Scorpion: A Modular Sensor Fusion Approach for Complementary Navigation Sensors

Kyle Kauffman[1], Daniel Marietta[2], John Raquet[3], Daniel Carson[4], Robert C. Leishman[5], Aaron Canciani[6], Adam Schofield[7], Michael Caporellie[8]

***Abstract*—There is a great need to decrease our reliance on GPS by utilizing novel complementary navigation sensors. While a number of complementary navigation sensors have been studied, each one has trade-offs in availability, reliability, accuracy and applicability in various environments. The development of a robust estimator therefore requires the integration of many diverse sensors into a sensor fusion platform. Unfortunately, as the number of sensors added to the system grows larger, so does the difficulty of developing a sensor fusion solution that optimally integrates them all into a single navigation estimate. In addition, a sensor fusion solution with many sensors is susceptible to sensor failures, modeling errors, and other phenomena which can cause degradation of the fusion solution.**

In this paper, we propose an open architecture for sensor fusion that allows for the development of modular navigation filters, sensor integration strategies, and integrity algorithms. The primary goal of this architecture is to allow for the rapid development of a novel complementary PNT sensor, fusion strategy, or integrity algorithm without modification of any other part of the system. In the future, this architecture will enable the community to develop a repository of well-tested software modules for sensor fusion which will in turn allow for the iterative development of robust estimators, where users may pick and choose the components that they wish to use from the repository and build an estimator that fits their application. In addition, domain experts in the community on a particular sensor phenomenology may contribute modules to the repository without needing to be experts in all aspects of sensor fusion. To facilitate this community engagement, we have developed an open source implementation of the architecture, which will be made available as a reference implementation of the architecture and approach. This paper details the design and overall approach to the open architecture, as well as shows some experimental results that were obtained by running flight data through the reference implementation.

***Index Terms*—Bayesian Filter, Navigation, Kalman Filtering, Open Architecture, Modular, Sensor Fusion**

I. Introduction

Modern navigation systems are expected to provide high-accuracy solutions in a wide variety of environments. The traditional integrated inertial navigation system (INS)/GPS navigation system has limited ability to operate in many common scenarios where GPS service is degraded or not available, such as in urban canyons and indoors [1]. The typical approach to allow these navigation systems to continue operating under these conditions is to augment the GPS/INS system with an additional set of sensors, which complement the baseline system. The data extracted from these complementary position, navigation, and time (PNT) sensors is combined with the measurements from the GPS/INS system to produce an optimal estimate of a navigation system's location via a sensor fusion algorithm.

Many complementary PNT sensors have been analyzed and evaluated by the community, including light detection and ranging (LIDARs), radars, electro-optical/infra-ref (EO/IR) cameras, barometers, magnetometers, and many others [2]–[8]. Each of these sensors has advantages as well as different phenomena which limit their operation. For example, optical sensors are easily obstructed by cloud cover or weather, whereas radio frequency (RF) sensors can penetrate cloud cover. The addition of a larger set of alternative sensors allows the navigation platform to operate under more diverse environmental conditions. Each additional sensor also provides a source of new information about the vehicles' navigation solution, resulting in a lower error bound on the overall navigation solution. Thus the combination of a wide variety of sensors results in a better and more robust navigation solution.

Unfortunately, as the number of complementary sensors added to a navigation system grows, so too does the complexity of integrating them and developing a sensor fusion algorithm that can process all of the available information. Typically, sensor fusion algorithms are custom-developed for a specific set of sensors. The addition of a new sensor to a working navigation solution requires careful redesign of the algorithm to optimally use the new information. Alternatively, the new sensor's measurements can be processed separately from the primary sensor fusion algorithm into a measurement that is easy to incorporate into the primary sensor fusion algorithm, such as a position update. This approach, referred to as loosely-coupled sensor fusion, limits the ability of the sensor fusion algorithm to optimally extract information from the sensor data.

In this paper, we present a modular approach to developing tightly-coupled sensor fusion algorithms that allows for rapid algorithm development with a varying set of sensors. With our approach, sensor modules are developed for each

[1]Kyle Kauffman, *IS4S*, Beavercreek, OH, USA, kyle.kauffman@is4s.com
[2]Daniel Marietta, *IS4S*, Beavercreek, OH, USA, daniel.marietta@is4s.com
[3]John Raquet, *IS4S*, Beavercreek, OH, USA, john.raquet@is4s.com
[4]Daniel Carson, *IS4S*, Beavercreek, OH, USA, daniel.carson@is4s.com
[5]Robert C. Leishman, *Air Force Institute of Technology*, WPAFB, OH, USA, robert.leishman@afit.edu
[6]Aaron Canciani, *Air Force Institute of Technology*, WPAFB, OH, USA, aaron.canciani@afit.edu
[7]Adam Schofield, *CCDC/C5ISR*, APG, MD, USA,
[8]Michael Caporellie, *CCDC/C5ISR*, APG, MD, USA,

complementary PNT sensor. Each sensor module describes two things: 1) a mathematical model of the relationship of the sensor data to the navigation system's position, velocity, attitude, and 2) a stochastic model of time-varying errors necessary to capture the dynamic noise/errors inherent in the sensor's phenomenology. Crucially, a sensor module for a particular sensor has an isolated mathematical model which has no dependency on any other sensor existing in the system. The module is specified without requiring knowledge of the existence or non-existence of system states that are not related to the module. In other words, the full details of the state vector do not need to be known when the sensor module is defined. This allows for a navigation system designer to rapidly change the set of sensors they want in their system by selecting a set of sensor modules that match the desired complementary sensor set. Once a set of modules has been selected, the designer can combine them with a core filtering algorithm to construct a sensor fusion algorithm capable of incorporating data from any of the sensors described by the selected modules. Like the sensor modules, the core filtering algorithm is also pluggable, and can be swapped out with another algorithm without changing anything else in the system. For example, only one line of code needs to be changed in order to switch from an extended Kalman filter to an unscented Kalman filter.

The sensor module's mathematical model is designed to allow advanced tightly coupled integrations that would be poorly represented by loosely-coupled approaches, such as time difference of arrival (TDOA), differential carrier phase integration, and scalar map-based geo-localization from non-linear maps. Thus our modular approach achieves the performance of tightly integrated approaches but without the burden of manual integration required to build a traditional non-modular tightly coupled sensor fusion algorithm.

In this paper, we will first describe the overall modular approach and design. We'll then discuss the design of the sensor modules, including stateful representations, solution integrity, computational performance, filter performance, and model flexibility. Finally, we will develop the design and integration of several common sensor modules, including examples of their use.

II. Background

A. Previous Work

Filtering is a well known approach to sensor fusion that has been described in many textbooks [9]–[11] and articles [12]–[14]. It is an active research area with many new techniques being developed continuously [15]–[17]. Due to the level of activity in the community developing new estimation filters, an optimal filtering approach picked today for a particular application may become quickly outdated.

There are many different communities which approach sensor fusion from different perspectives. For example, the robotics community often considers optimal estimation from the perspective of Bayesian networks, often in the form of a factor graph [18]. The navigation community often formulates the problem of sensor fusion as a state observer in a control system [9]. Throughout this paper, we will formulate the problem as a state space representation consistent with [9], but substituting a shorthand notation of $\mathbf{x}_k$ for the value of $\mathbf{x}$ at time t_k , $\mathbf{x}(t_k)$ and $\mathbf{f}_k(...)$ for time-dependent functions $\mathbf{f}(t_k, ...)$.

