Numerical Integration of Attitude Filter Equations

Author: Katrina Ashton

Supervisor: Jochen Trumpf

Course: ENGN2706 Engineering Research and Development Project (Methods)

Submission: 29 May 2015

Acknowledgements

I would like to thank Jochen Trumpf for his guidance and help. I would also like to thank Mohammad Zamani and Conor Horgan, whose work this project is based on.

Abstract

This report compares various numerical implementations of the Geometric Approximate Minimum-Energy filter (GAME filter) and the Multiplicative Extended Kalman Filter (MEKF). The numerical integration methods considered are Euler's method and modified versions of Choi's method and Möbius schemes for Riccati Differential equations. The implementations are evaluated in terms of resulting observer error, computational time and robustness. There is no clear frontrunner in all three criteria, however the GAME filter implemented with the modified Möbius method shows good performance overall. The detailed simulation results in this report provide useful information for the selection of a filtering algorithm and its numerical implementation based on application specific requirements.

Contents

1 Glossary and Notation

1.1 Abbreviations

ARE: Algebraic Riccati Equation

ASP: Automated Synthesis Program

EKF: Extended Kalman Filter

GAME: Geometric Approximate Minimum-Energy

MEKF: Multiplicative Extended Kalman Filter

ODE: Ordinary Differential Equation

RDE: Riccati Differential Equation

UAV: Unmanned Aerial Vehicle

UKF: Unscented Kalman Filter

1.2 Notation

The lower index operator, written as $(.)_\times : \mathbb{R}^3 \to \mathrm{so}(3)$, denotes the skew-symmetric matrix

$$\Omega_\times = \begin{bmatrix} 0 & -\Omega_3 & \Omega_2 \\ \Omega_3 & 0 & -\Omega_1 \\ -\Omega_2 & \Omega_1 & 0 \end{bmatrix}$$

The symmetric projector, written as $\mathbb{P}_S$, is defined by

$$\mathbb{P}_S(M) := \frac{1}{2}(M + M^T)$$

for a matrix $M^{n\times n}$. Here, $(.)^T$ denotes the matrix transpose.

A Gaussian distribution, with mean μ and standard deviation σ, is written as $\mathcal{N} \sim (\mu, \sigma^2)$.

2 Introduction

Attitude is the orientation of an object in 3 dimensional space. Most flying vehicles utilise automatic or semi-automatic navigation. Knowing the attitude of the vehicle is important for these navigation systems to function properly. Therefore, attitude estimation algorithms are deployed in aeroplanes, unmanned aerial vehicles (UAVs) and spacecraft.

Attitude estimation generally occurs by comparing the orientation of a body-fixed frame attached to the object to that of a fixed inertial reference frame. Measurements are taken of the direction of vectors such as gravity and the Earth's magnetic field with respect to the body-fixed frame. Because the directions of these vectors with respect to the inertial frame are known, this allows the two frames to be compared. It is well known that two linearly independent direction measurements uniquely determine 3D attitude [20]. These direction measurements are not, however, perfectly accurate. Attitude filtering systems compare measurements to the predictions of a dynamic model in order to produce accurate attitude estimates. These attitude filters often give rise to ordinary differential equations (ODEs) which must be solved at each time step when implementing the filter.

These ODEs cannot be solved analytically, so numerical integration must be used. Numerical integration uses the value of the derivative at a certain

time to approximate the solution of the ODE after some small time step.

The aim of this project is to compare different numerical implementations of two attitude filters: the Multiplicative Extended Kalman Filter (MEKF) and the Geometric Approximate Minimum-Energy (GAME) filter. The focus is on finding numerical integration methods that could be used to robustly implement the GAME filter.

3 Literature Review

3.1 Attitude Filters

Many attitude filters are based on the Kalman Filter. The Kalman Filter is only usable for linear systems [13]. However, it was adapted for use in non-linear systems, such as attitude dynamics. The Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) [12, 25] are examples of such adaptations. Extended Kalman filters are based on local linearizations of the system dynamics. Whereas Unscented Kalman filters compute the propagation of state uncertainty using samples of the statistical distribution at special locations, the so-called sigma points.

The Multiplicative Extended Kalman Filter (MEKF) is the industry standard for attitude estimation [28]. It parametrizes attitude using a quaternion and uses a three-component error vector [18]. In contrast, the Geometric Approximate Minimum-Energy (GAME) filter considers attitude estimation directly on the Lie group of special orthogonal matrices SO(3) [27]. This

allows for more accurate estimations of the attitude at each time step, as it removes the errors introduced by transforming the system using only first order instead of second order geometry. Also, this allows the filter derivation to be done using rotation matrices directly, thereby avoiding any potential issues with non-unique parametrizations and singularities.

Preliminary comparisons with other filtering methods show that the GAME filter outperforms the MEKF, the unscented quaternion estimator, the right-invariant extended Kalman filter and the nonlinear constant gain attitude observer in Monte-Carlo simulations [27]. Some further testing compares the MEKF and GAME filter in more detail [11]. However, this testing uses Euler methods for the numerical integration of the gain equations. Euler methods are not used in practice for attitude estimation, as they are not very accurate and can lead to results that do not match the geometry of any possible solution. Therefore, it is not clear if the GAME filter would outperform current filters in real-world applications.

3.2 Numerical Integration of Riccati Differential Equations

The gain equations of the MEKF and the GAME filter are both generalised Riccati differential equations (RDEs). The gain equation of the GAME filter can be written as [27]

$$\dot{P} = Q + \frac{1}{2}P(2u - Pl)_\times - \frac{1}{2}(2u - Pl)_\times P + P(A - S)P,$$

where $P \in \mathrm{Sym}(3)$ is a symmetric 3×3 matrix, $(.)_\times$ is the lower index operator, which yields a skewed-symmetric 3×3 matrix, and the remaining

matrices and vectors have the required formats.

This is similar to RDEs of the form

$$\dot{X} = Q + XA + BX - XRX \tag{1}$$

where X is an $m \times n$ matrix and Q, A, B and R are $m \times n$, $n \times n$, $m \times m$ and $n \times m$ time-varying matrices, respectively. Equations of this form have been widely studied [1, 3, 22].

Many ODEs, such as the ones arising from the GAME filter, cannot be solved analytically. Using numerical integration, however, the solution can be approximated to the desired degree of accuracy. Numerical integration takes an ODE and the current value of the unknown variable and outputs an approximation of what the unknown variable will be after some small time step.

The most simple method of numerical integration for RDEs is unrolling the RDE into a system of vector ODEs, a technique that is not unique to RDEs. For the RDE (1), the resulting system is of size mn, and must be solved at each time step [4]. Thus this is not an efficient method of solving large RDEs.

Many of the specialised methods of numerical integration for matrix RDEs involve transforming the RDEs into systems of linear first-order ODEs [6, 15, 17, 19, 23, 26]. Most of these methods are based on the following theorem, which is proven in [22].

Theorem 1:

Consider a RDE of the form (1) with initial condition $X(t_0) = V_0 U_0^{-1}$, $t_0 \leq t \leq T$. Then $X(t) = V(t)U^{-1}(t)$ is a solution of the RDE if and only if $\begin{bmatrix} U(t) \\ V(t) \end{bmatrix}$ is a solution of $\begin{bmatrix} \dot{M}(t) \\ \dot{N}(t) \end{bmatrix} = \begin{bmatrix} -A(t) & R(t) \\ Q(t) & B(t) \end{bmatrix} \begin{bmatrix} M(t) \\ N(t) \end{bmatrix}$.

Rewriting X in this way allows the RDE to be solved at any specific time if the fundamental matrix of $\begin{bmatrix} U(t) \\ V(t) \end{bmatrix}$ is known. That is, the matrix Φ for which $\begin{bmatrix} U(t) \\ V(t) \end{bmatrix} = \Phi(t, s) \begin{bmatrix} U(s) \\ V(s) \end{bmatrix}$ for all times t and s.

For time-invariant systems, the fundamental matrix can be found using matrix exponentiation. The Davison-Maki method [6] uses this fact combined with Theorem 1. The matrix exponential is updated at each time step and used to solve the linear system associated with the RDE. However, this process involves the inversion of matrices whose columns become more linearly dependent as the time increases. Therefore, the process can become unstable over time.

The automatic synthesis program (ASP) [14] fixes this problem by rewriting the formula used by Davison-Maki to depend on t_k and t_{k+1} instead of t and t_0. When the ASP is used with small time steps, it only involves the inversion of matrices that approach the identity matrix as time increases [4]. Also, the matrix exponential no longer needs to be updated. However, the size of time

step necessary may become prohibitively small [26].

Vaughan [26] avoids this issue by changing the equations used by the ASP into a form that expresses the solution in terms of constant matrices and terms containing only exponentials with negative real parts. This prevents essential information from being coded in numbers with very different magnitudes, which is what causes the issue with the ASP. However, this method is only applicable to the time-invariant case and therefore cannot be used on the gain equation of the GAME filter.

Kenney and Leipnik [15] also modifies the Davison-Maki method. This method is reinitialised at each time step and thereby the algorithm is recast as a recursion. This also solves the problem of the Davison-Maki method becoming unstable over time and removes the need to update the matrix exponential. This method is more suitable to the time-invariant case, but can still be used on time-dependent RDEs if the fundamental matrix is known.

In general, algorithms based on Theorem 1 will not be very accurate for stiff RDEs, as the effects of stiffness may cause $U(t)$ to be ill conditioned for inversion [5]. This limits their applicability to solving the gain equation of the GAME filter, as it is stiff.

Schiff and Shnider [24] also build on Kenney and Leipnik's modified Davison-Maki method. They call their class of methods Möbius schemes. Their advantage comes from viewing the RDE as a flow on the Grassmannian of n-dimensional subspaces of a $(n + m)$-dimensional vector space. This ap-

proach allows for numerical integration past singularities and for stiffness to be resolved. The ability to integrate past singularities is not relevant in this case, because the GAME filter should avoid singularities. The ability to resolve stiffness, however, allows this method to be used even though it uses Theorem 1.

