Class Template Arguments	39
Type Aliases	39
Constructors	39
Accessors	39
12.2 Gravity Calculations – <code>class Gravity<></code>	39
Class Template Arguments	39
Type Aliases	40
Constructors	40
Accessors	40
III Chandra Cookbook	41
13 Clock and Timing	43
13.1 Performance Timers	43
Timing a function	43
Conditional timing of a function	43
Loop frequency measurement	43

Loop frequency statistics	43
14 Digital I/O	45
14.1 Reading a digital pin	45
14.2 Writing to a digital pin	45
14.3 Reading a mechanical switch/button	45
14.4 Manual control of an LED	45
14.5 Automatic PWM of an LED	45
14.6 Automatic PWM of an LED with asynchronous update	45
14.7 Event on a pin falling (rising) edge	45
14.8 Reading a Quadrature Encoder	45
15 Analog I/O	47
15.1 Raw reading of a single ADC channel	47
15.2 Scaled reading of a single ADC channel	47
15.3 Reading multiple ADC channels	47
16 Math	49
16.1 Conuct a matrix	49
16.2 Solve a system of equations	49
16.3 Invert a matrix	49
17 Control	51
17.1 Kalman	51
17.2 PID	51
17.3 IMU Calibration	51
18 Aero	53
18.1 Atmosphere	53
Pressure and density at a known altitude	53
Altitude from pressure	53
18.2 Gravity	53
Acceleration due to gravity on the Earth's surface	53
Gravitational acceleration above Earth's surface	53
IV Appendicies	55
A Setting up MCUXpresso for LPC Series Microcontrollers	57
A.1 Installation	57
A.2 Optimization and Standards	57
A.3 Include Paths	57
A.4 Preprocessor Definitions	57
B Porting Guide	59
B.1 Low-Level Register Access	59
B.2 Clock and Frequencies	59
B.3 GPIO	59
B.4 ADC	59
B.5 Timers	59
B.6 I^2C	59
B.7 SPI	59
B.8 USART	59
B.9 (Optional) DAC	59
B.10 (Optional) Special Peripherals	59
C Notes on IMU Calibration and Errors	61
C.1 Static Calibration	61

C.2	Dynamic Errors	61
C.3	Modified Calibration Forms	62
	Three-Parameter (bias only) or Six-Parameter Calibration	62
	Hardware Sensor Bias	63
C.4	Calculation of Calibration Vectors	63
	Orthogonal Cubic Vectors	63
	Orthogonal Cubic/Octagonal Vectors	64
	Tetrahedral Vectors	64
D	References	65

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 be accessed from an explicit event loop. There is a single “core” .cpp file that defines the whole Chandra runtime¹. 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 platforms². 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. Fourth there 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.

¹Currently 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.

²In future, it may make sense to optimize 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.

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 Independence

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 *Hardware Abstraction Layer*. Porting of the HAL functionality 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, there is an in depth API (*Application Programmer Interface*) 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 problems³. 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 appendices 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.

³Unless 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 optimized¹ 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 (*Light Emitting Diode*). 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(); // Initialize 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 }
```

¹Compilation 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.

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 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,

```
1  #ifndef HW_H
2  #define HW_H
3  #include <leds.h> // LED object definitions
4
5  extern chandra::io::LED led; // Define the LED object
6  #endif
```

the *hw.cpp* file,

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

and the *main.cpp* file,

