AAS 17-121

ON THE AUTOMATIC GENERATION OF RECURSIVE ATTITUDE DETERMINATION ALGORITHMS

Tucker McClure*

A method for reducing the development time of a recursive attitude determination system for a spacecraft by means of using software-generated algorithms is discussed. The capability of a software package to integrate the relevant spacecraft models and to automatically generate various recursive state estimation algorithms is demonstrated, enabling rapid iteration of the design from an early prototype to a detailed and mature algorithm that can meet accuracy and runtime requirements. This process is demonstrated for a small satellite, beginning with a generic sigma-point filter for the initial prototype and proceeding through to an extended Kalman filter and finally an efficient and stable square-root form. Simulated results for commercial-off-the-shelf sensors are provided.

INTRODUCTION

The design of an attitude determination system for a spacecraft drives many aspects of spacecraft development, including sensor selection, mechanical placement of the sensors, pointing bandwidth, electrical buses, and flight computer requirements. It is therefore desirable to have a representative attitude determination algorithm early in the program's life, allowing the team to simulate various sensor combinations against the performance requirements. However, designing attitude determination algorithms requires a substantial amount of work, including creating sensor models, deriving Jacobian matrices, creating a simulation, implementing the state estimation architecture, testing for implementation errors, and testing for proper statistical functionality. Completing this work early in the program can be difficult. Further, if the designer wishes to start simple and iterate towards the final algorithm, each iteration can be a lengthy process involving new theoretical work and code. Finally, there are many excellent and well-documented variations within each type of state estimation architecture, but these variations can change the implementation substantially, and coding them by hand is a time-consuming and error-prone task that not all programs' schedules can afford.

This paper investigates a process by which many of the above tasks can be simplified by using a software package to automatically integrate spacecraft equations of motion and sensor models into various state estimation algorithms. In this process, the work specific to the spacecraft domain (sensor models, Jacobians) still falls to the developer, but the software package will automate the implementation and testing of the state estimation algorithms, and will nearly eliminate the time required for studying algorithm variations. This allows the creation of an early initial prototype and much-reduced time to maturity.

To demonstrate the process, this paper considers the development of a classic and relatively simple filter that estimates attitude error and gyro bias error using a star tracker for measurements.

*Consultant & Developer, An Uncommon Lab, 1732 Aviation Blvd. #512, Redondo Beach, CA 90278.

First, a sigma-point filter will be developed, as this requires only two simple models and relatively little theoretical work. This will serve as the early initial attitude determination system, and will be exercised in simulation to determine what performance can be expected from a potential set of sensors. Next, an extended Kalman filter will be developed. This requires two additional models, but will run much faster than the sigma-point filter, and may be suited to implementation on the real flight computer. Finally, a square-root (UD) filter will be developed for its improved numerical properties, and it will be shown that this variation on the extended Kalman filter requires essentially no additional development time, despite that the code is almost completely different.*

MODELS

Star Tracker Model

The star tracker measures its orientation with respect to a star-fixed frame and returns this measurement as a quaternion, $\tilde{q}_k$ for a measurement produced on sample k. If the errors are small rotations given by the rotation vector $\nu_{s,k} \sim \mathcal{N}(0, R)$, and if q_k is the true attitude quaternion, then the measurement model is given in Eq. (1), where q_r is a function that converts a rotation vector to a quaternion as in Eq. (22) and $\otimes$ refers to quaternion composition, as in Eq. (23).†

$$\tilde{q}_k = q_r(\nu_{s,k}) \otimes q_k \tag{1}$$

The star tracker is modeled as having an error about its boresight and usually smaller cross-boresight errors. If the first axis is the boresight direction, σ_x is the standard deviation of the errors about the boresight, and σ_c is the standard deviation of the cross-boresight errors, then the measurement covariance matrix, R, is given in Eq. (2).

$$R = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_c^2 & 0 \\ 0 & 0 & \sigma_c^2 \end{bmatrix} \tag{2}$$

Gyro Model

The discrete gyro measurement is modeled as having a bias that undergoes a random walk. If b_{k-1} is the bias at some sample, then the bias on the next sample is given in Eq. (3), where $\nu_{b,k} \sim \mathcal{N}(0, \sigma_b^2 I)$ is a discrete random variable responsible for bias random walk and I is a 3-by-3 identity matrix.