Chandrasekhar algorithms [16] are another type of method for solving RDEs. These methods transform the RDE into two smaller coupled differential equations. These methods are generally faster, but less accurate than the Davison-Maki method [15].

Harnad et al. [10] derived superposition formulae such that RDEs of the form (1) could be expressed in terms of at most five known solutions. There are numerical integration methods based on these formulae, such as [21]. These methods allow for values of X at a specific time t_n to be found without first finding $X(t_i)$ for all $i < n$. However, this property is not particularly useful for numerically integrating the gain equation of the GAME filter, as navigation systems require the attitude to be found frequently instead of just at some specific time.

Another type of methods for solving RDEs are Matrix-valued methods [2]. These methods are based on algorithms for solving non-matrix ODEs. Backwards differentiation formulas (BDFs) are especially applicable for solving stiff RDEs, such as the gain equation of the GAME filter.

Choi's method [5] is of this type. This method assumes that the RDE is

integrated over a time interval $[t_0, T]$, every entry of Q, A, B and R is continuously differentiable and every entry of the solution is $r + 1$ times differentiable. $X(t)$ can be approximated by a matrix polynomial to obtain an algebraic Riccati equation (ARE). One step of Newton's method [8] can then be used to obtain a Sylvester equation, which can be solved using the Hessenberg-Schur method [7] to get an approximation of X. This method is suitable for solving stiff RDEs. Also, it only requires $O(n^3)$ flops per time step. Other widely used methods for stiff RDEs, not necessarily including Möbius schemes, require $O(n^6)$ flops [5, 4].

The MEKF has been implemented using Choi's method [11], however there were difficulties when attempting to implement the GAME filter as the gain equation can only be written in the form (1) if A and B depend on X. However, it is likely that methods developed for RDEs could be modified to work with the GAME filter. Some methods for solving time-varying RDEs are based on techniques for solving time-invariant RDEs. And when the gain equation of the GAME filter is consider at only a certain time t it is indistinguishable from a RDE (for detail see section 5). So it makes intuitive sense that such methods could be modified to work with the GAME filter.

This report considers modifications of Choi's method and the Möbius scheme. It also uses Euler's method, which is a numerical integration that takes a general ODE as input, and so does not need to be modified in order to implement the GAME filter.

4 Background Information

4.1 Attitude Filters

Practical attitude filters use measurements of the angular velocity of the robot and the direction of forces acting on the robot or features in the environment observed by the robot. These measurements can be used to produce two different estimates of the attitude in a predictor/corrector framework. The estimates are then balanced by the gain term to produce an estimate of the attitude which takes both sets of measurements into account.

The attitude kinematics are given by [29]

$$\dot{X}(t) = X(t)\Omega_{\times}(t)$$

where X is the rotation matrix describing the relative orientation of a body-fixed reference frame with respect to an inertial frame, and Ω is the angular velocity vector.

The measurement models used for this project include noise but not bias. The models are as follows [29]:

$$u(t) = \Omega(t) + Bv(t)$$
$$y_i = X(t)\mathring{y}_i + D_i w_i(t), \quad i = 1, 2$$

where u is the measurement of the angular velocity, y_i are the measurements of the direction of forces, $\mathring{y}_i$ are the known directions of these forces relative to the inertial frame, and v and w_i are the measurement errors for u and y_i, respectively. B and D_i are known coefficient matrices for the measurement

errors that allow for the modelling of different levels of sensitivity for individual measurement channels.

The dynamics of the attitude estimate $\hat{X}$ uses the gain P as follows [27]:

$$\dot{\hat{X}} = \hat{X}(u - Pl)_\times$$

where the innovation term, l is given by

$$l = \sum_i (R_i^{-1}(\hat{y}_i - y_i)) \times \hat{y}_i,$$

$R_i := D_i D_i^T$ and $\hat{y}_i := \hat{X}^T \mathring{y}_i$.

The Multiplicative Extended Kalman Filter (MEKF) and the Geometric Approximate Minimum-Energy (GAME) filter differ only in their gain equations. The gain equation of the MEKF is given by [29]

$$\dot{P} = Q + P\frac{1}{2}u_\times - \frac{1}{2}u_\times P - PSP$$

with

$$Q := BB^T$$
$$S := \sum_i (\hat{y}_\times^T R_i^{-1}(\hat{y}_i)_\times)$$

The gain equation of the GAME filter is given by [29]

$$\dot{P} = Q + \mathbb{P}_S(P(2u - Pl)_\times) - PSP + PAP$$

with Q and S defined as above and the additional term A defined by

$$A := trace(C)I - C$$
$$C := \sum_i \mathbb{P}_S(R_i^{-1}(\hat{y}_i - y_i)\hat{y}_i^T)$$

Note that the gain equation of the GAME filter also has a Pl term that is absent from the gain equation of the MEKF. It is the presence of this term that means that gain equation of the GAME filter is not a RDE, but a generalised RDE.

In order to simulate an attitude filter system, the values of parameters, such as B and D_i, need to be known. These parameters vary depending on the type of device whose attitude is being measured as well as the accuracy of its measuring devices. Zamani et al. [29] compiled typical values of these parameters for UAVs and satellites.

4.2 Euler's Method

Euler's method is non-geometric, that is, it does not require a specific form of ODE as input. A general ODE can be expressed as

$$y'(t) = f(t, y(t))$$

Euler's method is derived using Taylor expansion [9]. At each time step, the solution is replaced by the first term of its Taylor expansion. The derivative between each time step is approximated as constant, and so can be given by

$$y'(t) \approx \frac{y(t+h) - y(t)}{h}.$$

Rearranging this gives

$$y(t+h) = y(t) + hy'(t)$$

which can be written as

$$y_{n+1} = y_n + hf(t_n, y_n)$$

in recursive form.

4.3 Choi's Method

Choi's method [5] is geometric, requiring a RDE as input. This RDE can be expressed as

$$\dot{X} = Q + XA + BX - XRX$$

It is assumed that the RDE is integrated over a time interval $[t_0, T]$, every entry of Q, A, B and R is continuously differentiable and every entry of the solution is $r+1$ times differentiable. Therefore, if the solution at time t_i with $i = 0, 1, ..., k$ and $1 \leq r \leq k+1$ is known, then

$$\frac{1}{h}\sum_{v=1}^{r}\frac{1}{v}\Delta^v X_{k+1} = \dot{X}_{k+1}$$

Where h is the time step.

The following algebraic Riccati equation (ARE) can then be obtained

$$\bar{Q}_{k+1,r} + X_{k+1}\bar{A}_{k+1,r} + \bar{B}_{k+1,r}X_{k+1} - X_{k+1}\bar{R}_{k+1,r}X_{k+1}$$

where

$$\bar{A}_{k+1,r} = -\left(\frac{1}{2}\sum_{i=1}^{r}\frac{1}{i}\right) I_n + hA(t_{k+1})$$
$$\bar{B}_{k+1,r} = -\left(\frac{1}{2}\sum_{i=1}^{r}\frac{1}{i}\right) I_m + hB(t_{k+1})$$
$$\bar{Q}_{k+1,r} = \sum_{i=1}^{r}\frac{(-1)^{i-1}}{i}\begin{pmatrix} r \\ i \end{pmatrix} X_{k-i+1} + hQ(t_{k+1})$$
$$\bar{R}_{k+1,r} = hR(t_{k+1})$$

Newton's method [8] can then be used to obtain the following Sylvester equation,

$$X_{k+1,(i+1)}(\bar{A}_{k+1,r} - \bar{R}_{k+1,r}X_{k+1,(i)}) + (\bar{B}_{k+1,r} - X_{k+1,(i)}\bar{R}_{k+1,r})X_{k+1,(i+1)} + \bar{Q}_{k+1,r} + X_{k+1,(i)}\bar{R}_{k+1,r}X_{k+1,(i)} = 0.$$

This equation can then be solved using the Hessenberg-Schur method [7] to get an approximation of $X_{k+1,(i+1)}$.

4.4 Möbius Schemes

Möbius schemes [24] are also geometric, requiring a RDE as input.

Möbius schemes are based on Theorem 1. They use the Bernoulli substitution $X(t) = V(t)U^{-1}(t)$, where

$$\begin{bmatrix} \dot{U}(t) \\ \dot{V}(t) \end{bmatrix} = \begin{bmatrix} -A(t) & R(t) \\ Q(t) & B(t) \end{bmatrix} \begin{bmatrix} U(t) \\ V(t) \end{bmatrix} \tag{2}$$

and $U(0) = I, V(0) = X(0)$.

This allows the solution to be expressed in the form

$$X_{k+1} = (\Phi_{22}(h)X_k + \Phi_{21}(h))\,(\Phi_{12}(h)X_k + \Phi_{11}(h))^{-1}$$

where Φ is the fundamental matrix of a RDE of form (1).

This fundamental matrix is then approximated at each time step by

$$\Phi_{11}(t,h) = I - hA(t)$$
$$\Phi_{12}(t,h) = hR(t)$$
$$\Phi_{21}(t,h) = hQ(t)$$
$$\Phi_{22}(t,h) = I + hB(t)$$

This well approximates the fundamental matrix, with an error of only $o(h)$ [24].

5 Modifications to Möbius Schemes and Choi's Method

This section introduces a modification to the Möbius scheme and Choi's method which work with the gain equation of the GAME filter by approximating the gain equation as a Riccati equation at each time step.

As stated previously, the gain equation of the GAME filter is $\dot{P} = Q + \mathbb{P}_S(P(2u - Pl)_\times) - PSP + PAP$. If $(Pl)_\times$ were independent of P, then the

gain equation of the GAME filter would be a Riccati equation. So the idea behind the modification is to consider the gain equation of the GAME filter only at the current time, t. The gain equation of the GAME filter can then be approximated as $\dot{P} = Q + \mathbb{P}_S(P(2u - L)_\times) - PSP + PAP$, where L is independent of P, but it just so happens that $L(t) = P(t)l(t)$. The gain equation of the GAME filter can be approximated in this way at any time t, but a new L is obtained each time. This means that a different RDE approximates the gain equation at each time step.