```
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
8  #include "hw.h"
9
10 int main(void){
11     chandra::chrono::timestamp_clock::init(); // Initialize the processor clock
12
13     led.breathe(1s); // Configure the LED to breathe over 1 second
14
15     while(true) {
16         led.update(); // Update the LED as needed
17     }
18
19     return 0;
20 }
```

²NXP LPCxpresso845Max development board

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.

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

Matrix Operations

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

Matrix Views

NEED TO MAKE SURE THAT THE MATRIX VIEWS ARE WORKING AS I WANT THEM TO AND THAT THEY ARE USED THROUGH THE DRIVERS WHERE APPROPRIATE.

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

I²C

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 matrices is simple. This also has the advantage that it allows for dynamic modification of the matrices 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

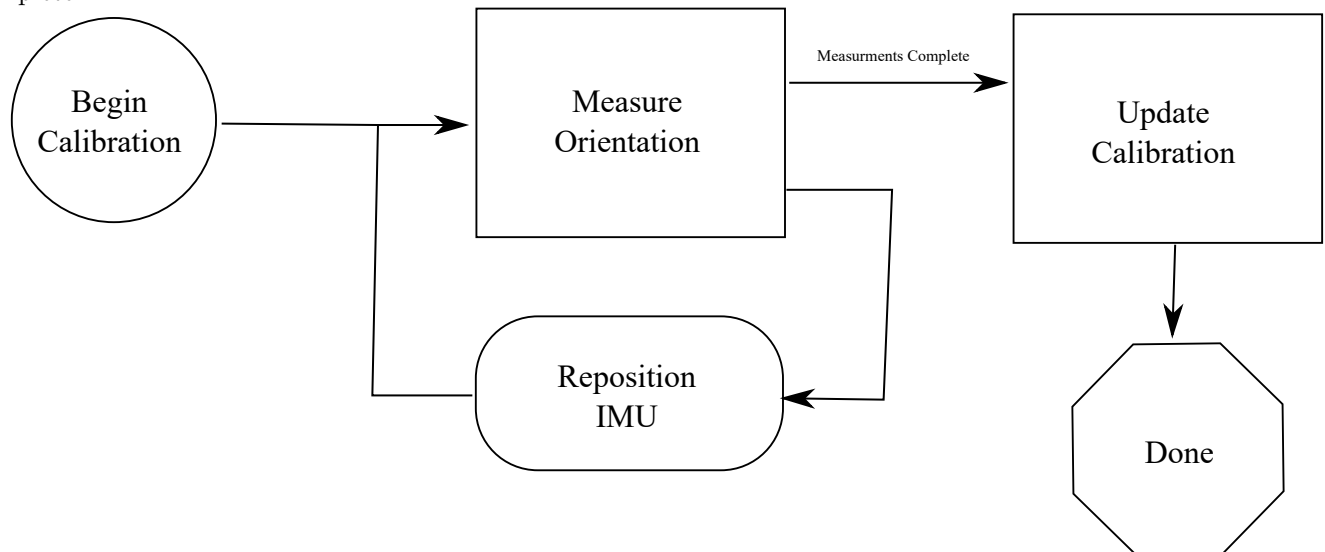
The code generator is written in the Python programming language and uses the SymPy library to handle symbolic representations of the Kalman equations. This is how it is able to do operation reduction (subexpression folding) as well as pre-calculation of constants and multiplication by unity and zero. 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 API¹ 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.

```

1 // Create the calibrator object with default parameters (orthogonal cubic calibration)
2 auto imu_calibrator = StaticIMUCalibrator<>(imu);
3 auto cal_start_result = imu_calibrator.init();
4 // ... reposition to cal_start_result.next_orientation
5 bool done = false;
6 while(!done) {
7     // Take a measurement step at the default orientation
8     auto cal_step_result = imu_calibrator.measure();

```

¹Application Programmer Interface

```
9         done = cal_step_result.complete;
10         // ... reposition to cal_step_result.next_orientation
11     }
12
13     // Calculate the calibration and apply to the IMU
14     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.

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

Atmosphere Calculations

Gravity Calculations

The gravity library provides calculation of surface and airborne gravity calculations to moderate accuracy. Both magnitude of effective gravity and vector results are available, depending upon configuration. The library provides a single class `Gravity<>` which is a highly configurable object that takes a number of compile-time arguments to determine the gravity model being used, the corrections applied, and if a vector value is desired. To account for simple usages, three aliases are also defined: `StandardGravity<>`, `StandardGravity<>`, and `StandardGravity<>`. These aliases of the gravity class provide both the lower precision, low-cost options and the highest-accuracy options. As such, they should be sufficient for most usages. Of course, Section 12.2 of this manual (in the API reference) provides information about all the features of the gravity object, .

Three models of gravity are available: a constant standard gravity, the 1980 IGF (International Gravity Formula), and WGS84 (World Gravity Survey). The constant gravity model is intended for low-accuracy, performance and memory critical applications and just returns the international standard gravity value of 9.80665ms^{-2} . The IGF and WGS84 approximations are much more accurate but have higher system requirements.

The corrections available are two free-air correction approximations, a slab correction, and a centripetal correction. Each of these corrections can be applied or ignored. Generally they are fairly small corrections and, as such, ignoring them is reasonable in the case of a constrained system. The low-precision approximation of the free-air correction, however, only costs a single multiplication and so should generally be included in all cases that airborne gravity estimates are needed.