$$b_k = b_{k-1} + \nu_{b,k} \tag{3}$$

This paper assumes a rate-integrating gyro is used, and so the measurement is the rotation vector, or equivalently the effective constant rotation rate over the sample. That is, if θ_k and $\hat{r}_k$ are the angle and unit axis of rotation over the sample, then the effective constant rotation rate is given by Eq. (4).

$$\omega_k = \frac{\theta_k}{\Delta t} \hat{r}_k \tag{4}$$

The measurement, $\tilde{\omega}_k$, is corrupted by bias and sample noise, $\nu_{g,k} \sim \mathcal{N}(0, \sigma_g^2 I)$, as:

$$\tilde{\omega}_k = \omega_k + b_{k-1} + \nu_{g,k} + \frac{1}{2}\nu_{b,k} \tag{5}$$

*The code used for this paper is available directly from the author.

†Rotation-related functions are given in the appendix.

where the final term accounts for the integration of the changing bias over the sample and is a linear approximation of the effect of the bias random walk. This paper uses this approximation for the sake of testing the filter in a discrete simulation.

For simplicity, the axes of the star tracker frame and gyro frame are assumed to be aligned with the desired body frame, and all produce their measurements with respect to the fixed stars.

FILTER PROCESS

The filters developed below will estimate the attitude quaternion and gyro bias over time, as in the classical Lefferts, Markley, and Shuster paper.[1] The following describes the process of propagating the estimated attitude quaternion and gyro bias (the state), running the filter with the new measurement, and correcting the estimates with the filter results. This process is common to all of the generated filters.

To begin, the following are presumed to be available: the previous sample's estimated quaternion, $\hat{q}_{k-1}$, the previous sample's estimated bias, $\hat{b}_{k-1}$, the previous sample's covariance, P_{k-1}, the current rotation rate measurement, $\tilde{\omega}_k$, and the star tracker measurement, $\tilde{q}_k$ (on samples when a new star tracker measurement is available).

First, the estimated rotation rate is calculated by subtracting the estimated bias from the measured rotation rate, as in Eq. (6).

$$\hat{\omega}_k = \tilde{\omega}_k - \hat{b}_{k-1} \tag{6}$$

Second, the estimated rotation rate is used to propagate the estimated quaternion to the current time, using Eq. (7). This produces the predicted attitude quaternion, $\hat{q}_k^-$, where the minus sign denotes "prior to correction". The bias is not expected to change, so the propagated bias, $\hat{b}_k^-$, is in fact simply $\hat{b}_{k-1}$.[1–3]

$$\hat{q}_k^- = q_r\left(\Delta t \hat{\omega}_k\right) \otimes \hat{q}_{k-1} \tag{7}$$

Third, the star tracker measurement is expressed not as a quaternion, but as a "pseudomeasurement" — the difference between the measurement and the predicted attitude, expressed as generalized Rodrigues parameters (GRPs). This is shown in Eq. (8), where p_q denotes a function that converts a quaternion to GRPs as in Eq. (25) and the $^{-1}$ denotes a quaternion inverse (opposite rotation).

$$z_k = p_q\left(\tilde{q}_k \otimes \hat{q}_k^{-1}\right) \tag{8}$$

Fourth, the filter is executed, and it returns the attitude correction, $\delta\hat{p}_k$, the bias correction, $\delta\hat{b}_k$, and the updated estimate covariance, P_k. This step will be referred to as the "filter step", and the two correction terms, taken together as $\delta\hat{x}_k = \begin{bmatrix}\delta\hat{p}_k^T & \delta\hat{b}_k^T\end{bmatrix}^T$, will be referred to as the "error state".

Fifth, the propagated quaternion and bias are corrected via Eq. (9) and Eq. (10), where q_p is a function that converts GRPs to a quaternion as in Eq. (26). This gives the updated quaternion and bias, and it completes the filter process for the current sample.

$$\hat{q}_k = q_p\left(\delta\hat{p}_k\right) \otimes \hat{q}_k^- \tag{9}$$

$$\hat{b}_k = \delta\hat{b}_k + \hat{b}_{k-1} \tag{10}$$

On samples for which no star tracker measurement is available, the quaternion, bias, and covariance will still be propagated by the filter, but of course there will be no corrections ($\delta\hat{p}_k = 0$ and $\delta\hat{b}_k = 0$).