There are several software libraries that are designed to aid users in the development of a state estimator, such as Bayes++ [19], the Matlab®Control System Toolbox®, and others [20], [21]. These focus on providing well-tested implementations of known algorithms to the user. For example, pykalman [21] provides implementations of the unscented Kalman filter (UKF) and extended Kalman filter (EKF). However, neither it nor the other libraries provide any facilities to help a user merge two different filters written by two different authors with two different sensor sets. In general, a user wishing to perform this task would need to be an expert in the design of both filters and manually merge the two filters by careful analysis of their shared and distinct states. In contrast, our proposed approach allows for filter algorithms to be represented in a way that would allow a user to take two independently developed sensor integration strategies and merge them into a single filter without requiring further analysis of the components.

B. Problem Formulation

The problem scope we are considering in this paper is estimating a set of time-varying values $\mathbf{x}$ given a set of observations $\mathbf{z}$ which contain information about $\mathbf{x}$. Let $\mathbf{x}_k$ be the $M \times 1$ vector representing the value of $\mathbf{x}$ at time t_k and $\mathbf{z}_k$ be the $N \times 1$ set of observations collected at time t_k. Then we assume that the way $\mathbf{x}$ changes from one time epoch to the next is well-modeled by

$$\mathbf{x}_k = \mathbf{g}(\mathbf{x}_{k-1}) + \mathbf{w}_k, \quad \mathbf{w}_k \overset{\text{iid}}{\sim} N(0, \sigma_w) \tag{1}$$

where $E[\mathbf{w}_k \mathbf{w}_k^T] = \mathbf{Q}_k$, and that the observations are related to the state vector $\mathbf{x}$ by

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k, \quad \mathbf{v}_k \overset{\text{iid}}{\sim} N(0, \sigma_v) \tag{2}$$

where $E[\mathbf{v}_k \mathbf{v}_k^T] = \mathbf{R}_k$, $\mathbf{g}$ is the *discrete-time propagation function*, $\mathbf{h}$ is the *measurement model function*, and $\mathbf{w}$ and $\mathbf{v}$ are additive white Gaussian noise (AWGN) sources. Estimators that are able to estimate parameters modeled as in (1) and (2) are known as AWGN *filters*. A designer's goal is to build filters that estimates $\mathbf{x}$ at time t_k ($\mathbf{x}_k$) given the measurement ($\mathbf{z}_k$) along with all prior received measurements ($\mathbf{z}_{k-1}, \mathbf{z}_{k-2}, \ldots$). As measurements come in, the filter will continually refine the estimate of $\mathbf{x}$ using the new information, calculating a new estimate $\mathbf{x}_{k+1}$ when the measurement $\mathbf{z}_{k+1}$ is received, and so forth.

In this paper, we will write $\hat{\mathbf{x}}_k^+$ to denote an estimate of $\mathbf{x}_k$ given the measurement at time k ($\mathbf{z}_k$) along with all prior received measurements ($\mathbf{z}_{k-1}, \mathbf{z}_{k-2}, \ldots$). We will write $\hat{\mathbf{x}}_k^-$ to denote an estimate of $\mathbf{x}_k$ given all prior received measurements ($\mathbf{z}_{k-1}, \mathbf{z}_{k-2}, \ldots$) but *not* including the measurement at time k ($\mathbf{z}_k$).

III. Modular Architecture Design

A. Overall Approach

Our current goals for Scorpion are:

- Develop a framework for building estimators that is flexible enough to support advanced research filters (ultra-tightly coupled GPS, SLAM, etc.) but remains as simple to use as possible. To do this, we support several levels of problem model complexity, ranging from the standard model (defined in II-B) to more complex models for sampled filters and factor graphs. In this paper, we will focus on the standard model.
- Build an architecture which enables both simulated/post-processing of data sets and real-time processing of sensor data.
- Enable different organizations to share modular code. If the sensor API is fully modular, two organizations may independently write their own modules for their own sensors and then later add them both to the same filter.
- Let users test the effects of different sensor integration strategies, filter types, or sensor fidelity by using pluggable modules.
- Allow access to the library from other software languages.
- Support filter deployment on performance-critical projects, such as embedded platforms, massively parallel distributed systems doing Monte Carlo analysis, and everything in-between.
- Provide a set of off-the-shelf sensor modules for common sensors than can be easily plugged into a new filter project. This will eliminate the need to reinvent the wheel for common problems like GPS/INS integration or barometric aiding.
- Make the core classes abstracted to support usages other than navigation (e.g. using a particle filter to estimate the frequency jitter of a phase lock loop (PLL)).

It is important to note that standardizing sensor messages falls outside the scope of Scorpion. This is accomplished by different projects like All Source Positioning and Navigation (ASPN).

B. Modular Filter Design

As discussed in Section II-B, the goal of a Scorpion estimator solving the standard model is to estimate $\mathbf{x}$ at time t_k ($\mathbf{x}_k$) given the measurement ($\mathbf{z}_k$) at time t_k along with all prior received measurements ($\mathbf{z}_{k-1}, \mathbf{z}_{k-2}, \ldots$).

A Scorpion user must therefore describe the following things to have a fully described estimator:

- The number of states x in our state space, including both quantities we wish to estimate and nuisance states that are required to well-model the system but are not ultimately of interest.
- The dynamics of how those states propagate forward in time, described by $\mathbf{g}$ and $\mathbf{w}$ (known as the dynamics model).
- The method by which a raw measurement $\mathbf{z}_k$ is related to the state vector x, described by $\mathbf{h}$ and $\mathbf{v}$ (known as the measurement model).
- The filtering algorithm one wants to apply to compute $\hat{\mathbf{x}}_k^+$ from $\hat{\mathbf{x}}_k^-$ (known as the update strategy)
- The filtering algorithm one wants to apply to compute $\hat{\mathbf{x}}_k^-$ from $\hat{\mathbf{x}}_{k-1}^+$ (known as the propagate strategy)

In order to facilitate modularity, Scorpion breaks down the description of these quantities and algorithms into three main pieces: StateBlocks, MeasurementProcessors, and filters.

A StateBlock represents a collection of states whose propagation is not dependent on any states outside of the given StateBlock. This means that the size of a StateBlock could range from a single state (e.g. an estimate of a sensor bias) to several states (e.g. the estimate of errors in an inertial solution). StateBlocks are required to describe $\mathbf{g}$ and $\mathbf{Q}_k$ (the discrete-time process noise covariance matrix) for the set of states contained in the block, given $\mathbf{x}_k$, t_k, and t_{k+1}. In addition, to facilitate certain estimators, such as the EKF, which require linearization points, the StateBlock provides the Jacobian of $\mathbf{g}$ (first-order Taylor series expansion) which is denoted $\mathbf{\Phi}$.