The gravity calculations are cross-checked and also compared to the values generated by the calculator at: <http://walter.bislins.ch/bloge/index.asp?page=Earth+Gravity+Calculator>. It should be noted that the gravity vectors returned on this webpage are in ECEF (Earth Centered-Earth Fixed) coordinates. The examples in this documentation, on the other hand, use a local tangent

NEU (*N*orth-*E*ast-*U*p) coordinate system. The gravity object is also able to return vectors in ECEF coordinates if desired.

Part II

Library API

4 Math

4.1 Linear Algebra

Matricies

`#include <matrix_core.h>`

Namespace: `chandra::math`

The matricies implemented by Chandra are fixed-size and optimized for small controls and geometry systems. The main class template definition is

```
template<class Value, class Rows, class Columns>
class Matrix;
```

where the value type defines a single value in the matrix and the `Rows` and `Columns` parameters define the shape of the matrix.

Types and Constant Members

THIS HAS A DESCRIPTION OF THE AVAILABLE TYPE DEFINES AND CONSTANT MEMBER VARIABLES. FOR INSTANCE, IS VECTOR, IS SQUARE, ETC.

Construction

Given a 3x3 matrix type defined as follows,

```
using matrix_t = chandra::math::Matrix<double, 3, 3>;
```

we explore the available constructors. To begin with, the default constructor,

```
matrix_t m;
```

The zero-argument constructor is defaulted as is the copy-constructor.

simply reserves the matrix memory for `m`. It does not initialize that memory. As a result, a default constructed matrix should be treated as containing random values. If known values are required, then another construction method with explicit initialization should be used instead.

One such constructor is the initializer-list constructor which takes an arbitrary list of elements to use in constructing the matrix. The signature of this constructor is,

```
Matrix(std::initializer_list<value_t> _values, const bool& _row_major = true);
```

The `_values`, initializer-list contains an arbitrary number of values while the second parameter selects if the matrix is to be initialized in row-major (the default) or column-major order. This constructor is safe with an initialization list of any length. If the list is shorter than the number of elements, some elements will simply not be initialized while a list that is longer than the number of elements will fully initialize the matrix and exit. A few examples are in order. First, initialization of our 3x3 matrix with a short list using the initializer-list constructor.

```
matrix_t m1{ {1, 2, 3, 4, 5} }; // Initialize in row-major order
matrix_t m2{ {1, 2, 3, 4, 5}, false }; // Initialize in column-major order
```

Here both row- and column-major examples are shown. The results are that,

$$m_1 = \begin{bmatrix} 123 \\ 45? \\ ??? \end{bmatrix} \quad (4.1)$$

$$m_2 = \begin{bmatrix} 14? \\ 25? \\ 3?? \end{bmatrix} \quad (4.2)$$

As can be seen, the default row-major order fills the rows – left to right – first and runs out at the end of the list. The remainder of the values in the matrix are uninitialized and are – thus – unknown in the general case. Column-major construction works similarly but fills down the columns first, running out in a similar fashion. With an initializer list at least as long as the final matrix, the matrix will be fully initialized. For example,

```
matrix_t m1{ {1, 2, 3, 4, 5, 6, 7, 8, 9} };
matrix_t m2{ {1, 2, 3, 4, 5, 6, 7, 8, 9}, false };
```

with,

$$m_1 = \begin{bmatrix} 123 \\ 456 \\ 789 \end{bmatrix} \quad (4.3)$$

$$m_2 = \begin{bmatrix} 147 \\ 258 \\ 369 \end{bmatrix} \quad (4.4)$$

DEFAULT COPY-CONSTRUCTOR

The final basic constructor is a converting constructor to allow conversion of size-matched matrices with different value types. Here, the `Rows` and `Columns` parameters come from the type that is being constructed while `V` is the type of the matrix to construct from.

```
template<class V>
constexpr Matrix<const Matrix<V, Rows, Columns>& _other>;
```

Internally, the converting constructor uses a `static_assert` to check that the source datatype is convertible to the destination datatype.