A SOFTWARE PACKAGE FOR GENERATING STATE ESTIMATION ALGORITHMS

The filter step in the above process can be created automatically using a software package for generating state estimation algorithms. The package examined here is called *kf from An Uncommon Lab.*† *kf runs in MATLAB and, given models of the user's problem, can create efficient state estimation algorithms using a variety of architectures and options.

The *kf engine creates filters by representing numerous symbolic snippets of code with associated assumptions. For instance, one snippet of code may calculate the Kalman gain for a general measurement covariance matrix, but another snippet of code may calculate the Kalman gain assuming that the measurement covariance matrix is diagonal. The latter is much faster when the assumption holds. The engine has a very large collection of possible snippets, and these can be pieced together to form different filters. The snippets also have comments that explain what they do, and these comments become part of the generated code.

When the user specifies the desired outputs (*e.g.*, the corrected state, corrected covariance, and innovation covariance), the engine can go about finding snippets that provide the desired outputs and satisfy the user's stated assumptions. Each selected snippet will then be examined to see what its inputs are. For instance, if the snippet $K = P_{xy}P_{yy}^{-1}$ is selected, then the algorithm knows that it must now find snippets that provide P_{xy} and P_{yy}. It recursively searches through the available snippets, always looking for the fastest snippets that can provide each variable it needs in order to continue.

Snippets are also created for the user's functions and data, such as the propagation and measurement functions, functions to calculate the Jacobians, and the process and measurement noise covariance matrices. With these, it can generate code that integrates with the user's models.

Some values cannot be provided by a snippet. For instance, there can be no snippet providing the measurement vector; that must come from the user. When the algorithm finds a variable that it can't provide, it makes the variable a top-level input to the algorithm.

Once all of the desired outputs can be derived from top-level inputs, the engine proceeds to find a way to implement the algorithm efficiently as code. For instance, some snippets may only need to be executed in certain cases, and so can go inside part of an if-else block. To begin, the algorithm declares that only the top-level inputs are available, and it finds the first snippet that can be executed. This becomes the first line of code in the generated filter. It then adds that snippet's outputs to its list of available variables and searches for the next snippet that can be executed, which becomes the next line of code. In this way, a variety of state estimation algorithms are automatically pieced together. Nowhere, for instance, is the outline of a standard extended Kalman filter specified in *kf, but when assumptions leading to an EKF are selected, the standard EKF equations are the result. Further, the code only contains what is necessary for the user's particular problem, allowing the generated algorithms to be clean and relatively easy to read.

Additionally, the engine can use the propagation and measurement functions and noise covariance

*`http://www.anuncommonlab.com/starkf/`

†For transparency, it should be noted that *kf is the author's own work.

matrices to create a second piece of code: a truth simulation that exercises the filter. For some number of samples, this simulation will take random draws for the process and measurement noise according to their covariance matrices, will update a true state and measurement with these, and will run the filter on the resulting measurement. Further, a Monte-Carlo wrapper will also be generated. In this way, the filter can be automatically checked for consistency with a simulation in which the filter's propagation and measurement functions are exact and the random numbers are in fact white and Gaussian. This is useful for debugging immediately, before the code is integrated into the real spacecraft simulation where errors might be harder to find.

It is this engine that makes the iterative development process for an attitude determination algorithm much easier, as it (1) enables one to create a filter early in the program's life with no wasted effort, (2) makes it trivial to try variations on a filter over time, and (3) makes it easier to test the filters precisely. This will be investigated below.

Aside from *kf, there is one other software package known to the author for generating custom state estimation algorithms, called AUTOFILTER, developed at NASA Ames. This software has many of the same motivations as *kf, but interestingly works in a much different manner. AUTOFILTER uses a template to define the desired filter, and the user provides the appropriate propagation and measurement functions as symbolic math. AUTOFILTER operates on the symbolic math, producing Jacobian functions automatically, and filling in the template with details from the user's functions and inputs. It can generate the filters as MATLAB and C. Though interesting, AUTOFILTER was not used in this work because it is not actively maintained and because many propagation and measurement functions are not properly described by the symbolic math that it requires. However, it should be noted that AUTOFILTER was used on an orbit problem and executed on an embedded computer — an impressive feat.[4]

INITIAL SIGMA-POINT FILTER

Motivation

For the initial prototype filter, a sigma-point filter (also known as an unscented Kalman filter or UKF) will be constructed. The reason for choosing a sigma-point filter is that it does not require analytical Jacobians, and so can often be implemented more quickly than an extended Kalman filter. It also requires fewer assumptions about the nature of the propagation and observation functions, and so works more generally than an extended Kalman filter. This paper will for the most part follow the work of Crassidis and Markley in Reference 3.