A MeasurementProcessor represents the relationship of a measurement to the state vector. It must produce $\mathbf{z}_k$, $\mathbf{h}(\mathbf{x})$, and $\mathbf{R}_k$ (the measurement noise covariance matrix) from a raw sensor measurement, $\mathbf{x}_k$, and $\mathbf{P}_k$) (the covariance associated with $\mathbf{x}_k$). Additionally, to facilitate certain estimators like the EKF which require linearization points, the StateBlock provides the Jacobian of $\mathbf{h}$ (first-order Taylor series expansion) which is denoted $\mathbf{H}$.

Finally, a StandardFilter is an iterative sensor fusion engine which can take in a list of one or more StateBlocks and MeasurementProcessors as well as the raw measurements from a sensor to produce an estimate of all the states in every StateBlock at the current time. It consists of two components: a pluggable measurement update strategy (i.e. calculating $\mathbf{x}_k^+$ from $\mathbf{x}_k^-$) and a pluggable propagation strategy (i.e. calculating $\mathbf{x}_k^-$ from $\mathbf{x}_{k-1}^+$).

The basic interactions between these modules can be seen in Fig. 1. This figure shows the flow of data which occurs when a filter is sent a measurement. First the filter checks if the current filter estimate is up to date with the measurement time of validity. If it isn't, then the filter requests the dynamics model from each StateBlock and propagates the states by sending the dynamics model to the filter's propagation strategy.

Then, once the filter has a state estimate and covariance at the same time as the raw sensor measurement, it finds the MeasurementProcessor specified by the measurement. The filter then sends the measurement and state information to the MeasurementProcessor and requests the measurement model from it that describes how the measurement relates to the states. The filter then computes the updated state by passing the measurement model to its update strategy, yielding the estimator's solution incorporating the measurement.

The data flow and operations performed by a StateBlock can be seen in Fig. 2. The center column represents the StateBlock's memory and processes, while the outside columns rep-

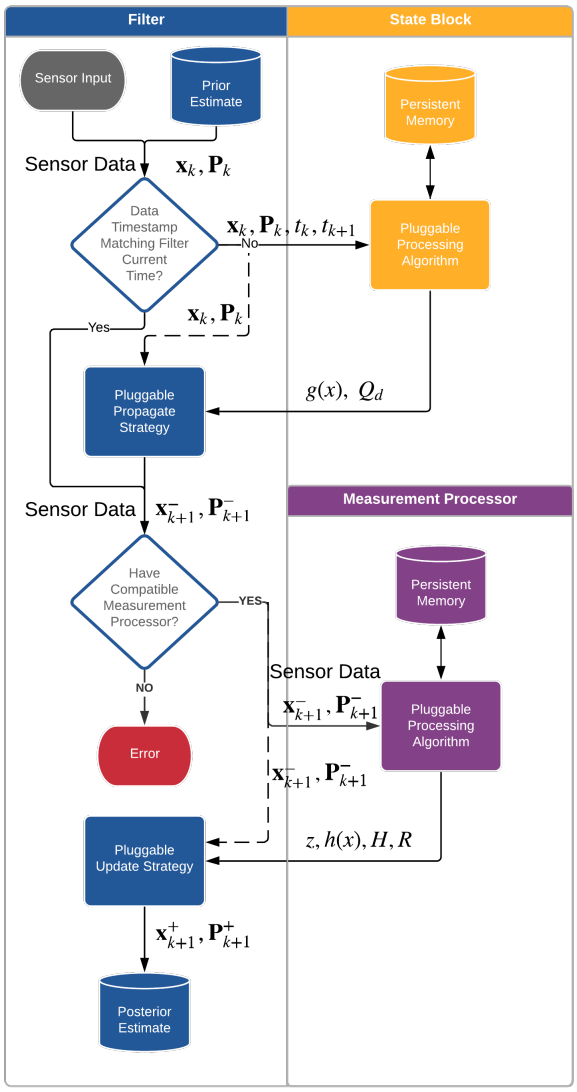

Fig. 1. Propagation and Update for a Single StateBlock and Single Measurement Processor. Note that control flows along the path of the solid-line arrows. The dotted line arrows indicate that the data computed at the origin of the arrow is also used at the arrow's destination.

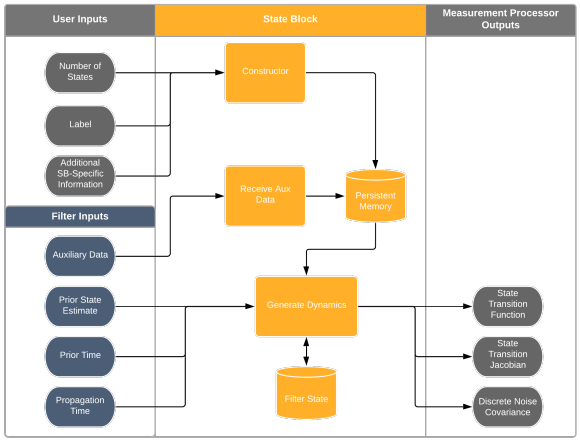

Fig. 2. Data Flow in a StateBlock

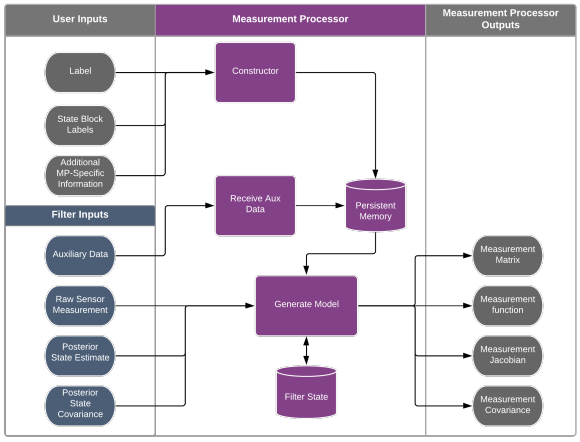

Fig. 3. Data Flow in a MeasurementProcessor

resent externals inputs and outputs. The StateBlock is created by the user supplying a series of inputs to the constructor. All StateBlocks are required to specify their number of states and a label, which is a unique identifier. This way multiple state blocks can be instantiated and added to the same filter.

All StateBlocks are required to implement a "receiveAuxData" function. This serves as a channel for getting arbitrary data into the StateBlock asynchronously after its construction. For example, a StateBlock which models the errors in an Inertial Navigation System's solution might need to know information about the solution to propagate the errors.

The core of the StateBlock is the "generateDynamics" function. This function is called by the filter with a prior state estimate, the time of the prior state estimate, and the time the filter wants to propagate the state to. Some filtering problems require the StateBlocks to be able to add states to the filter or remove states from the filter. If a reference to the filter was passed to the StateBlock, either through its constructor on initialization or via auxiliary data, then the StateBlock can modify the filter as needed.

The data flow and operations performed by a MeasurementProcessor can be seen in Fig. 3. Similarly to Fig. 2, the center column represents the MeasurementProcessor's memory and processes, while the outside columns represent externals inputs and outputs. All MeasurementProcessors must declare a label (unique identifier) that can be used to refer to it, and a list of labels of every StateBlock it is related to.

Like the StateBlock, the MeasurementProcessor must also implement a "receiveAuxData" function to receive any additional information required by the MeasurementProcessor to produce the measurement model.