In addition to the standard constructors, there are a number of static methods that construct specific types of matrices. The simplest are single-value filled matrices. The three implemented construction methods in this class have the signatures,

```
static constexpr matrix_t Filled(const value_t& _value);
static constexpr matrix_t Zeros();
static constexpr matrix_t Ones();
```

The first of these will fill the matrix with an arbitrary value. For instance, with our previously defined 3x3 matrix type `matrix_t`,

```
auto m = matrix_t::Filled(42);
```

The matrix ends up as,

$$m = \begin{bmatrix} 424242 \\ 424242 \\ 424242 \end{bmatrix} \quad (4.5)$$

As the names imply, `Zeros` and `Ones` result in matrices filled with the value 0 and 1 respectively.

Vectors

```
#include <matrix_vectors.h>
Namespace: chandra::math
```

Construction

The vector constructors are a subset of the matrix constructors as a result of vectors being a subclass of matrices. As a result, the static construction methods are inherited and are, therefore, still available for vectors.

4.2 Quaternions**4.3 Rotations****4.4 Math Utilities**

5 Units

6 Chrono

6.1 System Frequencies

6.2 Timestamp Clock

6.3 Conversions between `std::chrono` and `chandra::units`

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

7 General Purpose Input/Output

8 Analog Input/Output

9 Communications

9.1 I^2C

9.2 SPI

9.3 USART

10 Drivers

10.1 External Analog Input/Output Drivers

10.2 Temperature Sensor Drivers

10.3 Pressure Sensor Drivers

10.4 Inertial Motion Sensor Drivers

11 Control

11.1 Recursive Filters

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

11.3 Kalman Filtering

11.4 PID Control

11.5 Static IMU Calibration

12 Aero

12.1 Atmospheric Calculations – `class Atmosphere<>`

```
#include <atmosphere.h>
Namespace: chandra::aero
```

Class Template Arguments

Type Aliases

Constructors

Accessors

12.2 Gravity Calculations – `class Gravity<>`

```
#include <gravity.h>
Namespace: chandra::aero
```

Class Template Arguments

```
template<
    class Value,
    GeoidCoordinates VectorCoordinates = None,
    uint32_t Flags = ApproxFreeAirCorrect,
    class AngleUnits = rad,
    class LengthUnits = m,
    class AccelerationUnits = m_per_s2
>
class Gravity;
```

The gravity class has six compile-time arguments, with the last five defaulted. The first argument, `Value`, is the value type to be used. For full accuracy of the WGS84 model, this should be either `double` or `long double`, but it can be `float` for less demanding applications or even user-defined types ¹.

The second argument, `VectorCoordinates`, is currently limited to *None*, *ECEF*, or *NEU* with *None* being the default. If the vector coordinates are *None*, the constructed gravity object will not even include the vector accessor methods and the additional code to generate vectors will not be included. The code is added when a valid form of coordinates are selected. In addition to defaulting to magnitude only mode in the class itself, the aliases function in the same way – defaulting to magnitude only but allowing the passing of coordinates to *turn on* vector support. These valid coordinates are defined in the *GeoidCoordinates* enumeration found in the *geoid.h* header.

The third argument, `Flags`, is an integer that contains all the implemented features such as the gravity model and enabled corrections. These flag values are public static members of the `GravityConfig` class, also in the `chandra::aero` namespace of the *gravity.h* header.

TALK ABOUT THE FEATURES AND WHAT THE VALID FLAGS.

¹The `lamb`, fixed-point types will work as a type, though they require special implementations of the models and are less accurate.

Type Aliases

Constructors

Accessors

Part III

Chandra Cookbook

13 Clock and Timing

13.1 Performance Timers

Timing a function

Conditional timing of a function

Loop frequency measurement

Loop frequency statistics

14 Digital I/O

14.1 Reading a digital pin

14.2 Writing to a digital pin

14.3 Reading a mechanical switch/button

14.4 Manual control of an LED

14.5 Automatic PWM of an LED

14.6 Automatic PWM of an LED with asynchronous update

14.7 Event on a pin falling (rising) edge

14.8 Reading a Quadrature Encoder

15 Analog I/O

15.1 Raw reading of a single ADC channel

15.2 Scaled reading of a single ADC channel

15.3 Reading multiple ADC channels

16 Math

16.1 Conuct a matrix

16.2 Solve a system of equations

16.3 Invert a matrix

17 Control

17.1 Kalman

17.2 PID

17.3 IMU Calibration

18 Aero

18.1 Atmosphere

Pressure and density at a known altitude

Altitude from pressure

TODO: do both in standard conditions and in non-standard conditions.