The intuitive idea behind why this might work is that the Möbius scheme approximates the fundamental matrix at a time t based only on the values of the coefficients at time t. At time t, the gain equation of the GAME filter and its Riccati approximation coincide, so the approximation of the fundamental matrix obtained using the Möbius scheme also coincides. Therefore a good approximation of the fundamental matrix of the gain equation of the GAME filter at time t is obtained, which can then be used to estimate the gain at time $t + h$.

The same modification is used for Choi's method. Although it turns out that this does not work as well as it does for the Möbius scheme (see section 6.1).

6 Simulation Study

In this section, various implementations of the Multiplicative Extended Kalman Filter (MEKF) and the Geometric Approximate Minimum-Energy (GAME)

filter are compared using MATLAB. The MEKF is implemented using Euler's method, Choi's method and the Möbius scheme. The GAME filter is implemented using Euler's method, and the modified versions of Choi's method and the Möbius scheme described in section 5. The attitude system itself - including angular velocity and measurement noise - is also simulated. This means that the true attitude is known, and so can be compared to the estimate. It also reduces possible sources of error and allows for easier manipulation of parameters.

The algorithms used to implement each method are described below, for an input of the form (1), and a time step of h. The full code of the simulation study is provided in appendix 1.

Euler's method

$$X_{k+1} = X_k + h\dot{X}_k$$

Möbius schemes

$$\Phi_{11}(t,h) = I - hA(t)$$

$$\Phi_{12}(t,h) = hR(t)$$

$$\Phi_{21}(t,h) = hQ(t)$$

$$\Phi_{22}(t,h) = I + hB(t)$$

$$X_{k+1} = \left(\Phi_{22}(t,h)X_k + \Phi_{21}(t,h)\right)\left(\Phi_{12}(t,h)X_k + \Phi_{11}(t,h)\right)^{-1}$$

Choi's method

$$\bar{A}_{k+1,r} = -\left(\frac{1}{2}\sum_{i=1}^{r}\frac{1}{i}\right) I_n + hA(t_{k+1})$$

$$\bar{B}_{k+1,r} = -\left(\frac{1}{2}\sum_{i=1}^{r}\frac{1}{i}\right) I_m + hB(t_{k+1})$$

$$\bar{Q}_{k+1,r} = \sum_{i=1}^{r}\frac{(-1)^{i-1}}{i}\begin{pmatrix} r \\ i \end{pmatrix} X_{k-i+1} + hQ(t_{k+1})$$

$$\bar{R}_{k+1,r} = hR(t_{k+1})$$

The following Sylvester equation is then obtained,

$$X_{k+1,(i+1)}(\bar{A}_{k+1,r} - \bar{R}_{k+1,r}X_{k+1,(i)}) + (\bar{B}_{k+1,r} - X_{k+1,(i)}\bar{R}_{k+1,r})X_{k+1,(i+1)}$$
$$+\bar{Q}_{k+1,r} + X_{k+1,(i)}\bar{R}_{k+1,r}X_{k+1,(i)} = 0.$$

This Sylvester equation is then solved using the Hessenberg-Schur method, described in [7]

The following criteria are used to evaluate the implementations.

1. The resulting observer error, including value of the noise floor, the noise amplitude in steady-state and the rate of convergence,
2. Computational time, and
3. Robustness, in terms of both changing the time step and changing the noise level of the measurements.

The value and amplitude of the noise floor indicate how accurate the attitude filter is when operating in steady-state. The rate of convergence is less important in some applications, as the filter can begin running before it needs to be used. This is not, however, always possible. The computational time

is important because attitude filters have to run in real time. Also, some applications, such as quad-copters, require light-weight and therefore less powerful computers. Robustness is important because different applications experience different conditions. Therefore, an attitude filter that works well for one application might not work at all for another application. Changing the step size corresponds to changing the sampling frequency of the devices used to measure the angular velocity and direction of forces. This has the same effect as scaling the angular velocity by the inverse of the time step. Hence this criteria is relevant to application with measuring devices that have slow sample rates and applications where the robot has a high angular velocity.

6.1 Resultant Observer Error

This section looks at the error for typical conditions for an UAV. This corresponds to the following values of variables [29]

Time step	0.001 (s)
Simulation time	50 (s)
Angle of rotation initialisation error	$\mathcal{N} \sim (0, 60^2)^\circ$
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$
Input error B	$\mathcal{N} \sim (0, 25^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 30^2)\frac{^\circ}{s}$

Table 1: Typical parameters for an UAV

The following results are averaged over 100 runs. These results, and the

results in the rest of this report are for value of $r = 1$ for Choi's method and the modified Choi's method.

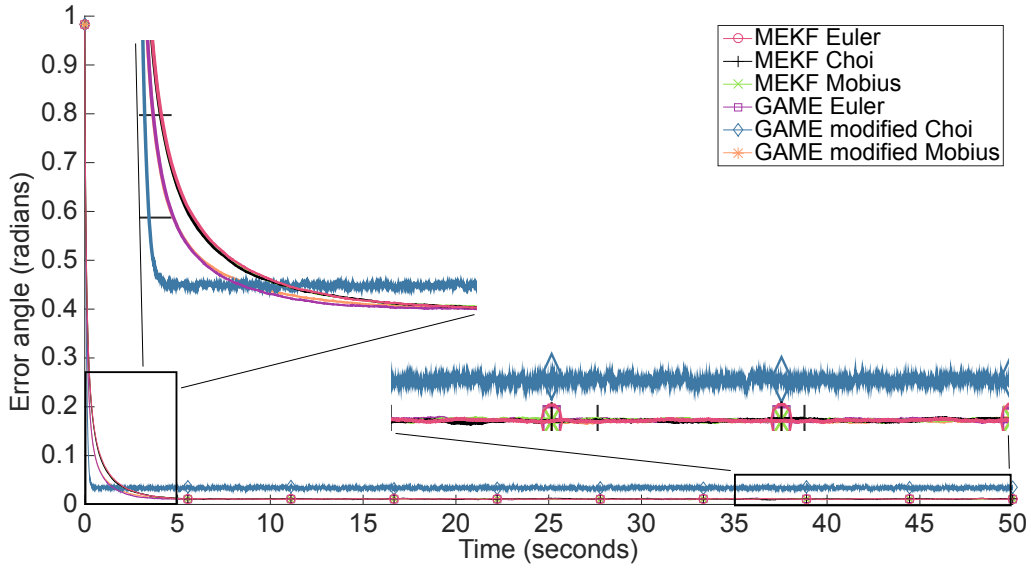


Figure 1: Resultant observer error for the MEKF and GAME filter, under typical UAV conditions

The convergence rate, value of the noise floor and noise amplitude are fairly similar for every implementation except for the GAME filter implemented with the modified Choi's method. The modified Choi's method is converging the fastest, but the noise floor is higher and the noise amplitude is greater. The higher noise floor could be corrected using gain scaling, but the greater noise amplitude cannot be easily fixed.

The GAME filter is converging slightly faster than the MEKF. But there does not appear to be a significant difference in the levels of the noise floor or noise amplitude between either the attitude filters or implementations, except for the GAME filter implemented with the modified Choi's method.

The rolling time integral of the angle of error is provided below, in order to compare the accumulation of error over time more easily.

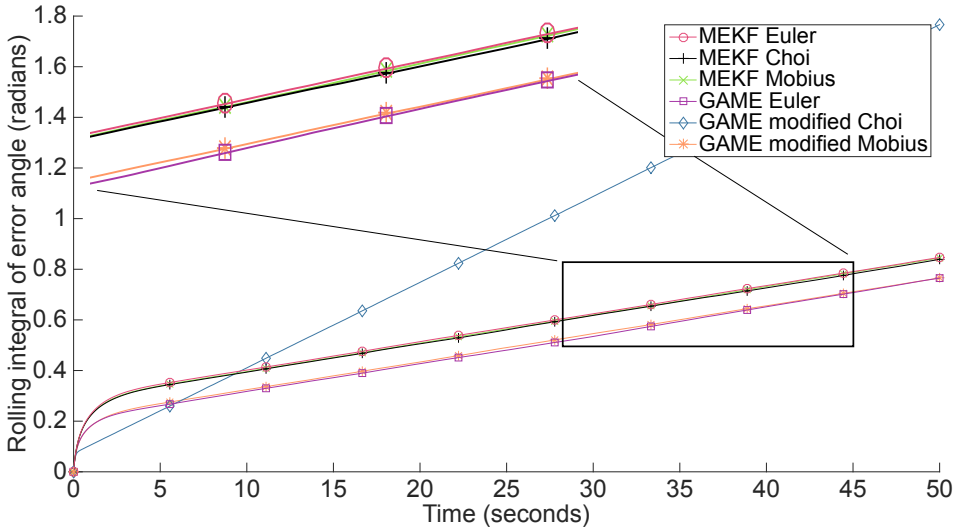


Figure 2: Rolling time integral of resultant observer error for the MEKF and GAME filter, under typical UAV conditions

This confirms the above observations that the GAME filter converges slightly faster than the MEKF, but there is otherwise not much of a difference between the attitude filter or implementations, with the exception of the modified Choi's method.

The fact that the GAME and MEKF converge to the same noise floor is expected, because their gains were scaled to achieve this.

The modified Choi's method, however, is producing unexpected results. The modified Möbius scheme and the modified Choi's method have the same error propagation properties, as they are using the same approximation of the gain

equation of the GAME filter. However, they are producing different results, so something must be different about their error accumulation. This could be due to the fact that the derivation of Choi's method relies on the differentiability of the solution. The approximation used to modify these methods (see section 5) has the same value of the gain equation at any specific time t, but not necessarily the same derivative.

From these results, it seems that the choice of attitude filter is more important than the choice of numerical integration scheme for minimising error under typical UAV conditions. Although it is possible to choose a numerical integration scheme that does not work well for implementing the attitude filter (i.e. the modified Choi's method), in which case the choice of implementation does have a significant impact.