The core of the MeasurementProcessor is the "generateModel" function. It is called by the filter with a raw measurement and a set of propagated states. This set of states is

```
val filter = SensorEKF()
```

Listing 1: Creating an EKF.

determined by the list of StateBlock labels that the MeasurementProcessor declared it would update. This function processes the sensor measurement and produces a measurement model, which the filter uses to update the states. If needed, the MeasurementProcessor may also add or remove StateBlocks from the filter. For example, a simultaneous localization and mapping (SLAM) processor might need to add states to track new targets that it had just discovered in the latest raw sensor data.

Note that directly specifying the set of StateBlock labels that a measurement processor relates to has the potential to couple a MeasurementProcessor to a particular filter design. For example, consider the example given at the end of Section II-A, wherein two users had built different filters but had chosen different StateBlocks to represent their states in their respective filters. If the StateBlocks chosen by the two users are different, then a MeasurementProcessor would have to choose which StateBlock it relates to in its list of StateBlock labels, making it incompatible with the other StateBlock. Because our goal is to have modular MeasurementProcessors that are independent of which state representations (StateBlocks) are chosen to be in the filter, this is less than ideal. See Section III-D for our solution to decoupling MeasurementProcessors for the particular StateBlocks loaded into a filter.

C. Example: Estimating Altitude

We will now illustrate how the Scorpion approach applies to PNT estimation with an example. Consider the problem of estimating the altitude of a vehicle. Suppose we have a altimeter sensor which gives us a measurement of the altitude but which contains a bias. In order to estimate the altitude we could model two states: one for the altitude itself and one to estimate the bias in the altimeter measurements. First we'll define an EKF, as shown in Listing 1.

"SensorEKF" is an implementation of the standard model, and contains an implementation of both an update strategy and propagate strategy. However, the filter initially contains no state blocks or measurement processors, as it is up to the user to decide the design of the filter by adding modules into the filter. We will therefore define a StateBlock for our altitude as shown in Listing 2.

This StateBlock inherits from the class "StateBlock", which means it is required to implement the "numStates" and "label" properties and the "receiveAuxData" and "generateDynamics" functions. In the constructor arguments, "label" is a string which serves as a unique identifier or name of the StateBlock and "Q" is a value specific to this StateBlock and sets the sigma of the random walk.

Next we'll consider the properties this StateBlock defines. The value "numStates" is set to 1, indicating to the filter that this StateBlock contains one state. For clarity and convenience we have set a variable "Phi" to store the Jacobian of $\mathbf{g}(\mathbf{x})$, or $\mathbf{\Phi}$.

```
class AltitudeState(override var label: String,
        var Q: Double) : StateBlock {
    override var numStates = 1
    var Phi = mat[1.0]

    override fun receiveAuxData(auxData: Any) {}

    override fun generateDynamics(xhat: Matrix<Double>,
            timeFrom: Timestamp, timeTo: Timestamp):
            Dynamics {

        val dt = timeTo.time - timeFrom.time

        val Qd = mat[Q * dt]

        fun g(xhat: Matrix<Double>){return Phi * xhat}

        return Dynamics(g, Phi, Qd)
    }
}
```

Listing 2: Sample Altitude StateBlock.

```
val altitudeStateBlock = AltitudeState("altitude", 10.0)
filter.addStateBlock(altitudeStateBlock)
filter.setStateBlockEstimate(mat[100])
filter.setStateBlockCovariance(mat[10])
```

Listing 3: Adding an Altitude StateBlock.

All StateBlocks are required to implement the method "receiveAuxData". This method allows the user to send the state block arbitrary data. In this case the state block does not need any additional data to compute the dynamics model so it will not be used.

Finally, the StateBlock is required to implement "generateDynamics". This is the core function of the StateBlock which generates the dynamics model for the filter so the filter can propagate the state. $\mathbf{Q}$ is produced by multiplying "Q" by the elapsed time. $\mathbf{g}(\mathbf{x})$ is defined as a function which, given $\mathbf{x}_k$, returns $\mathbf{\Phi}\mathbf{x}_k$. Since we previously defined $\mathbf{\Phi}$ to be equal to 1, the state estimate will not change over time when propagated. Last, a dynamics model is constructed from these terms and returned to the filter.

We have now fully described the StateBlock. Next, we can create an instance of it and add it to the filter. By default the estimate and covariance are initialized to zero. For the purpose of this example we will assume that we have some a priori information about about our altitude, which is that the initial altitude estimate is 100 meters and covariance is 10 meters squared. This is shown in Listing 3.

Next we will create a StateBlock to model the altimeter bias as a random walk, as shown in Listing 4. This StateBlock is very similar to the previous one. Since the bias is constant and, by extension, the state covariance does not change during propagation, there is no "Q" in the constructor and $\mathbf{Q}_d$ is set to a constant "0.1".

We'll again create an instance of this state block and add it to the filter and set it to an initial covariance of 100, shown in Listing 5. We have now described a set of states, including their initial conditions and propagation model. Next we will need to define a MeasurementProcessor to describe to the filter how it can extract information from the raw altimeter measurement, shown in Listing 6.

Similar to the StateBlock, the MeasurementProcessor needs

```
class ConstantBiasState(override var label: String):
        StateBlock {

    override var numStates = 1
    var Phi = mat[1.0]
    var Qd = mat[0.1]

    override fun receiveAuxData(auxData: Any) {}

    override fun generateDynamics(xhat: Matrix<Double>,
            timeFrom: Timestamp, timeTo: Timestamp):
            Dynamics {

        fun g(xhat: Matrix<Double>){return Phi * xhat}

        return Dynamics(g, Phi, Qd)
    }
}
```

Listing 4: Sample Altitude Bias StateBlock.

```
val biasStateBlock = ConstantBiasState("altimeterBias")
filter.addStateBlock(biasStateBlock)
filter.setStateBlockCovariance(mat[100])
```

Listing 5: Adding Altitude Bias StateBlock.

its own unique identifier, which is "label". The MeasurementProcessor, however, also needs the labels of the StateBlocks which contain the states the MeasurementProcessor is updating (in the interest of simplicity, for this example we are not using aliasing– instead the MeasurementProcessor directly relates to the StateBlock known to be already loaded into the filter). This is set using the "stateBlockLabels" parameter in the constructor.

The inputs to the "generateModel" function are a measurement, the state estimate, and the state covariance for the MeasurementProcessor to use when calculating its MeasurementModel. The covariance is unused in this MeasurementProcessor.

This measurement processor defines $\mathbf{z}$ as the altitude measured by the altimeter. $\mathbf{H}$ is a matrix which directly maps both states to the measurement. $\mathbf{h}(\mathbf{x})$ is a function which, given $\mathbf{x}_k$, returns $\mathbf{H}\mathbf{x}_k$. Last the measurement covariance matrix is extracted from the measurement reported by the altimeter.

Now that we've defined the MeasurementProcessor, we need to add it to the filter, shown in Listing 7.