Sigma-point filters work by creating a set of hypothetical states ("sigma points") surrounding the current estimate. Each of these states is propagated, and the covariance of the propagated set is calculated, providing an accurate assessment of the new covariance. Predicted observations for each sigma point are also calculated, and the covariance of the predicted observation can be determined similarly. These two matrices are used to form the Kalman gain. Then, the expected value of the propagated state and its covariance are corrected. This paper will not dwell on UKF theory, but will instead describe the propagation function and observation function for each sigma point, as these are the only parts required to build the filter.

Propagation of the Sigma Point

The hypothetical sigma point i at sample $k-1$ will be defined by four quantities: the attitude error, $\delta p_{i,k-1}$, the bias error, $\delta b_{i,k-1}$, the gyro sample noise, $\nu_{g,i,k}$, and the gyro bias random walk

noise, $\nu_{b,i,k}$. For this hypothetical state, the bias is constructed from the nominal bias for the sample, plus the sigma point's bias error.

$$b_{i,k-1} = \delta b_{i,k-1} + \hat{b}_{k-1} \tag{11}$$

The estimated quaternion is constructed from the sigma point's attitude error combined with the nominal attitude quaternion.

$$q_{i,k-1} = q_p\left(\delta p_{i,k-1}\right) \otimes \hat{q}_{k-1} \tag{12}$$

The rotation rate is then constructed by removing the hypothetical bias and noise from the measurement.

$$\omega_{i,k} = \tilde{\omega}_k - \hat{b}_{i,k-1} - \nu_{g,i,k} - \frac{1}{2}\nu_{b,i,k} \tag{13}$$

With the estimated rotation rate in place, the propagated quaternion for this sigma point is determined similarly to Eq. (7).

$$q_{i,k} = q_r\left(\Delta t \omega_{i,k}\right) \otimes \hat{q}_{i,k-1} \tag{14}$$

The propagated attitude error, δp_k^-, is then calculated as the difference between the propagated quaternion for the sigma point and the nominally propagated quaternion, $\hat{q}_k^-$.

$$\delta p_{i,k}^- = p_q\left(q_{i,k} \otimes \left(\hat{q}_k^-\right)^{-1}\right) \tag{15}$$

The propagated bias error, δb_k^-, is of course different only by the bias random walk noise.

$$\delta b_{i,k}^- = \delta b_{i,k-1} + \nu_{b,i,k} \tag{16}$$

This completes the propagation of $\delta p_{i,k-1}$ to $\delta p_{i,k}^-$ and $\delta b_{i,k-1}$ to $\delta b_{i,k}^-$ for this sigma point.

Observation of the Sigma Point

For each sigma point, the innovation vector, y_i (the error between the measured attitude and predicted attitude), must be determined, and the errors are taken to be GRPs. This can be expressed directly in terms of the pseudomeasurement, z_k, and the sigma point's attitude error, $\delta p_{i,k}$, as in Eq. (17), where $\otimes$ represents GRP composition, given in Eq. (27). Note that changing the sign of GRPs reverses the rotation.

$$y_i = z_k \otimes -\delta p_{i,k} \tag{17}$$

Process Noise for the UKF

In the UKF, the process noise consists of the gyro sample noise and bias random walk noise, and hence the process noise covariance matrix is given in Eq. (18), where I is a 3-by-3 identity matrix and the zeros have the appropriate dimensions. In the generated filter, $\nu_{g,i,k}$ and $\nu_{b,i,k}$ will be constructed from this matrix.

$$Q = \begin{bmatrix} \sigma_g^2 I & 0 \\ 0 & \sigma_b^2 I \end{bmatrix} \tag{18}$$

Implementation

The first step towards creating the sigma-point filter was to implement the propagation and observation functions in MATLAB and define the process noise and measurement noise covariance matrices. Then, the sigma-point filter could be created using *kf. To begin, the "sigma-point" filter type was selected, which determines which set of snippets is used to build the filter.

The names of the propagation and observation functions were set in the options, and the process noise and measurement noise covariance matrices (constants for this example) were specified as well. A few configuration options were necessary, such as specifying the inputs to these functions.