6.2 Computational Time

The code of the simulation study is not optimised. However the majority of the code is implementing the attitude system and thus is the same for all of the methods. Therefore, the relative times of the methods should be fairly accurate. Although the possibility does exists that there are more efficient ways to implement one or more of the numerical integration schemes.

The following results are also for conditions typical for UAVs and are averaged over 100 runs. The times are given in μs per time step.

	Euler's method	Choi's method	Möbius scheme
MEKF	164.272	252.466	166.108
GAME	186.99	282.932	196.828

Table 2: Computational time for MEKF and GAME filter under typical UAV conditions

The computational time was also measured under a variety of different conditions. The full set of parameters and measurements is listed in appendix 2. The general trends are that the maximum time and the time step do not have much of an effect, but increasing the measurement error increases the computational time slightly. The averaged results are presented below.

	Euler's method	Choi's method	Möbius scheme
MEKF	167.427	255.454	169.243
GAME	191.116	286.919	200.959

Table 3: Computational time for MEKF and GAME filter averaged over various conditions

The relative speeds of the implementations seem to be about the same in both the typical and averaged cases. The MEKF is faster than the GAME filter, Euler's method is slightly faster than the Möbius scheme, and Choi's method is much slower.

This is as expected. The GAME filter has all of the terms of the MEKF, as well as additional terms. So it makes sense that it is slower. Euler's method only uses basic operations like addition and multiplication, which for 3×3

matrices do not take very long. Möbius schemes use basic operations as well, but they also require matrix inversion. However, inverting a 3×3 matrix is not very computationally intensive. Choi's method, however, requires solving a Sylvester equation, so it makes sense that it is significantly slower.

From these results, it seems that both the choice of attitude filter and numerical integration method make a significant difference to the computational time, but the numerical integration method can have more of an impact.

6.3 Robustness

In this section is concerned with the failure points of each of the methods. Additional graphs are included in appendix 3.

For the time step, only Euler's method stops producing output completely. This failure occurs after a time step of $h = 0.05$s for the GAME filter and $h = 0.043$s for the MEKF. The modified Choi's method has higher noise amplitude than the other methods, but this was the case even under typical UAV conditions. For the rest of the methods, the noise amplitude increases at approximately the same rate up to about $h = 1$s. However at around $h = 1.5$s the GAME filter with the modified Möbius scheme becomes much noisier than the MEKF with Choi's method or the Möbius scheme (see figure 3, which is from 1 run). Although at this point all of the implementations are extremely noisy, and therefore they would be unlikely to be used under such conditions.

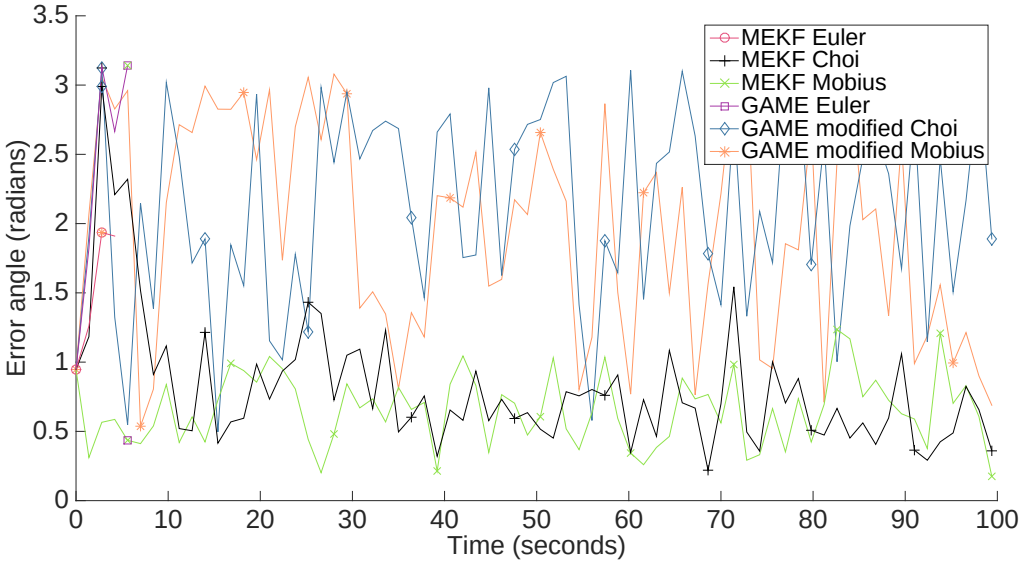


Figure 3: Resultant observer error for the MEKF and GAME filter, with a time step of 1.5

When increasing the standard deviation of the input error or the measurement error, only the GAME filter implemented with the modified Choi's method stops producing qualitatively correct output.

Certain combinations of the standard deviations of the input and measurement error affect different implementations differently. For example, the MEKF and the GAME filter differed for the following parameters.

Time step	0.001 (s)
Simulation time	60 (s)
Angle of rotation initialisation error	$\mathcal{N} \sim (0, 60^2)^\circ$
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$
Input error B	$\mathcal{N} \sim (0, 100^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 120^2)\frac{^\circ}{s}$

Table 4: Parameters for figure 4

The following graph is averaged over 100 runs.

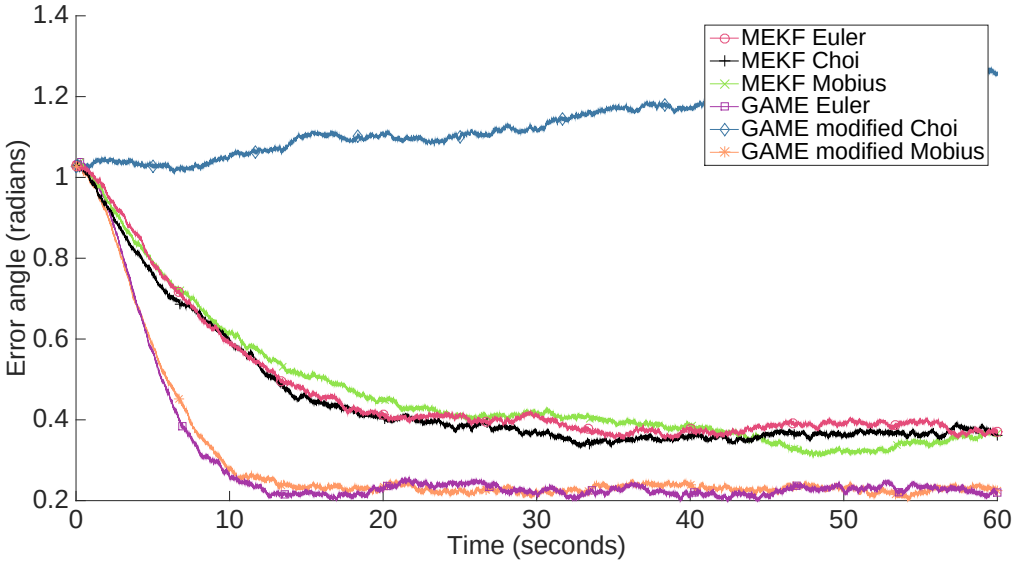


Figure 4: Resultant observer error for the MEKF and GAME filter, under conditions listed in table 4

Compared to the results under typical conditions for UAVs, the convergence rate is slower and the noise floor and noise amplitude is higher. Interestingly, the MEKF is now converging to a higher noise floor than the GAME filter. Also, the GAME filter with the modified Choi's method appears to stop converging.

The difference in the noise floors of the MEKF and GAME filter is likely due to the fact that the GAME filter takes into account second order geometry, whereas the MEKF is a first order approximation. When the difference between the estimated value of the rotation matrix and the true value is small, the first order and second order approximations are very close. But when the difference is big, the second order geometry plays a larger role.

The MEKF and GAME filter do differ under some circumstances, as discussed above. However, in most of the investigated cases, all of the implementations respond in the same way and at approximately the same rate.

A high standard deviation for the input error results in a fast convergence rate but a high noise floor and noise amplitude. This could be due to the fact that when B is large, Q is also large, which means $\dot{P}$ is large, and thus P is large. A large gain results in a faster convergence rate but a higher noise floor, as the estimate is more affected at each time step. The big variation in measurements of the angular velocity means that the dynamic model is not accurate, leading to increased noise.

Increasing the standard deviation of the measurement error makes all of the implementations very slow to converge. This could be due to the fact that when D_i are large, R_i are also large, so R_i^{-1} are small. R_i are not used in either MEKF or GAME filter directly, only R_i^{-1}. A small value of R_i^{-1} means that S, A and l are small. This means $\dot{P}$ and thus P are small. Also l scales the amount the attitude estimate is affected by the gain. Therefore large D_i result in the gain having less effect on the attitude estimate at each time step, and therefore a slower convergence rate.

From these results, it seems that both the choice of attitude filter and numerical integration method make a difference to the robustness with respect to the time step. Although the numerical integration method has much more of an impact. In many situations the implementations seem to be equally robust with respect to the error. However, for some combinations of large

input and measurement errors, it seems that the attitude filter can make a significant difference to the robustness with respect to the error. The numerical integration scheme still does not appear to have much of an effect. Although it is possible to choose a numerical integration scheme that does not work well for implementing the attitude filter (i.e. the modified Choi's method), in which case the choice of implementation does have a significant impact.

7 Conclusion

No best attitude filter or implementation is found by this report, as each has strengths and weaknesses. Therefore, which one should be used depends on the requirements of the application. However, the modified Möbius scheme does work relatively well for implementing the Geometric Approximate Minimum-Energy filter (GAME) filter. It is slightly slower than Euler's method, but it is much more robust with respect to increases in the value of the time step. This makes it a better choice for applications with measuring devices with slow sample rates or robots that experience a high angular velocity.

The full advantages of the GAME filter are not demonstrated by this report, as the measurement error model does not include bias. When bias is included, the GAME filter converges more quickly and has a lower noise floor than the Multiplicative Extended Kalman Filter (MEKF) and many other widely used filters [29]. Therefore, modifying the measurement error model of this project to include bias would likely help to further differentiate the

attitude filters, and possibly their implementations. The inclusion of bias would also make the simulation more realistic. At the very least, the performance of the modified Möbius scheme relative to Euler's method should be checked with bias.