We now have a filter that is modeling a set of states for estimating altitude and which includes the necessary information to incorporate measurements from a altimeter. We can

```
class AltitudeMeasurementProcessor(
        override var label: String,
        override var stateBlockLabels: Array<String>):
        MeasurementProcessor{

    override fun generateModel(meas: Measurement,
            xhat: Matrix<Double>, P: Matrix<Double>):
            MeasurementModel {

        val z = meas.altitude
        val H = mat[1, 1] // Creates a 1x2 matrix of ones
        fun h(xhat: Matrix<Double>){return H * xHat}
        val R = meas.variance

        return MeasurementModel(z, h, H, R)
    }
}
```

Listing 6: Sample Altitude MeasurementProcessor.

```
val altimeterProcessor = AltitudeMeasurementProcessor(
    "altimeter", ["altitude", "altimeterBias"])
filter.addMeasurementProcessor(altimeterProcessor)
```

Listing 7: Initializing Altitude MeasurementProcessor.

```
val BARO_SIGMA = 10
altimeterMeasurement = Measurement("altimeter",
    createAltitude(120.0, BARO_SIGMA * BARO_SIGMA,
    Time(10.0)))
filter.update(altimeterMeasurement)
```

Listing 8: Incorporating an Altitude Measurement.

therefore send the filter a measurement and it will produce an updated estimate using the measurement, shown in Listing 8.

The "altimeter" string tells the filter to send this measurement to the measurement processor with the same label. We create an measurement with given the altitude measurement ("120.0"), covariance of the measurement ("BARO_SIGMA * BARO_SIGMA") and a time for which the measurement is valid ("Time(10.0)"). In a real application the altimeter data would be captured from the sensor instead of simulated as we've done here.

When we created the filter it defaulted to an initial time of 0.0 seconds. When we send the filter this measurement, it will first propagate its states to 10 seconds then apply the measurement as an update. We can the retrieve the updated altitude and bias states by the method shown in Listing 9.

D. Aliasing

As discussed at the end of Section III-B, MeasurementProcessors having direct references to a specific set of StateBlocks and the representation/layout of the states within those StateBlocks leads to coupling between the MeasurementProcessor and a particular state space representation. Consider for example a MeasurementProcessor developed by a user that assumes the filter contained a set of states representing position in "latitude, longitude, altitude" (LLH) representation. Now suppose another filter developer designed a filter to estimate position using Earth-centered Earth-fixed (ECEF) states instead. When the first user attempted to load their MeasurementProcessor into the second developer's filter, the MeasurementProcessor would not find the set of states that it required in the filter (LLH), and consequently would not be compatible with the second developer's filter.

In this section, we will define a new concept called "State Aliases", which allow MeasurementProcessors to operate inside of filters which do not have the states they expect to be present. In the above example, an alias would be developed that maps ECEF coordinates into LLH coordinates. Then, the alias would be loaded into the filter. The filter would now contain a concrete StateBlock, represented in ECEF coordinates, and a *virtual* StateBlock, which contained the

```
val altitudeEst = filter.getStateBlockEstimate("altitude")
val altitudeCov = filter.getStateBlockCovariace("altitude")
val biasEst = filter.getStateBlockEstimate("altimeterBias")
val biasCov = filter.getStateBlockCovariace("altimeterBias")
```

Listing 9: Retrieve the updated altitude and bias states.

same position as the concrete one but represented in LLH. The LLH position solution is directly computed from the ECEF StateBlock, and thus it is just an *alias* for the concrete block, not a new StateBlock with its own sets of estimates. However, MeasurementProcessors are free to depend on aliased (virtual) StateBlocks, which decouples the implementation of MeasurementProcessors from StateBlocks.

Next we'll develop the mathematical model for our aliasing approach. In order to incorporate a set of measurements represented by the vector $\mathbf{z}$ into a filter, the user must be able to relate the measurements to the state vector. Assuming independent, zero-mean AWGN, the *measurement function* that relates N states to M measurements takes the form

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k) + \mathbf{v}_k \tag{3}$$

with the $M \times N$ *measurement matrix* defined as the matrix of partial derivatives of $\mathbf{h}$ with respect to $\mathbf{x}$

$$\mathbf{H}_k(\hat{\mathbf{x}}_k^-) \triangleq \left.\frac{\partial \mathbf{h}_k(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}_k^-} \tag{4}$$

Suppose that a user has a previously developed measurement function

$$\mathbf{z}_k = \mathbf{g}_k(\mathbf{y}_k) + \mathbf{v}_k \tag{5}$$

written against a state vector $\mathbf{y}$ that they wish to incorporate into a MeasurementProcessor, where $\mathbf{y}$ is a set of states related to $\mathbf{x}$ by some arbitrary function $\mathbf{s}()$ such that

$$\mathbf{y}_k = \mathbf{s}_k(\mathbf{x}_k) \tag{6}$$

As noted in the introduction of this section, to incorporate this model into a filter using a state representation $\mathbf{x}$ without aliasing, the user would need to adapt $\mathbf{g}(\mathbf{y})$ to relate to $\mathbf{x}$ instead. Alternatively, they could create a StateBlock representing $\mathbf{y}$ and adjust any other modular elements of the system to work with $\mathbf{y}$ as well. In either case, the model would still be coupled to a specific state representation, but aliasing allows the incorporation of the model in its original form without this coupling.

To perform aliasing the user constructs a VirtualStateBlock class which enables a real StateBlock to be converted into an alternate representation. A simple VirtualStateBlock implementation that just scales a 1-element StateBlock, such as might be done for a conversion of units used for a particular state, is shown in Listing 10.

Each instance of this class possesses a label `target` by which it is referenced, as well as the label `current` of the concrete StateBlock it converts. In other words, this class transforms a StateBlock from the current representation to some target representation. When some filter element requests a StateBlock with the `target` label, the filter looks for a VirtualStateBlock with this label and passes the actual StateBlock estimate and covariance values through the `convert` function and returns the result to the requesting element. Listing 11 demonstrates how to add an instance of the VirtualStateBlock shown in Listing 10 to a sample setup.

```
class ScaledVsb(VirtualStateBlock):
    def __init__(self, c, t, s):
        self.current = c
        self.target = t
        self.scale = s

    def convert(self, ec):
        ec.estimate *= self.scale
        ec.covariance *= self.scale**2
        return ec

    def jacobian(self, x):
        return array([[self.scale]])
```

Listing 10: Sample Scaling VirtualStateBlock.

```
filter = SensorEKF(start_time)
# Actual block the filter will maintain
block = Single('real')
filter.addStateBlock(block)

vsb = Scaled('real', 'scaled', 0.5)
my_filter.addVirtualStateBlock(vsb)

# Can now do either
filter.getStateBlockEstimate('real')
filter.getStateBlockEstimate('scaled')

# Measurement can now be taken against
# aliased state
proc = SingleProcessor('proc', 'scaled')
filter.addMeasurementProcessor(proc)
```

Listing 11: VirtualStateBlock Usage.