Because the observation function represented by Eq. (17) in fact returns the *innovation vector* for the sigma point instead of the predicted observation, an option was selected to this effect.

The measurement noise was declared to be "additive" to the innovation vector, which reduces the number of sigma points necessary for the filter and hence allows it to run faster.[3]

Because *kf can generate an example simulation in order to test the generated filter, a true measurement function was provided.

Finally, some output options were necessary, such as the name for the generated filter and a location for the generated code. Using these, *kf created the custom algorithm, basic simulation, and Monte-Carlo wrapper.

The generated simulation and Monte-Carlo test were executed immediately to verify that the filter would run correctly. This ideal simulation provides a fast way to find implementation errors, and in fact, revealed a mistake that the author had made in his first implementation. Once the error was corrected, these two tests ran with theoretically appropriate results, suggesting a valid implementation.

Simulation Results

The UKF was integrated into a spacecraft simulation for detailed testing. In this simulation, the ideal spacecraft dynamics are implemented, with an ideal rate-limited controller driving the spacecraft to various orientations. The gyro was modeled after the Sensonor STIM 210 MEMS gyro unit, running at 20Hz, and the star tracker was modeled as having 200 arcsec boresight errors and 100 arcsec cross-boresight errors, with a sample time of 1 sec. The simulation was executed for 60 sec, and Figure 1 shows the estimation errors alongside the 95% confidence bounds for the sensor errors.

The simulation was then run 1000 times as part of a Monte-Carlo test. In the final 100 samples of the flights, the root-mean-square total attitude error was 272 arcsec, which seems acceptable (for reference, when no bias estimation was performed, the error was 1349 arcsec). A statistical analysis of the results was also conducted, which included the evaluation of the normalized estimation error squared (NEES), the normalized mean estimation error (NMEE), the normalized innovation squared (NIS), and the total autocorrelation of the residuals (TAC). These tests determine how well the estimation errors correspond with the estimate covariance matrix and whether the residuals are reasonably white. The tests are very well documented in Reference 5, pages 232-236. In brief, the NEES test and NMEE test determine whether the estimation errors are consistent with the filter's covariance matrix over time. The NIS test is similar, but applies to the measurement residuals and innovation covariance, and the TAC test determines if the residuals are reasonably white. The results

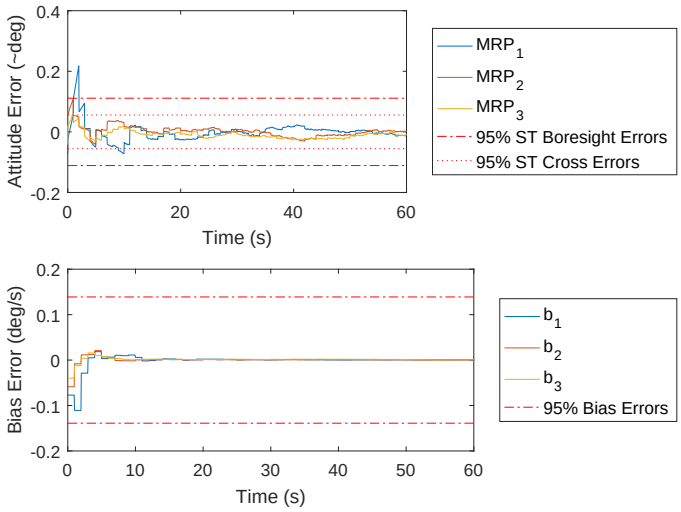

Figure 1. Attitude and Bias Errors

of the NEES test are plotted in Figure 2, and all of the results are summarized in Table 1. By all tests, the filter appeared reasonably consistent with uncorrelated residuals.