18.2 Gravity

Acceleration due to gravity on the Earth's surface

Gravitational acceleration above Earth's surface

Part IV

Appendices

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.

$$\vec{y}_c = \mathbf{W}\vec{y}_m + \vec{b} \quad (\text{C.1})$$

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 matrices 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,

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

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,

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

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,

$$\vec{b}(T) = \frac{(\vec{b}_2 - \vec{b}_1)}{(T_2 - T_1)}(T - T_1) + \vec{b}_1 \quad (\text{C.4})$$

and,

$$\mathbf{W}(T) = \frac{(\mathbf{W}_2 - \mathbf{W}_1)}{(T_2 - T_1)}(T - T_1) + \mathbf{W}_1 \quad (\text{C.5})$$

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 there 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,

$$\vec{y}_c = \mathbf{S}\vec{y}_m + \vec{b} \quad (\text{C.6})$$

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,

$$\vec{y}_c = \vec{y}_m + \vec{b} \quad (\text{C.7})$$

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,

$$\vec{y}_c = \mathbf{W}\vec{y}_b \quad (\text{C.8})$$

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,

$$\vec{y}_c = \mathbf{W}(\vec{y}_m + \vec{b}_m) \quad (\text{C.9})$$

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

$$\mathbf{W}(\vec{y}_m + \vec{b}_m) = \mathbf{W}(\vec{y}_m) + \vec{b} \quad (\text{C.10})$$

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

$$\mathbf{W}\vec{y}_m + \mathbf{W}\vec{b}_m = \mathbf{W}(\vec{y}_m) + \mathbf{W}\mathbf{W}^{-1}\vec{b} \quad (\text{C.11})$$

and simplifying,

$$\vec{b}_m = \mathbf{W}^{-1}\vec{b} \quad (\text{C.12})$$

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.

C.4 Calculation of Calibration Vectors

The calibration vectors for different forms of calibration are calculated from the known field strengths and the the reference orientation. For static calibration, the known field strengths include the magnetic and gravitational fields. The reference orientations depend upon the form of calibration as defined in the following sections. The simplest method of calibration (from the standpoint of calculating vectors) is an orthogonal cubic vector system. Using the same calibration frame, it is also possible to do an orthogonal cubic/octohedral calibration. Both of these options provides enough measurements to do a full 12-parameter calibration of each sensor. For a minimal calibration (3-parameter), a tetrahedral calibration frame is sufficient. As such, it would be good to provide that as an option.

The vector orientations are described using yaw-pitch-roll Tait-Bryant angles in radians. Within the code, these orientations should be calculated in a singularity-free method. For ease of explanation, however, the Tait-Bryant angles are sufficient.

Orthogonal Cubic Vectors

INSERT A DRAWING OF THE CALIBRATION FRAME WITH THE AXES MARKED.

Sample	Yaw (rad)	Pitch (rad)	Roll (rad)
0	0	0	0
1	0	0	$\pi/2$
2	0	$\pi/2$	0
3	0	$-\pi/2$	0
4	0	0	$-\pi/2$
5	0	π	0

Orthogonal Cubic/Octagonal Vectors

INSERT A DRAWING OF THE CALIBRATION FRAME WITH THE AXES MARKED.

Sample	Yaw (rad)	Pitch (rad)	Roll (rad)
0	$-\pi/4$	0.615479708671	0
1	$\pi/4$	0.615479708671	0
2	$3\pi/4$	0.615479708671	0
3	$-3\pi/4$	0.615479708671	0
4	$-3\pi/4$	3.75707236226	0
5	$3\pi/4$	3.75707236226	0
6	$\pi/4$	3.75707236226	0
7	$-\pi/4$	3.75707236226	0

The pitch rotation is based on placing the octagon corner vertical. As such, the base rotation required can be calculated as,

$$\alpha_{pitch} = \text{atan}(1/\sqrt{2}) + \pi \quad (\text{C.13})$$

with, the y value (1) being the normalized height of the calibration frame and the x value ($\sqrt{2}$) being the length of the frame diagonal. For the upper corners, this value can be used directly while π needs to be added to the value for the bottom corners.

CHECK THIS!!!!

Tetrahedral Vectors

NOTE: FOR THE MOST EFFICIENT CALIBRATION, THE IMU IS NOT PARALLEL TO ANY OF THE TETRAHEDRON'S FACES.

INSERT A DRAWING OF THE CALIBRATION FRAME WITH THE AXES MARKED.

INSERT THE TABLE OF VECTORS.

D References

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