Continuing with the example of a user provided measurement function $\mathbf{g}_k(\mathbf{y}_k)$, we show how the extended Kalman filter update equations are modified to allow substitution of $\mathbf{g}(\mathbf{y})$ for $\mathbf{h}(\mathbf{x})$. First we recall the standard EKF update equations as presented in [**?**]:

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}(\mathbf{z}_k - \mathbf{h}_k[\hat{\mathbf{x}}_k^-]) \tag{7}$$

$$\mathbf{P}_k^+ = \mathbf{P}_k^- - \mathbf{K}_k\mathbf{H}_k\mathbf{P}_k^- \tag{8}$$

$$\mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^{\mathbf{T}}{}_k[\mathbf{H}_k\mathbf{P}_k^-\mathbf{H}^{\mathbf{T}}{}_k + \mathbf{R}_k]^{-1} \tag{9}$$

with

$$\mathbf{H}_k \triangleq \mathbf{H}[t_k; \hat{\mathbf{x}}_k^-] \tag{10}$$

Beginning by substituting (6) into (5) and setting equal to (3) we obtain

$$\mathbf{h}_k(\mathbf{x}_k) = \mathbf{g}_k[\mathbf{s}_k(\mathbf{x}_k)] \tag{11}$$

and therefore from (4)

$$\mathbf{H}_k(\hat{\mathbf{x}}_k^-) \triangleq \left.\frac{\partial \mathbf{g}_k(\mathbf{s}_k(\mathbf{x}_k))}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}_k^-} \tag{12}$$

which is evaluated via the chain rule to get

$$\mathbf{G}_k(\hat{\mathbf{x}}_k^-) = \mathbf{H}_k(\hat{\mathbf{x}}_k^-) = \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}\bigg|_{\mathbf{x}=\hat{\mathbf{x}}_k^-} \tag{13}$$

In (13) $\frac{\partial \mathbf{g}}{\partial \mathbf{y}}$ is provided directly by the user model, and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is just the Jacobian matrix of the aliasing function which is supplied separately to the filter, each of which can be independently evaluated. All that remains is to substitute (11) and (13) into (7), (8) and (9). Letting

$$\mathbf{G}_k(\hat{\mathbf{y}}_k^-) = \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \tag{14}$$

and

$$\mathbf{S}_k(\hat{\mathbf{x}}_k^-) = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \tag{15}$$

we obtain

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - \mathbf{g}_k[\mathbf{s}_k(\hat{\mathbf{x}}_k^-)]) \tag{16}$$

$$\mathbf{P}_k^+ = \mathbf{P}_k^- - \mathbf{K}_k\mathbf{G}_k\mathbf{S}_k\mathbf{P}_k^- \tag{17}$$

$$\mathbf{K}_k = \mathbf{P}_k\mathbf{S}_k^{\mathrm{T}}\mathbf{G}_k^{\mathrm{T}}[\mathbf{G}_k\mathbf{S}_k\mathbf{P}_k^-\mathbf{S}_k^{\mathrm{T}}\mathbf{G}_k^{\mathrm{T}} + \mathbf{R}_k]^{-1} \tag{18}$$

Unfortunately, the efficacy of a VirtualStateBlock is bound by the same constraints that apply to the filter in which it is used. Aliasing the covariance matrix in a linearized filter like an EKF is performed using the Jacobian of the aliasing function, so such a transformation is only valid to first order. Certain operations, like setting the initial conditions of a VirtualStateBlock are also not possible due to the fact that the transform functions may not be invertible. Despite these limitations, for StateBlocks which are approximately linearly related aliasing can lower the coupling between MeasurementProcessors and StateBlocks.

E. Advanced Filters

One of the goals of Scorpion is to support the building of a library of filters, similar to the capabilities of other navigation libraries mentioned in Section II-A, but also including much more advanced filter types such as factor graphs and Monte-Carlo methods. In order to incorporate these filters into scorpion, the standard model, described in Section II-B, is augmented with several more advanced models, which model more advanced filtering models such as sampled state representations. These advanced models are out of scope for this paper, however due to the modular nature of Scorpion the advanced filters which implement more advanced filter models also implement the standard model. It is therefore possible to formulate the problem according to the linearized AWGN assumptions of Section II-B but still utilize a filter that uses a more advanced model then an EKF or UKF. In order to interact with one of these more advanced filters, one often must utilize a few additional methods outside of the set of methods that are defined in the standard model. In this section, we will demonstrate the usage of the Rao-Blackwellized Particle Filter [22] (RBPF) implementation included in Scorpion via the standard model and a few additional methods to configure the filter.

```
filter.markAsParticle("pinson15", listOf(0,1))
```

Listing 12: Marking States as Particles.

```
val filter = SensorRBPF(numPart = 5_000)
```

Listing 13: Setting Particle Number in RBPF Constructor.

The RBPF is able to estimate systems with states that have arbitrarily non-linear dynamics and measurement models as long as it has access to a proportionally large number of particles. Thus, the RBPF trades computational efficiency for the ability to handle problems that are intractable for a linearized filter such as the EKF. As an example, the RBPF could be used on a problem with the following measurement equation:

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{w}_k \tag{19}$$

where $\mathbf{w}$ is additive white Gaussian noise. The RBPF could not be used with the following measurement equation:

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k)\mathbf{w}_k \tag{20}$$

Where $\mathbf{w}$ is multiplicative white Gaussian noise. The RBPF could also not be used on a problem with the following measurement equation:

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{w}_{ng,k} \tag{21}$$

where $\mathbf{w}_{ng,k}$ is non-Gaussian noise.

States in the RBPF default to standard Gaussian states. This indicates that the states are propagated and updated with standard Kalman Filter operations. States are marked as particles based on their state block name. For every state block in which you wish to mark particles, you must make a separate state marking function call. Along with the state block name you must provide the state marking function with a list of states to mark as particles. This list references the states within the state block only. The first state in a given state block is 0, the second is 1, etc. Listing 12 would mark the first two states of the pinson15 state block as particle states. This command would remain the same even if there were multiple state blocks added to the filter, because the list only references states within the named state block.

Particle filters often need more tuning than traditional filters to achieve best performance. There are several tuning parameters built into the RBPF:

- Number of Particles
- Single Jacobian Mode
- Resampling Threshold
- Jitter

The number of particles is a constructor parameter for the RBPF. The default is 10,000. Computational time increases roughly linearly with the number of particles. Higher dimensional filtering problems will require more particles than low dimensional problems. Setting the number of particles is shown in Listing 13.

```
val filter = sensorRBPF(calcSingleJacobian = true)
```

Listing 14: Setting Jacobian Parameter in RBPF Constructor.

```
filter.resamplingThreshold = 0.8
```

Listing 15: Resampling Threshold.

The RBPF uses both Kalman Filter equations as well as particle operations. When using Kalman filter equations the filter needs both measurement model Jacobians as well as dynamics model Jacobians. These Jacobians are linearizations of the measurement model or dynamics model about a point. In the RBPF, this point can either be the filter mean, leading to a Single Jacobian or an individual particle state, leading to N Jacobians, where N is the number of particles. Based on the interaction of the particle states and Gaussian states, using a single Jacobian may be exactly equivalent to using N Jacobians. Other times it is not exact but close enough to make no practical difference in filter performance. The user selects which mode to operate with using a named parameter in the constructor. This is illustrated in Listing 14.

Particle filters tend to accumulate the majority of their weight in just a few particles. This problem is called sample degeneracy and is what necessitates the resampling step of a particle filter. Once a particle filter resamples, particles with low weight are removed, and copies of existing high weight particles are created. This leads to a second particle filtering issue called sample impoverishment. This occurs when particles lose diversity due to many particles sharing the exact same value. Resampling is required to stop sample degeneracy but resampling too often leads to sample impoverishment. The resampling threshold is a single number between 0 and 1 which decides how often to resample. Resampling less often helps reduce sample impoverishment. The resampling threshold can be changed as shown in Listing 15.