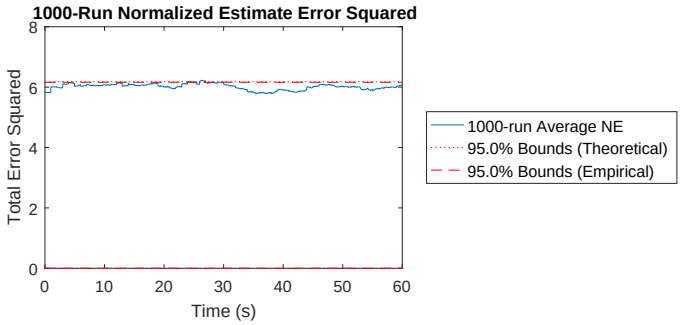

Figure 2. 1000-Run Average Normalized Estimation Error Squared for UKF

This completes the development of the UKF. The only parts to be developed by hand were the propagation and observation functions and process noise and measurement noise covariance matrices. This is a substantial reduction in the amount of time necessary to create and test a UKF by hand, and the generated UKF is nonetheless customized directly to the spacecraft attitude problem. The generated UKF could be used in the simulation for sensor trade studies or to give a realistic basis for attitude control development.

EXTENDED KALMAN FILTER

Motivation

The UKF may serve well in initial studies, but for the sensors under consideration, it requires far more runtime than is necessary. An extended Kalman filter (EKF) with sequential scalar updates will be much faster, and is expected to have nearly the same performance. This filter will now be developed, approximately following the classical Lefferts, Markley, and Shuster outline.[1] However,

the filter implemented here uses the GRPs as the error state for consistency with the UKF above.

Two new things will be necessary to implement the EKF: the Jacobian of the propagation function and the effective process noise covariance.

Jacobian of the Propagation Function

The Jacobian of the propagation function, $F(\omega)$, will not be derived here, but the following is a first-order discretization of the Jacobian provided in Reference 6, page 258, where $^\times$ is described in Eq. (24). It is a function of the true rotation rate (or, for the filter, of the estimated rotation rate).

$$F(\omega) = \begin{bmatrix} I - \Delta t \omega^\times & -\Delta t I \\ 0 & I \end{bmatrix} \tag{19}$$

To check this Jacobian, the UKF's propagation function was again useful. The Jacobian of the propagation function at a random state was approximated using the finite-difference method with the UKF's propagation function, and this result was compared against the analytical Jacobian for that state, above. This verified the analytical Jacobian and was a convenient tertiary benefit of having started with the UKF.

In fact, this Jacobian was very easy, and it is appropriate to think that the UKF was unnecessary — that the EKF could have been implemented just as quickly. For this particularly simple filter, this is true. However, many filters have additional states or other subtleties, and the calculation of the Jacobian is often much more difficult than this. In such cases, the benefits would be more clearly felt. For instance, the author has used this process on more complex filters for both spacecraft and UAVs, where the Jacobians required much more development, and the ability to start quickly with a UKF, and then transition to an EKF and mature over time was decidedly helpful on a tight schedule. The ability to use the UKF as a truth test for the EKF and its Jacobians was additionally helpful.

Effective Process Noise

In the UKF, the process noise consisted of the bias random walk noise and gyro sample noise, and these affected the state propagation in nonlinear ways. For the EKF, the process noise is expected to add directly to the state propagation, as in $\delta x_k = f(\delta x_{k-1}) + \nu_{\text{eff}}$ where $\nu_{\text{eff}} \sim \mathcal{N}(0, Q_{\text{eff}})$. This will be called the effective process noise, and its covariance is given in Eq. (18). *

$$Q_{\text{eff}} = \begin{bmatrix} \left(\sigma_g^2 + \frac{1}{4}\sigma_b^2\right) \Delta t^2 I & -\frac{1}{2}\sigma_b^2 \Delta t I \\ -\frac{1}{2}\sigma_b^2 \Delta t I & \sigma_b^2 I \end{bmatrix} \tag{20}$$

Like the Jacobian above, the effective process noise covariance matrix was also tested by comparing its results with results obtained using the finite-difference method on the UKF propagation function.

Implementation

The implementation of the EKF in *kf involved a few steps. First, the "linear and extended" filter type was selected, which selects the appropriate snippets.

*Note that the factor of $\frac{1}{4}$ in Eq. (20) is given as $\frac{1}{3}$ in Reference 6, page 260, which is more accurate for gyro models driven by continuous white noise. However, the simulation deals with a discrete approximation, and for consistency, $\frac{1}{4}$ is the appropriate factor. The difference is negligible in practice, as σ_g^2 generally dwarfs σ_b^2 by many orders of magnitude.

Note that no propagation function is necessary, as the quaternion and bias are propagated outside of the generated filter code. The propagated attitude error and bias error, $\delta\hat{p}_k^-$ and $\delta\hat{b}_k^-$, can therefore be provided directly as inputs (and will actually always be zeros). For this reason, the propagated state was declared to be a direct input to the generated filter function.