The fact that a geometric method for RDEs was successfully modified to work with the GAME filter means that it could be possible to find more such methods that work for the GAME filter. Such methods would allow implementations to be better optimised for an application by taking advantage of the strengths of different numerical integration schemes.

Also, as mentioned in section 6.2, no the code used in this report is not optimised. In practical applications, the code would likely be optimised, in order to reduce the computational time. Optimising the code might change the results in the computational time section if a more efficient algorithm were used to implement one or more of the methods. It might also change other results due to effects on the numerics, although any effect is unlikely to be significant as optimising the code should have orders of magnitude less effect than changing the step size.

References

[1] H. Abou-Kandil. *Matrix Riccati equations: in control and systems theory*. Springer Science & Business Media, 2003.

[2] P. Benner and H. Mena. Bdf methods for large-scale differential riccati equations. *Proc. of Mathematical Theory of Network and Systems, MTNS*, 2004:10, 2004.

[3] S. Bittanti, A. J. Laub, and J. C. Willems. *The Riccati Equation*. Springer New York, 1991.

[4] C. H. Choi. A survey of numerical methods for solving matrix Riccati differential equations. In *Southeastcon'90. Proceedings., IEEE*, pages 696–700. IEEE, 1990.

[5] C. H. Choi and A. J. Laub. Efficient matrix-valued algorithms for solving stiff Riccati differential equations. *IEEE Transactions on Automatic Control*, 35(7):770–776, 1990.

[6] E. J. Davison and M. Maki. The numerical solution of the matrix Riccati differential equation. *IEEE Transactions on Automatic Control*, 18(1): 71–73, 1973.

[7] G. Golub, S. Nash, and C. Van Loan. A Hessenberg-Schur method for the problem AX+ XB= C. *IEEE Transactions on Automatic Control*, 24(6):909–913, 1979.

[8] C.-H. Guo and P. Lancaster. Analysis and modificaton of Newton's

method for algebraic Riccati equations. *Mathematics of Computation of the American Mathematical Society*, 67(223):1089–1105, 1998.

[9] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Oridnary Differential Equations I: Nonstiff Problems*. Springer, 2000.

[10] J. Harnad, P. Winternitz, and R. Anderson. Superposition principles for matrix Riccati equations. *Journal of Mathematical Physics*, 24(5): 1062–1072, 1983.

[11] C. Horgan. Numerical implementation of minimum-energy filters. Student project report, Australian National University, 2012.

[12] S. J. Julier and J. K. Uhlmann. New extension of the Kalman filter to nonlinear systems. In *AeroSense'97*, pages 182–193. International Society for Optics and Photonics, 1997.

[13] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[14] R. E. Kalman and T. Englar. *A user's manual for the automatic synthesis program*. CR-475. National Aeronautics and Space Administration, 1966.

[15] C. S. Kenney and R. Leipnik. Numerical integration of the differential matrix Riccati equation. *IEEE Transactions on Automatic Control*, 30 (10):962–970, 1985.

[16] D. G. Lainiotis. Generalized Chandrasekhar algorithms: Time-varying models. *IEEE Transactions on Automatic Control*, 21(5):728–732, 1976.

[17] A. J. Laub. A Schur method for solving algebraic Riccati equations. *IEEE Transactions on Automatic Control*, 24(6):913–921, 1979.

[18] E. J. Lefferts, F. L. Markley, and M. D. Shuster. Kalman filtering for spacecraft attitude estimation. *Journal of Guidance, Control, and Dynamics*, 5(5):417–429, 1982.

[19] R. Leipnik. A canonical form and solution for the matrix Riccati differential equation. *The Journal of the Australian Mathematical Society. Series B. Applied Mathematics*, 26(03):355–361, 1985.

[20] F. Markley and J. Crassidis. *Fundamentals of Spacecraft Attitude Determination and Control*. Space Technology Library. Springer New York, 2014.

[21] D. Rand and P. Winternitz. Nonlinear superposition principles: a new numerical method for solving matrix Riccati equations. *Computer physics communications*, 33(4):305–328, 1984.

[22] W. T. Reid. *Riccati differential equations*. Elsevier, 1972.

[23] I. Rusnak. Almost analytic representation for the solution of the differential matrix Riccati equation. *IEEE Transactions on Automatic Control*, 33(2):191–193, 1988.

[24] J. Schiff and S. Shnider. A natural approach to the numerical integration of Riccati differential equations. *SIAM Journal on Numerical Analysis*, 36(5):1392–1413, 1999.

[25] M. C. VanDyke, J. L. Schwartz, and C. D. Hall. Unscented Kalman filtering for spacecraft attitude state and parameter estimation. *Depart-*

ment of Aerospace & Ocean Engineering, Virginia Polytechnic Institute & State University, Blacksburg, Virginia, 2004.

[26] D. Vaughan. A negative exponential solution for the matrix Riccati equation. *IEEE Transactions on Automatic Control*, 14(1):72–75, 1969.

[27] M. Zamani. *Deterministic attitude and pose filtering, an embedded Lie groups approach*. PhD thesis, Australian National University, 2013.

[28] M. Zamani, J. Trumpf, and R. Mahony. Minimum-energy filtering for attitude estimation. *IEEE Transactions on Automatic Control*, 58(11): 2917–2921, 2013.

[29] M. Zamani, J. Trumpf, and R. Mahony. Nonlinear attitude filtering: A comparison study. *arXiv preprint arXiv:1502.03990*, 2015.

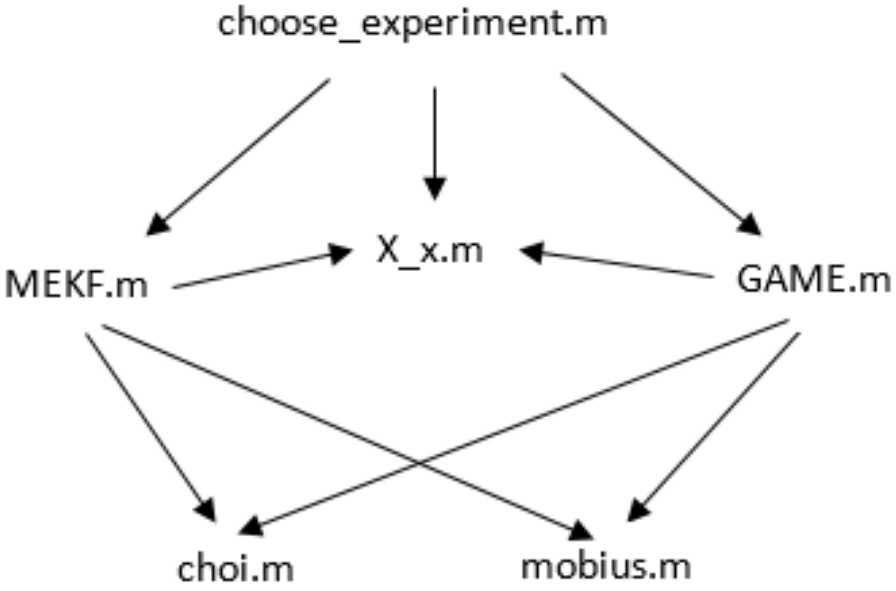
8 Appendices

8.1 Appendix 1: Code

This simulation study includes

1. four main functions,
2. some minor functions which the four main functions call and
3. a top-level script which is used to choose parameters for experiments.