The default value is 0.75 and the filter will throw an error if the number is not between 0 and 1.

Jitter is an Ad-Hoc method to address the problem known as sample impoverishment. Jitter adds white Gaussian noise to the filter states after the resample step of the particle filter. The amount of noise to add is a tuning parameter chosen by the filter designer. An example code snippet is shown in Listing 16. This example adds white Gaussian noise with a variance of 5.0 to the third state (index 2) of the block "stateBlockLabel"

For multiple states we can use a mapping shown in Listing 17. By default no jitter is added to any states after a resample step.

```
filterRBPF.addJitter("stateBlockLabel", 2, 5.0)
```

Listing 16: Adding Jitter, Method 1.

```
filterRBPF.addJitter("stateBlockLabel", mapOf(2 to 5.0,
    3 to 10.0))
```

Listing 17: Adding Jitter, Method 2.

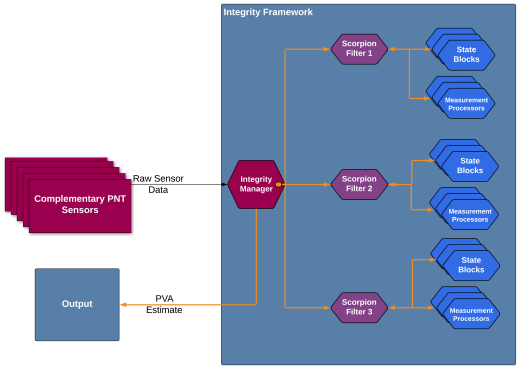

Fig. 4. An integrity framework using Scorpion's modular capability to build isolated filters to testing and evaluation of complementary PNT sensors.

F. Integrity Framework

While complementary PNT sensors are often used to augment GPS/INS with the goal of increasing the accuracy of the navigation solution, the addition of many new sensors opens the solution up to errors caused by sensor failure, modeling error, or other phenomena which affect these new sensors. It is therefore undesirable to incorporate arbitrary new sensors into a sensor fusion solution until the inputs have first been validated.

One of the benefits of using a modular architecture like Scorpion to build filters is that it allows for rapid development of filters with different integration strategies and sensor sets. While this capability is useful for building custom filters for a specific sensor platform, it can also be leveraged to build multi-model adaptive estimators (MMAE). We can use this capability to build a MMAE framework for testing the validity of new complementary PNT sensors added to the system.

Fig. 4 illustrates the configuration of a MMAE integrity monitor framework that uses the modularity of Scorpion to build isolated filters that test and evaluate complementary PNT sensors. The sensor data is intercepted by a integrity manager, which creates a bank of filters internally as needed. When new sensors are added to the system, the integrity manager can add additional filters to the bank to evaluate the statistical properties of a filter which incorporates the new sensor. Each filter has a separate isolated set of MeasurementProcessors and StateBlocks so that the filters do not interfere with each other.

The integrity manager can include arbitrary logic inside of it to decide when to allow new complementary sensors being tested in the MMAE filter bank to be added to the estimate that is produced the output. A modular integrity manager has been previously developed and integrated into Scorpion [23]. This manager makes the algorithm used to evaluate, correct, reject, and integrate new sensors into the various filter banks pluggable, and thus is a modular implementation of the integrity manager shown in Fig. 4. Please see [23] for more information on the details of the integrity framework implementation.

IV. Results

In this section we show how one can create and modify filters to test various configurations on a given problem set. In this example we compare how an EKF, UKF and Rao-Blackwellized PF behave when used to process various subsets of GPS position, altitude and angle-to-feature measurements to correct a free-running inertial. Actual experimental flight test sensor data is used to generate the results.

In the first example each filter is supplied with a 15-state Pinson-style StateBlock that models position, velocity, attitude and inertial sensor bias errors, and a MeasurementProcessor capable of ingesting angles to known ground features and generating position and attitude updates. All initial conditions were identical in each scenario. In the case of the particle filter, the tilt errors were configured as particle states and only 1000 samples were used.

In lines 2-4 of Listing 18, we select one of the filters to use. On line 7 we initialize our selected StateBlock, the first argument being a String label (`ps`) by which we can reference the states, and add it to the filter in line 8. Line 9 is the creation of the MeasurementProcessor, which is supplied with a label for itself (`fp`) and the label of the StateBlock(s) it will update. It is also then added to the filter.

At this point each module may be supplied with any additional information it may need. For example, StateBlocks can be supplied with initial state estimates and covariance, or a MeasurementProcessor may require information about how a sensor is installed relative to the vehicle. There are bespoke functions for some common operations (line 12), and catch-all giveStateBlockAuxData and giveMeasurementProcessorAuxData methods that allow arbitrary information to be passed to a StateBlock or MeasurementProcessor, which will contain logic on how to interpret that data. In all cases, information supplied to the filter is routed to the destination module using the label.

Once all elements have been initialized, the filter is ready to process data. As sensor data is received, it is packaged with a module label for the filter to ingest and passed through the appropriate function; filter updates will be passed to the 'update' function, time-varying module parameters such as linearization points through giveStateBlockAuxData. In this example we are receiving angles to ground features as sensor updates, and the StateBlock we supplied is an error-state (as opposed to whole-state) representation. The MeasurementProcessor must generate a model that maps the inertial error states to the whole valued sensor measurements; to do so it must have knowledge of the reference trajectory generated by the inertial at the time the sensor measurement was produced. The MeasurementProcessor therefore requires that the sensor measurement be paired with the appropriate reference trajectory point, as in line 16.

Fig. 5 shows a snapshot of the difference between the filter corrected position (error state plus inertial) along the local North axis and a reference solution provided by another system. After a period of free-running inertial during which

```
# Can freely swap between any of these
filter = SensorEKF(start_time)
# filter = SensorUKF(start_time)
# filter = SensorRBPF(start_time, 1000)

# Create modules and add to filter
block = PinsonBlock('ps', imuModel())
filter.addStateBlock(block)
proc = FeatureProcessor('fp', 'ps')
filter.addMeasurementProcessor(proc)
# Init with e.g., covariance, lever arms
filter.setStateBlockCovariance('ps', cov)

# For each measurement received...
filter.propagate(sensor_meas.time)
paired = PairedPva(sensor_meas, ins)
filter.update(Measurement('fp', paired))
```

Listing 18: Sample Code for Filter Setup

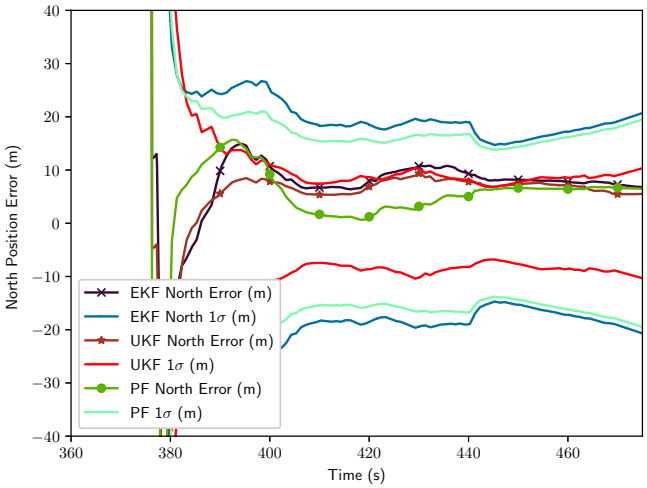

Fig. 5. Filter Comparisons, Angle Measurements Only.

no filter updates were supplied, at approximately 380 seconds into the data the filters begin receiving the angle updates.