Next, the function to calculate the propagation Jacobian, $F(\omega)$, was specified.

Recall that the pseudomeasurement, z_k, consists of the GRPs for the measured attitude with respect to the predicted attitude. The predicted measurement, $\hat{z}_k$, is therefore simply $\delta\hat{p}_k^-$, or the first three indices of the 6-by-1 error state vector, δx_k^-. In such a case, the observation function is clearly $\hat{z}_k = H\delta\hat{x}_k^-$ where $H = \begin{bmatrix} I & 0 \\ & _{3\times3} \end{bmatrix}$ is the observation Jacobian. There is no need to specify this function and Jacobian to *kf, and there is no need for it to literally multiply by the identity matrix internally. Instead, the software provides an option to specify that the observation is a subset of the state, and the generated filter will handle its internal operations efficiently. This option was selected.

The effective process noise and measurement noise covariance matrices were provided to the engine as constants. Further, the measurement noise covariance matrix was declared to be diagonal, enabling sequential scalar updates. This prevents the filter from needing to calculate a matrix inversion, instead performing three trivial scalar inversions, which is much faster in implementation.[1]

With these options set, the EKF code, basic simulation, and Monte-Carlo test were generated. These were executed to verify the implementation, and all appeared to be correct.

Simulation Results

The EKF was then incorporated into the spacecraft simulation, acting as a drop-in replacement for the UKF. The simulation and 1000-run Monte-Carlo test were executed, using the same random number draws as for the UKF, and there were no meaningful differences in performance. A few figures are compared in Table 1.

Table 1. Results for 1000-Run Monte-Carlo Test

Test	UKF	EKF	Units
RMS Total Attitude Error (final 100 samples)	272	272	arcsec
NEES	97.5	97.6	% of errors inside theoretical 95% bounds
NMEE	97.1	97.1	% of errors inside theoretical 95% bounds
NIS	88.5	88.5	% of errors inside theoretical 95% bounds
TAC	95.0	95.6	% of errors inside theoretical 95% bounds

UD FILTER

Extended Kalman filters can have a problem with the covariance matrix losing symmetry or even developing a negative eigenvalue over time due to roundoff errors. This can be mitigated in a number of ways, but an elegant method is to operate on the U and D factors of the covariance matrix instead of operating on the covariance matrix itself. In this method, if P is the covariance matrix, U and D are maintained so that $UDU^T = P$, where U is upper unitriangular and D is diagonal. This provides no chance for a loss of symmetry and effectively provides a greater word length for the underlying data.[7] The theory of UD filters will not be discussed here, except to say that they require substantially different implementations, despite producing theoretically identical results to the EKF

with very nearly the same runtime. Because it takes time to implement, it is difficult to "try" a UD filter on a problem, and many practitioners therefore avoid them. However, since *kf can generate a UD, and since it requires no additional information from the developer, there was no reason not to try it as part of the study.

The process of implementing the UD filter using *kf was very simple; the EKF configuration was modified to change the uncertainty type from "Covariance Matrix" to "UD Factors". Nothing else was required. The code was regenerated and incorporated into the simulation. Running the Monte-Carlo test showed identical results to the EKF, so the results need not be reproduced here. The UD filter could now be selected as the desired algorithm for the spacecraft attitude determination system, and this completes the investigation.

CONCLUSIONS

A process for developing a filter by starting with an easy-to-implement form and continuing towards a more mature form intended for flight was investigated. Throughout, the spacecraft-specific models were developed by hand, as usual, but the core filter development was automated with a software package that generated code fit specifically to the spacecraft models, reducing the time necessary to develop these filters and test their implementations. For the UKF, two simple models were developed, and the filter could then be generated. For the EKF, a Jacobian and an effective process noise covariance matrix were developed, and this was all that was necessary. Testing the UD variation on the EKF required merely changing an option in the software package. Similar options could be tested quickly, and uncommon combinations of filtering options could be selected to generate new filters, without requiring that the user derive a new form by hand.

Naturally, a developer should not rely on a software product to generate code without an understanding of the generated algorithm. However, it can be far easier to learn about a particular filter type when one has a working prototype to examine. Further, even for developers with a deep understanding of the possible options and years of experience, coding each variation of a filter by hand takes a substantial amount of time and is error-prone. The use of a software product eliminates these problems and is therefore expected to provide benefits to both novices and experienced developers of attitude determination systems.