These scripts interact as shown below.



```
%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: choose_experiment.m

% This script calls MEKF.m and GAME.m with the chosen parameters.
% This allows for easy comparison for attitude filter implementations
% with different parameters.

% This script allows for the following parameters to be changed:
    % The attitude filter(s) simulated
    % The numerical integration method(s) used to simulate each
        % attitude filter
    % Which measurements are displayed (graphs, times)
    % The maximum time of the simulation
    % The time step
    % The amount the velocity is scaled by
    % The value of r used for Choi's method
    % The data type used (error coefficient matrices, etc.)

clear;
%%
%%% CHOOSE METHOD
%Method used for numerical integration:
 %0 = OFF
 %1 = ON
%Colours for graph:
 %y, m, c, r, g, b, w, k

%%%MEKF:
%Euler (non-geometric)
```

```
MEKF_euler = 1;
%Colour
MEKF_euler_c = 'b';

%Choi
MEKF_choi = 1;
%Colour
MEKF_choi_c = 'g';

%Mobius
MEKF_mobius = 1;
%Colour
MEKF_mobius_c = 'k';

%%%GAME:
%Euler (non-geometric)
GAME_euler = 1;
%Colour
GAME_euler_c = 'r';

%pseudo-Choi (extremely noisy)
GAME_choi = 1;
%Colour
GAME_choi_c = 'y';

%pseudo-Mobius
GAME_mobius = 1;
%Colour
GAME_mobius_c = 'm';

%%
```

```
%%% CHOOSE WHICH GRAPHS
%Number of runs to average over
num = 1;

%Choose whether each graph is on or off:
 %0 = OFF
 %1 = ON
%Error vs time
err = 1;

%Rolling time integral
int = 0;

%Display time taken for each method
time = 0;

%%%

%%
%%% CHOOSE OTHER VARIABLES
%Choose data type
 %0 = Custom
 %1 = UAV (high noise)
 %2 = Satellite (low noise) - took values from paper, but not working
%Note, setting this to 2 will change the velocity simulation
Data_type = 1;

%Measurement directions:
%Normed to allow for easier changes
y1_d = [1;0;0];
y1_d = y1_d/norm(y1_d);
```

```
y2_d = [0;1;0];
y2_d = y2_d/norm(y2_d);

%Time:
h = 0.001;
t_0 = 0;
t_max = 50;

%Used in Choi's method:
r = 1;

%Scale velocity:
V = 1;

%True rotation mean:
%Randomised after std_q0 is defined
X_n_t_mean = eye(3);

%Initial estimate of rotation:
X_n = eye(3);

%%%

%%
%%% SETTING ERROR COEFFICIENTS BASED ON Data_type
%%% Custom:
if Data_type == 0
%Measurement error coefficients:
B_o = degtorad(25)^2*eye(3);
D_1 = degtorad(180)^2*eye(3);
D_2 = degtorad(180)^2*eye(3);
```

```
%Initial value error estimate:
std_q0 = degtorad(60);

%Initial estimate of P:
P = (1/std_q0^2)*eye(3);
%%%

%%% UAV - do not change:
elseif Data_type == 1
%Measurement error coefficients:
B_o = degtorad(25)^2*eye(3);
D_1 = degtorad(30)^2*eye(3);
D_2 = degtorad(30)^2*eye(3);

%Initial value error estimate:
std_q0 = degtorad(60);

%Initial estimate of P:
P = (1/std_q0^2)*eye(3);
%%%

%%% Satellite - do not change:
elseif Data_type == 2
%Measurement error coefficients:
B_o = (0.31623^2*10^(-6))*eye(3);
D_1 = degtorad(1)^2*eye(3);
D_2 = degtorad(1)^2*eye(3);

%Initial value error estimate:
std_q0 = degtorad(60);
```

```
%Initial estimate of P:
P = (0.1/std_q0^2)*eye(3);
%%%
end
%%%

%%
%%% RESULTS:
t = [t_0:h:t_max]';

%Initialising variables:
i = 0;
MEKF_av1 = zeros(size(t));
MEKF_av2 = zeros(size(t));
MEKF_av3 = zeros(size(t));
GAME_av1 = zeros(size(t));
GAME_av2 = zeros(size(t));
GAME_av3 = zeros(size(t));
legend_names = {};
MEKF_av1_time = 0;
MEKF_av2_time = 0;
MEKF_av3_time = 0;
GAME_av1_time = 0;
GAME_av2_time = 0;
GAME_av3_time = 0;

%Legend:
if MEKF_euler == 1
    legend_names = [legend_names, {'MEKF Euler'}];
end
```

```
if MEKF_choi == 1
    legend_names = [legend_names, {'MEKF Choi'}];
end
if MEKF_mobius == 1
    legend_names = [legend_names, {'MEKF Mobius'}];
end
if GAME_euler == 1
    legend_names = [legend_names, {'GAME Euler'}];
end
if GAME_choi == 1
    legend_names = [legend_names, {'GAME modified Choi'}];
end
if GAME_mobius == 1
    legend_names = [legend_names, {'GAME modified Mobius'}];
end

%While loop to get averages:
while i < num
%Randomising X_n_t using std_q0:
v_rand = std_q0*rand(3,1);
X_n_t = X_n_t_mean*expm(X_x(v_rand));

%Incrementing counter:
i = i + 1;

%Getting results as a matrix:
if MEKF_euler == 1
    tic
    MEKF_results = MEKF(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
        X_n, B_o, D_1, D_2, P, V, 1, Data_type);
    MEKF_av1_time = MEKF_av1_time + toc;
```

```
    MEKF_av1 = MEKF_av1 + MEKF_results;
end
if MEKF_choi == 1
    tic
    MEKF_results = MEKF(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
        X_n, B_o, D_1, D_2, P, V, 2, Data_type);
    MEKF_av2_time = MEKF_av2_time + toc;
    MEKF_av2 = MEKF_av2 + MEKF_results;
end
if MEKF_mobius == 1
    tic
    MEKF_results = MEKF(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ....
        X_n, B_o, D_1, D_2, P, V, 3, Data_type);
    MEKF_av3_time = MEKF_av3_time + toc;
    MEKF_av3 = MEKF_av3 + MEKF_results;
end
if GAME_euler == 1
    tic
    GAME_results = GAME(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
        X_n, B_o, D_1, D_2, P, V, 1, Data_type);
    GAME_av1_time = GAME_av1_time + toc;
    GAME_av1 = GAME_av1 + GAME_results;
end
if GAME_choi == 1
    tic
    GAME_results = GAME(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
        X_n, B_o, D_1, D_2, P, V, 2, Data_type);
    GAME_av2_time = GAME_av2_time + toc;
    GAME_av2 = GAME_av2 + GAME_results;
end
if GAME_mobius == 1
```

```
    tic
    GAME_results = GAME(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
        X_n, B_o, D_1, D_2, P, V, 3, Data_type);
    GAME_av3_time = GAME_av3_time + toc;
    GAME_av3 = GAME_av3 + GAME_results;
end
end

%Averaging:
MEKF_av1 = MEKF_av1/num;
MEKF_av2 = MEKF_av2/num;
MEKF_av3 = MEKF_av3/num;
GAME_av1 = GAME_av1/num;
GAME_av2 = GAME_av2/num;
GAME_av3 = GAME_av3/num;

MEKF_av1_time = MEKF_av1_time/num;
MEKF_av2_time = MEKF_av2_time/num;
MEKF_av3_time = MEKF_av3_time/num;
GAME_av1_time = GAME_av1_time/num;
GAME_av2_time = GAME_av2_time/num;
GAME_av3_time = GAME_av3_time/num;

%Plotting results:
t = [t_0:h:t_max]';

%Error
if err == 1;
figure;

if MEKF_euler == 1
```

```
    plot(t, MEKF_av1, MEKF_euler_c); hold on
end

if MEKF_choi == 1
    plot(t, MEKF_av2, MEKF_choi_c); hold on
end

if MEKF_mobius == 1
    plot(t, MEKF_av3, MEKF_mobius_c); hold on
end

if GAME_euler == 1
    plot(t, GAME_av1, GAME_euler_c); hold on
end

if GAME_choi == 1
    plot(t, GAME_av2, GAME_choi_c); hold on
end

if GAME_mobius == 1
    plot(t, GAME_av3, GAME_mobius_c); hold on
end

xlabel('Time (seconds)');
ylabel('Angle of error (radians)');
legend(legend_names);

end

%Rolling integral
if int == 1;
```

```
figure;

if MEKF_euler == 1
    MEKF_int1 = h*cumtrapz(MEKF_av1);
    plot(t, MEKF_int1, MEKF_euler_c); hold on
end

if MEKF_choi == 1
    MEKF_int2 = h*cumtrapz(MEKF_av2);
    plot(t, MEKF_int2, MEKF_choi_c); hold on
end

if MEKF_mobius == 1
    MEKF_int3 = h*cumtrapz(MEKF_av3);
    plot(t, MEKF_int3, MEKF_mobius_c); hold on
end

if GAME_euler == 1
    GAME_int1 = h*cumtrapz(GAME_av1);
    plot(t, GAME_int1, GAME_euler_c); hold on
end

if GAME_choi == 1
    GAME_int2 = h*cumtrapz(GAME_av2);
    plot(t, GAME_int2, GAME_choi_c); hold on
end

if GAME_mobius == 1
    GAME_int3 = h*cumtrapz(GAME_av3);
    plot(t, GAME_int3, GAME_mobius_c); hold on
end
```

```
xlabel('Time (seconds)');
ylabel('Rolling integral of error angle (radians)');
legend(legend_names, 'Location', 'southeast');

end

%Computational time
if time == 1
if MEKF_euler == 1
    MEKF_av1_time
end

if MEKF_choi == 1
    MEKF_av2_time
end

if MEKF_mobius == 1
    MEKF_av3_time
end

if GAME_euler == 1
    GAME_av1_time
end

if GAME_choi == 1
    GAME_av2_time
end

if GAME_mobius == 1
    GAME_av3_time
```

```
end
end
%%%


%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: X_x.m

% This script implements the lower index operator

%%
function X_x = X_x(v)
X_x = [0, -1*v(3), v(2); v(3), 0, -1*v(1); -1*v(2), v(1), 0];
end


%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: X_x.m

% This script implements Choi's method

% It takes h, r, A, B, Q, R and X as input,
% where h is the time step, A, B, Q and R are coeeficients of an RDE
% of form (1), with X as the variable
% and every entry of the solution is r + 1 times differentiable

%%
```

```
function X = choi(h, r, A, B, Q, R, X, X_array)
%Finding the coefficients in the Sylvester equation
A_bar = find_A_bar(r,A,h);
B_bar = find_B_bar(r,B,h);
Q_bar = find_Q_bar(r,Q,h,X_array);
R_bar = find_R_bar(r,R,h);

% Solving the Sylvester equation
X = sylvester((B_bar - X*R_bar), (A_bar - R_bar*X), ...
    -1*(Q_bar + X*R_bar*X));
end

%%
function A_bar = find_A_bar(r,A,h)
sum = 0;
    for i = 1:r
        sum = sum + 1/i;
    end
A_bar = (-1*sum/2)*eye(3) + h*A;
end

%%
function B_bar = find_B_bar(r,B,h)
sum = 0;
    for i = 1:r
        sum = sum + 1/i;
    end
B_bar = (-1*sum/2)*eye(3) + h*B;
end
```

```
%%
function Q_bar = find_Q_bar(r,Q,h,X_array)
sum = 0;
if size(X_array, 2)/3 - 1 < r
    for i = 1:r
        sum = sum + (((-1)^(i-1))/i)*nchoosek(r,i)*X_array(:,1:3);
    end
else
    for i = 1:r
        k = size(X_array, 2)/3 - 1;
        sum = sum + (((-1)^(i-1))/i)*nchoosek(r,i)*...
            X_array(:,(3*(k-i+1)+1):(3*(k-i+1)+3));
    end
end
Q_bar = sum + h*Q;
end

%%
function R_bar = find_R_bar(r,R,h)
R_bar = h*R;
end



%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: X_x.m

% This script implements the Mobius scheme

% It takes h, A, B, Q, R and X as input,
```

```
% where h is the time step and A, B, Q and R are coeeficients of an RDE
% of form (1), with X as the variable

%%

function X = mobius(h, A, B, Q, R, X)
alpha = eye(3) + h*B;
beta = h*Q;
gamma = h*R;
delta = eye(3) - h*A;
X = (alpha*X + beta)*inv(gamma*X+delta);
end



%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: MEKF.m

% This script implments the MEKF with the chosen parameters.
% The main function is MEKF, but it also calls
    % find_O, which finds the angular velocity
    % fP, which finds the deriviative of the gain
    % fX, which finds the derivative of the rotation matrix using
        % the gain, the angular velocity measurement and the
        % innovation term
    % X_noisy, which adds noise with a certain variance to a
        % variable
    % find_S, which finds S using the measurements of forces and
        % their error coefficient matrices
```

```
% This function also calls X_x.m

% choi.m and mobius.m may also be called, depending on the numerical
% integration scheme chosen to implement the MEKF.

%%
function error_array = MEKF(h, r, t_0, t_max, y1_d, y2_d, ...
    X_n_t, X_n, B, D_1, D_2, P, V, method, data_type)
t = t_0;
Q = B*transpose(B);
R_1 = D_1*transpose(D_1);
R_2 = D_2*transpose(D_2);
error_array = [];
%P = inv(trace(inv(K0)*eye(3)-inv(K0)));
P_array = P;

while t <= t_max + h/10
    %O is true value of omega, u is the measurement
    O = V*find_O(t, data_type);
    u = X_noisy(O, B);

    %y1 and y2 give true values
    y1 = transpose(X_n_t)*y1_d;
    y2 = transpose(X_n_t)*y2_d;

    %Simulating measured yi values
    y1_meas = X_noisy(y1, D_1);
    y2_meas = X_noisy(y2, D_2);
    y1_meas = y1_meas/norm(y1_meas);
    y2_meas = y2_meas/norm(y2_meas);
```

```
%Estimating yi
y1_n = transpose(X_n)*y1_d;
y2_n = transpose(X_n)*y2_d;

S = find_S(y1_n, y2_n, R_1, R_2);

%Comparing true to estimate:
error_matrix = X_n*X_n_t';
error_angle = acos((error_matrix(1,1) + error_matrix(2,2) ...
    + error_matrix(3,3) - 1)/2);
error_array = [error_array; error_angle];

O_x = X_x(O);
u_x = X_x(u);

%Updating X_n_t using Euler's method - geometric
X_n_t = X_n_t*expm(h*O_x);

%Updating P using Euler's method -non-geometric (method 1):
if method == 1
P = P + h*fP(Q, P, u, S);

%Updating P using Choi's method (method 2):
elseif method == 2
P = choi(h, r, (u_x/2), -1*(u_x/2), Q, S, P, P_array);
P_array = [P_array, P];

%Updating P using a Mobius scheme (method 3):
elseif method == 3
P = mobius(h, (u_x/2), -1*(u_x/2), Q, S, P);
```

```
    end

    %Updating X_n using Euler's method - geometric:
    X_n = X_n*expm(h*fX(P, X_n, u, R_1, R_2, y1_n, y2_n, ...
        y1_meas, y2_meas));

    t = t + h;
end

end

%%
%Angular velocity simulation:
function O = find_O(t, type)
% w1_n = rand(1);
% w2_n = rand(1);
% w3_n = rand(1);
% O = [w1_n; w2_n; w3_n];
if type <= 1
O = [sin(2*pi*t/15); -1*sin(2*pi*t/18 + pi/20); cos(2*pi*t/17)];

elseif type == 2
O = sin(2*pi*t/150)*[1;-1;1];

end
end

%%
%Gain equation - used for Euler's method (non-geometric):
function dP = fP(Q, P, u, S)
dP = Q + P*X_x(u)/2 - X_x(u)*P/2 - P*S*P;
```

```
end

%%
%Relating gain to X_n:
function dX = fX(P, X_n, u, R_1, R_2, y1, y2, y1_meas, y2_meas)
l = cross((inv(R_1)*(y1 - y1_meas)),y1) + ...
    cross((inv(R_2)*(y2 - y2_meas)),y2);
dX = X_x(u - P*l);
end

%%
%Adding noise:
function X_noisy = X_noisy(M, k)
X_noisy = M + k * randn(3,1);
end

%%
%Working out S:
function S = find_S(y1, y2, R1, R2)
S = X_x(y1)'*inv(R1)*X_x(y1) + X_x(y2)'*inv(R2)*X_x(y2);
end



%%
% Author: Katrina Ashton
% Supervisor: Jochen Trumpf
% Script: GAME.m

% This script implments the GAME with the chosen parameters.
% The main function is GAME, but it also calls
    % find_O, which finds the angular velocity
```

```
    % fP, which finds the deriviative of the gain
    % fX, which finds the derivative of the rotation matrix using
        % the gain, the angular velocity measurement and the
        % innovation term
    % Ps, which finds the symmetric projection
    % X_noisy, which adds noise with a certain variance to a
        % variable
    % find_S, which finds S using the measurements of forces and
        % their error coefficient matrices
    % find_C, which finds C using the measurements of forces, their
        % knownreference directions and their error coefficient
        % matrices
    % find_A, which finds A using C

% This function also calls X_x.m

% choi.m and mobius.m may also be called, depending on the numerical
% integration scheme chosen to implement the MEKF.
%%
function error_array = GAME(h, r, t_0, t_max, y1_d, y2_d, X_n_t, ...
    X_n, B, D_1, D_2, P, V, method, data_type)
t = t_0;
Q = B*transpose(B);
R_1 = D_1*transpose(D_1);
R_2 = D_2*transpose(D_2);
error_array = [];
P_array = P;

while t <= t_max + h/10
    %O is true value of omega, u is the measurement
    O = V*find_O(t, data_type);
```

```
u = X_noisy(O, B);

%y1 and y2 give true values
y1 = transpose(X_n_t)*y1_d;
y2 = transpose(X_n_t)*y2_d;

%Simulating measured yi values
y1_meas = X_noisy(y1, D_1);
y2_meas = X_noisy(y2, D_2);
y1_meas = y1_meas/norm(y1_meas);
y2_meas = y2_meas/norm(y2_meas);

%Estimating yi
y1_n = transpose(X_n)*y1_d;
y2_n = transpose(X_n)*y2_d;

S = find_S(y1_n, y2_n, R_1, R_2);
C = find_C(y1_n, y2_n, y1_meas, y2_meas, R_1, R_2);
A = find_A(C);
l = find_l(R_1, R_2, y1_n, y2_n, y1_meas, y2_meas);

%Comparing true to estimate:
error_matrix = X_n*X_n_t';
error_angle = acos((error_matrix(1,1) + error_matrix(2,2) ...
    + error_matrix(3,3) - 1)/2);
error_array = [error_array; error_angle];

O_x = X_x(O);

%Updating X_n_t using Euler's method - geometric
X_n_t = X_n_t*expm(h*O_x);
```

```
    %Updating P using Euler's method -non-geometric (method 1):
    if method == 1
    P = P + h*fP(Q, P, u, S, A, l);

    %Updating P using Choi's method (method 2):
    elseif method == 2
    P = choi(h, r, -1*X_x(2*u-P*l)/2, X_x(2*u-P*l)/2, Q, S-A, ...
        P, P_array);

    %Updating P using a Mobius scheme (method 3):
    elseif method == 3
    P = mobius(h, -1*X_x(2*u-P*l)/2, X_x(2*u-P*l)/2, Q, S-A, P);
    end

    %Updating X_n using Euler's method - geometric:
    X_n = X_n*expm(h*fX(P, u, l));

    t = t + h;
end
end

%%
%Angular velocity simulation:
function O = find_O(t, type)
% w1_n = rand(1);
% w2_n = rand(1);
% w3_n = rand(1);
% O = [w1_n; w2_n; w3_n];
if type <= 1
O = [sin(2*pi*t/15); -1*sin(2*pi*t/18 + pi/20); cos(2*pi*t/17)];
```

```
elseif type == 2
O = sin(2*pi*t/150)*[1;-1;1];

end
end

%%
%Gain equation - used for Euler's method (non-geometric):
function dP = fP(Q, P, u, S, A, l)
dP = Q + Ps(P*X_x(2*u-P*l)) - P*S*P + P*A*P;
end

%%
%Relating gain to X_n:
function dX = fX(P, u, l)
dX = X_x(u - P*l);
end

%%
%Symmetric projection:
function sym = Ps(X)
sym = (X + X')/2;
end

%%
%Adding noise:
%(remember to normalise y1 and y2!)
function X_noisy = X_noisy(M, k)
X_noisy = M + k * randn(3,1);
end
```

```
%%
%Working out S:
function S = find_S(y1, y2, R1, R2)
S = X_x(y1)'*inv(R1)*X_x(y1) + X_x(y2)'*inv(R2)*X_x(y2);
end

%%
%Working out C:
function C = find_C(y1_n, y2_n, y1, y2, R1, R2)
C = Ps(inv(R1)*(y1_n-y1)*y1_n') + Ps(inv(R2)*(y2_n-y2)*y2_n');
end

%%
%Working out A:
function A = find_A(C)
A = trace(C)*eye(3) - C;
end

%%
%Working out l:
function l = find_l(R_1, R_2, y1, y2, y1_meas, y2_meas)
l = cross((inv(R_1)*(y1 - y1_meas)),y1) + cross((inv(R_2)*...
    (y2 - y2_meas)),y2);
end
```