As mentioned in the intro to this section, we are not limited to just angle measurements; this data set also contains other position data. Assume we wanted to modify the previous example to also incorporate the altitude measurements. We can do so with only two additional lines added to the filter initialization, as shown in Listing 19.

All that remains is to format the altitude sensor data into a Measurement tagged with the processor label `al` and pass to the update function. The results of this process are shown in

```
proc2 = AltitudeProcessor('al', 'ps')
filter.addMeasurementProcessor(proc2)
```

Listing 19: Adding an Altitude MeasurementProcessor

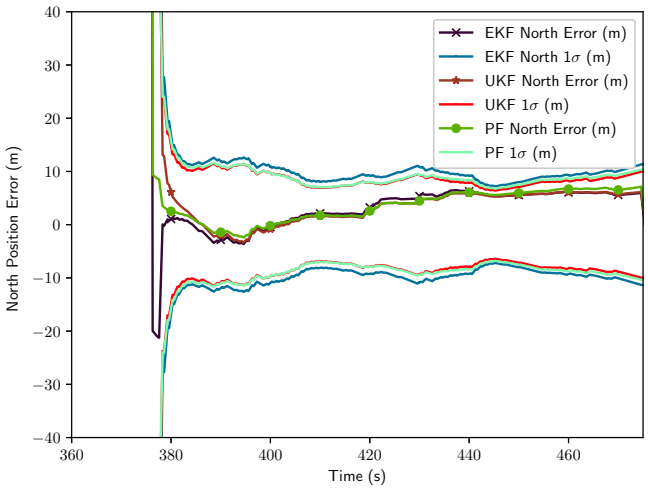

Fig. 6. Filter Comparisons, Angle and Altitude Measurements.

Fig. 6.

The point of these examples is not to show that any given setup outperforms another or is optimally tuned, but rather that by varying only three lines of code six different filter configurations were tested. Filter modification is not required to accommodate any modules, existing or new; all changes are confined to the 'controller' level.

V. Conclusion

In this paper, we developed a modular architecture for the integration of complementary PNT sensors. We showed that the architecture allows for the integration of advanced and classical filtering algorithms as well as a wide breadth of sensor integration techniques. We further discussed how the modular design allows for the development of high-integrity solutions, by using an MMAE integrity manager that builds modular filters to test and evaluate sensor models and errors. Finally, we built a software reference implementation of the architecture and demonstrated that it functioned properly on real-world experimental flight data.

References

[1] K. Fisher and J. F. Raquet, "Precision position, navigation, and timing without the global positioning system," *Air & Space Power Journal*, vol. 25, no. 2, pp. 24–33, 2011.

[2] M. Rabinowitz and J. Spilker J.J., "A new positioning system using television synchronization signals," *Broadcasting, IEEE Transactions on*, vol. 51, no. 1, pp. 51–61, 3 2005.

[3] G. Shippey, M. Jonsson, and J. N. B. Pihl, "Position Correction Using Echoes From a Navigation Fix for Synthetic Aperture Sonar Imaging," *Oceanic Engineering, IEEE Journal of*, vol. 34, no. 3, pp. 294–306, 7 2009.

[4] C. Toth, D. A. Grejner-Brzezinska, and Y.-J. Lee, "Terrain-based navigation: Trajectory recovery from LiDAR data," in *Position, Location and Navigation Symposium, 2008 IEEE/ION*, 5 2008, pp. 760–765.

[5] K. Kauffman, J. Raquet, Y. Morton, and D. Garmatyuk, "Real-time UWB-OFDM radar-based navigation in unknown Terrain," *IEEE Transactions on Aerospace and Electronic Systems*, 2013.

[6] M. J. Veth, R. K. Martin, and M. Pachter, "Anti-Temporal-Aliasing Constraints for Image-Based Feature Tracking Applications With and Without Inertial Aiding," *Vehicular Technology, IEEE Transactions on*, vol. 59, no. 8, pp. 3744–3756, 10 2010.

[7] A. Varshavsky, M. Y. Chen, E. de Lara, J. Froehlich, D. Haehnel, J. Hightower, A. LaMarca, F. Potter, T. Sohn, K. Tang, and I. Smith, "Are GSM Phones THE Solution for Localization?" in *Mobile Computing Systems and Applications, 2006. WMCSA '06. Proceedings. 7th IEEE Workshop on*, 4 2006, pp. 20–28.

[8] T. D. Hall, "Radiolocation using AM broadcast signals: The role of signal propagation irregularities," in *Position Location and Navigation Symposium, 2004. PLANS 2004*, 4 2004, pp. 752–761.

[9] P. S. Maybeck, *Stochastic models, estimation, and control, vol. 1*, 1979.

[10] S. Haykin, Ed., *KALMAN FILTERING AND NEURAL NETWORKS*. John Wiley & Sons, Inc, 2001.

[11] R. Brown and P. Hwang, *Introduction to Random Signals and Applied Kalman Filtering*, 3rd ed. Hoboken, NJ, USA: Wiley, 2002.

[12] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," *In Practice*, vol. 7, no. 1, pp. 1–16, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.6578&rep=rep1&type=pdf

[13] Z. H. E. Chen, "Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond," *Statistics*, vol. 182, no. 1, pp. 1–69, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.7415&rep=rep1&type=pdf

[14] S. J. Julier and J. K. Uhlmann, "Unscented Filtering and Nonlinear Estimation," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422, 2004.

[15] S. C. Patwardhan, S. Narasimhan, P. Jagadeesan, B. Gopaluni, and S. L. Shah, "Nonlinear Bayesian state estimation: A review of recent developments," *Control Engineering Practice*, vol. 20, no. 10, pp. 933–953, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.conengprac.2012.04.003

[16] F. Castanedo, "A review of data fusion techniques," *The Scientific World Journal*, vol. 2013, 2013.

[17] F. Auger, M. Hilairet, J. M. Guerrero, E. Monmasson, T. Orlowska-Kowalska, and S. Katsura, "Industrial applications of the kalman filter: A review," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 12, pp. 5458–5471, 2013.

[18] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2006.

[19] M. Stevens, "Bayes++ Bayesian Filtering," 2020. [Online]. Available: http://bayesclasses.sourceforge.net/Bayes++.html

[20] R. R. Labbe, *Kalman and Bayesian Filters in Python*. Online, 2018. [Online]. Available: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[21] D. Duckworth, "pykalman," 2012. [Online]. Available: https://pykalman.github.io/

[22] T. Schön, F. Gustafsson, and J. Nordlund, "Marginalized Particle Filters for Mixed Linear/Nonlinear State-space Models," Tech. Rep.

[23] J. D. Jurado, "AFIT Scholar AFIT Scholar Autonomous and Resilient Management of All-Source Sensors for Autonomous and Resilient Management of All-Source Sensors for Navigation Assurance Navigation Assurance Recommended Citation Recommended Citation," Tech. Rep. [Online]. Available: https://scholar.afit.edu/etd/2361