ACKNOWLEDGMENTS

The author wishes to thank Dr. Alexander Vandenberg-Rodes of the University of California, Irvine, for help in clarifying the ideas presented in this paper, for detailed feedback regarding *kf, and for drawing the author's attention to AUTOFILTER. The author also wishes to thank Dr. Johann Schumann of NASA Ames for discussions about AUTOFILTER.

REFERENCES

[1] E. J. Lefferts, F. L. Markley, and M. D. Shuster, "Kalman Filtering for Spacecraft Attitude Estimation," *Journal of Guidance*, Vol. 5, No. 5, 1982, pp. 417–429.

[2] M. D. Shuster, "A Survey of Attitude Representations," *Journal of Astronautical Sciences*, Vol. 41, No. 4, 1993, pp. 439–517.

[3] J. L. Crassidis and F. L. Markley, "Unscented Filtering for Spacecraft Attitude Estimation," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 4, 2003, pp. 536–542.

[4] J. Whittle and J. Schumann, "Automating the Implementation of Kalman Filter Algorithms," *ACM Transactions of Mathematical Software*, Vol. 30, No. 4, 2004, pp. 434–453.

[5] Y. Bar-Shalom, X. R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation: Theory, Algorithms, and Software*. New York, New York: John Wiley & Sons, Inc., 2001.
[6] F. L. Markley and J. L. Crassidis, *Fundamentals of Spacecraft Attitude Determination and Control*. New York, New York: Springer, 2014.
[7] B. J. Bierman, *Factorization Methods for Discrete Sequential Estimation*. Mineola, New York: Dover Publications, Inc., 1977.

APPENDIX: CONVERSIONS BETWEEN ROTATIONS

For a rotation of θ about unit axis $\hat{r}$, the corresponding rotation quaternion is given by Eq. (21). Note that the scalar is last in this formulation.

$$q_{\theta,r}(\theta, \hat{r}) = \begin{bmatrix} \sin\left(\frac{\theta}{2}\right)\hat{r} \\ \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{21}$$

The conversion from a rotation vector, r, to a quaternion follows naturally for $\|r\| > 0$.

$$q_r(r) = q_{\theta,r}(\|r\|, \frac{1}{\|r\|}r) \tag{22}$$

Quaternion composition is represented by $\otimes$ such that, if q_β is the rotation of frame C with respect to frame B, and q_α is the rotation of B with respect to A, then $q_\beta \otimes q_\alpha$ gives the rotation of C with respect to A, as:

$$q_\beta \otimes q_\alpha = \begin{bmatrix} -q_{\beta,1:3}^{\times} & q_{\beta,1:3} \\ -q_{\beta,1:3}^{T} & q_{\beta,4} \end{bmatrix} q_\alpha \tag{23}$$

where $q_{\beta,1:3}$ refers to the vector part of the quaternion, $q_{\beta,4}$ refers to the scalar, and $v^\times$ refers to a skew-symmetric matrix formed from v as below.

$$v^\times = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} \tag{24}$$

The conversion from a quaternion to corresponding generalized Rodrigues parameters (GRPs) is:

$$p_q(q) = \frac{f}{a + q_4} q_{1:3} \tag{25}$$

For the Gibbs vector, $a = 0$ and $f = 1$. For the traditional modified Rodrigues parameters, $a = 1$ and $f = 1$. This paper uses $a = 1$ and $f = 4$, for which the GRPs are approximately equal to the rotation vector for small angles.[3]

When $a = 1$, GRPs can be converted to a quaternion as in Eq. (26).

$$q_p(p) = \frac{1}{f^2 + \|p\|^2} \begin{bmatrix} 2fp \\ f^2 - \|p\|^2 \end{bmatrix} \tag{26}$$

The GRPs compose in the same manner as quaternions, and, when $a = 1$, their composition function is given by Eq. (27).

$$p_\beta \otimes p_\alpha = \frac{\left(f^2 - \|p_\alpha\|^2\right) p_\beta + \left(f^2 - \|p_\beta\|^2\right) p_\alpha - 2f p_\beta \times p_\alpha}{f^2 + f^{-2}\|p_\beta\|^2\|p_\alpha\|^2 - 2p_\beta \cdot p_\alpha} \tag{27}$$