8.2 Appendix 2: Computational Time Data

All of the data in this section is averaged over 100 runs. The units are μs per time step.

The first set of data is for typical UAV parameters for everything except the time step and the simulation time. That is,

Angle of rotation initialisation error	$\mathcal{N} \sim (0, 60^2)^\circ$
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$
Input error B	$\mathcal{N} \sim (0, 25^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 30^2)\frac{^\circ}{s}$

Table 5: Typical UAV conditions except for time step and simulation time

The following data was obtained using these parameters, with t_{max} as the simulation time, and h as the time step.

t_{max}	h	MEKF Euler	MEKF Choi	MEKF Möbius	GAME Euler	GAME modified Choi	GAME modified Möbius
20	0.001	161.785	255.365	162.850	186.130	285.835	196.640
20	0.005	157.975	249.725	158.725	180.725	281.300	191.450
10	0.001	161.250	252.590	163.080	183.250	283.600	193.210
30	0.001	165.480	253.710	169.127	188.790	285.383	193.277
50	0.001	164.272	252.466	166.108	186.990	282.932	196.828
10	0.01	167.600	255.100	168.800	190.400	286.300	200.400
20	0.01	165.550	258.150	168.150	190.250	287.400	199.050
30	0.01	167.533	255.100	169.700	190.767	286.200	201.167
40	0.01	166.500	254.550	169.025	189.750	286.025	199.100
50	0.01	166.840	254.600	168.400	189.880	286.060	199.300

Table 6: Computational time under parameters listed in table 5

Additional tests were also run to look at the effect of changing the standard deviation of the input and measurement error. The following parameters were common to all of these tests.

Time step	0.001 (s)
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$

Table 7: Common parameters for tests of computational time with standard deviation of errors varied

The results of the second set of tests are split into two tables for readability. t_{max} is the simulation time and $q0$ is the rotation initialisation error. The standard deviation, σ is listed for $q0$, B and D_i. That is, for $M \in \{q0, B, D_i\}$, $M = \mathcal{N} \sim (0, \sigma(M)^2)$.

t_{max}	$\sigma(q0)$	$\sigma(B)$	$\sigma(D_i)$	Euler	Choi	Möbius
20	90	50	60	171.945	257.530	173.545
20	60	50	60	174.030	259.810	175.495
20	60	100	120	174.355	258.140	175.910
50	60	100	120	173.286	257.734	175.200
50	120	100	120	173.002	257.238	174.532

Table 8: Computational time for the MEKF under parameters listed in table 7

t_{max}	$\sigma(q0)$	$\sigma(B)$	$\sigma(D_i)$	Euler	Modified Choi	Modified Möbius
20	90	50	60	196.180	290.030	206.865
20	60	50	60	198.420	291.685	209.805
20	60	100	120	199.325	290.865	210.025
50	60	100	120	198.006	290.260	208.782
50	120	100	120	197.884	289.908	208.488

Table 9: Computational time for the GAME filter under parameters listed in table 7

8.3 Appendix 3: Robustness Graphs

The graphs from the first set of tests are over just one run, as these tests are interested in the failure points of the implementations. Only the time step is changed during these tests. The other parameters have the following values.

Simulation time	20 (s)
Angle of rotation initialisation error	$\mathcal{N} \sim (0, 60^2)^\circ$
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$
Input error B	$\mathcal{N} \sim (0, 25^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 30^2)\frac{^\circ}{s}$

Table 10: Parameters for tests of robustness with respect to time step

The following graph is for a step size of 0.043. It can be seen that the

MEKF implemented with Euler's method experiences a large error spike before converging to the noise floor.

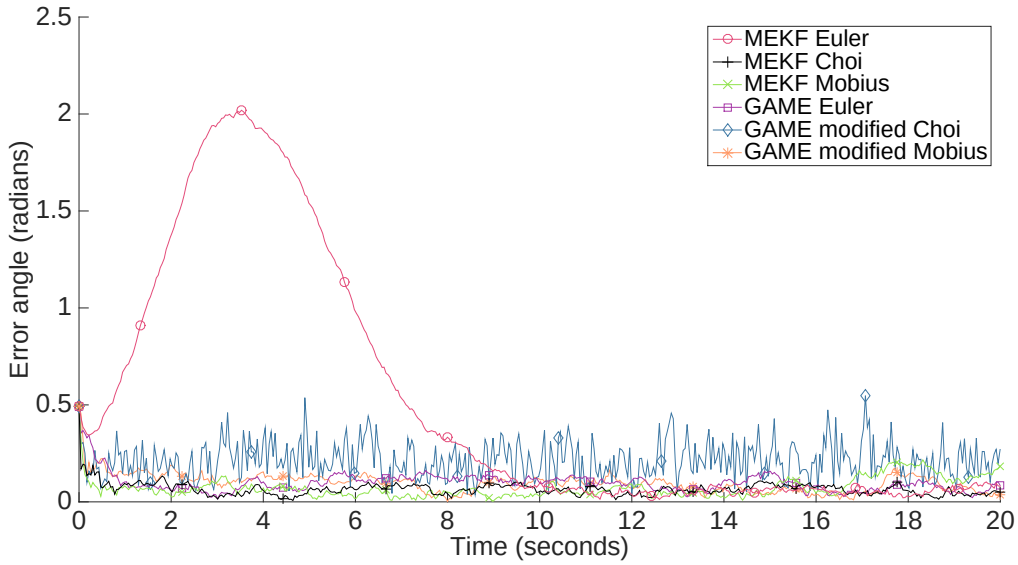


Figure 5: Resultant observer error for the MEKF and GAME filter, under conditions listed in table 10 and a time step of 0.043

The following graph is for a step size of 0.05. It can be seen that the GAME filter implemented with Euler's method experiences a large error spike for a while before converging to the noise floor.

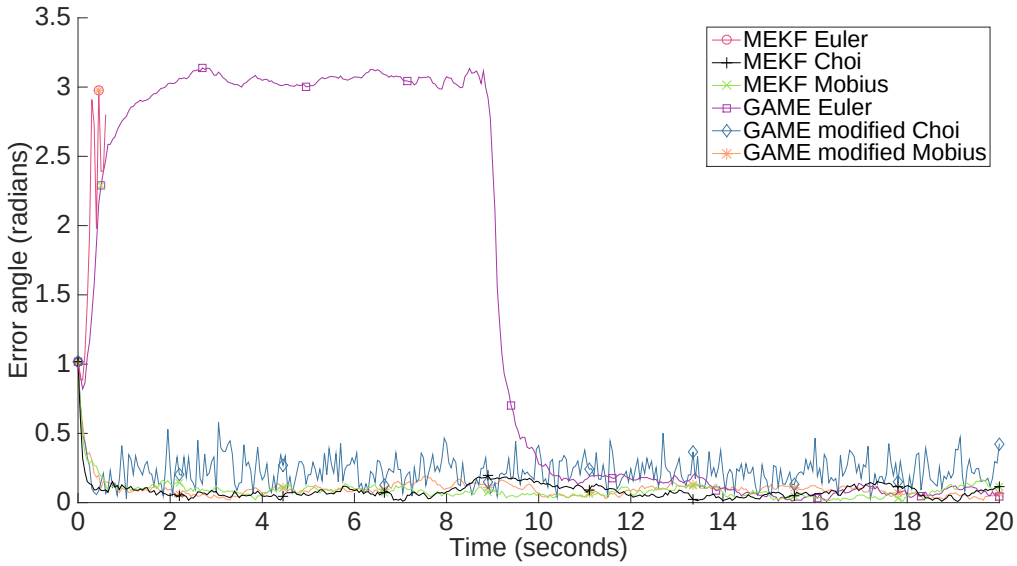


Figure 6: Resultant observer error for the MEKF and GAME filter, under conditions listed in table 10 and a time step of 0.05

The second set of tests looked at changing the errors. The parameters common to all of these tests had the following values.

Time step	0.001 (s)
Angle of rotation initialisation error	$\mathcal{N} \sim (0, 60^2)^\circ$
Reference directions	$\mathring{y}_1 = [1\ 0\ 0],\ \mathring{y}_2 = [0\ 1\ 0]$
Input signal	$\Omega = [\sin\left(\frac{2\pi}{15}t\right)\ -\sin\left(\frac{2\pi}{18}t + \frac{\pi}{20}\right)\ \cos\left(\frac{2\pi}{17}t\right)]\frac{rad}{s}$

Table 11: Parameters for tests of robustness with respect to standard deviation of errors

The effect of increasing the standard deviation of the input error can be demonstrated using the following additional parameters.

Simulation time	20 (s)
Input error B	$\mathcal{N} \sim (0, 100^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 30^2)\frac{^\circ}{s}$

Table 12: Additional parameters for a test of robustness with respect to the standard deviation of the input error

The following graph is averaged over 100 runs.

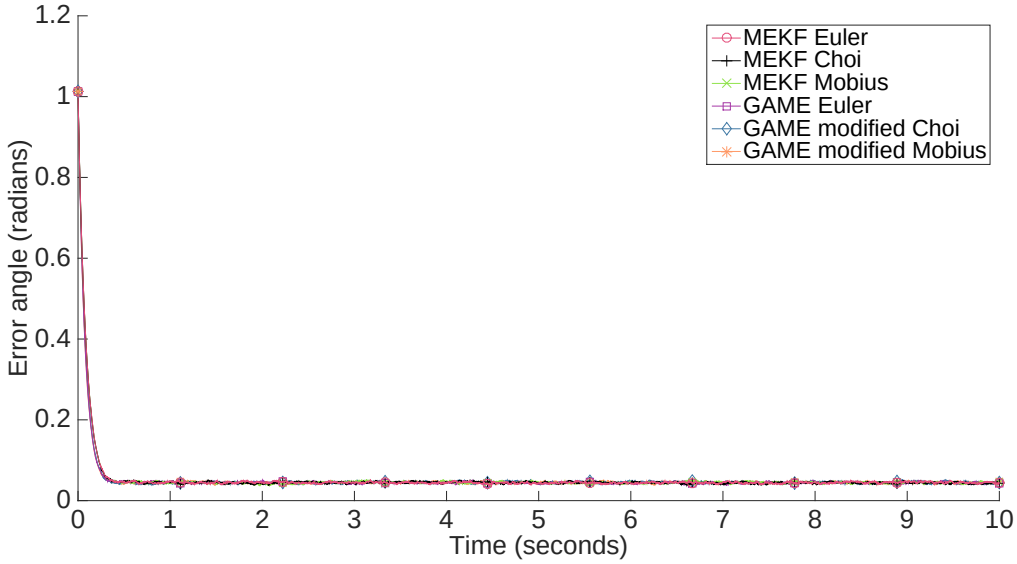


Figure 7: Resultant observer error for the MEKF and GAME filter, under conditions listed in tables 11 and 12

Compared to the results under typical conditions for UAVs the convergence rate is fast, but the noise floor and noise amplitude are high.

The effect of increasing the standard deviation of the measurement error can be demonstrated using the following additional parameters.

Simulation time	100 (s)
Input error B	$\mathcal{N} \sim (0, 25^2)\frac{^\circ}{s}$
Measurement error D_i	$\mathcal{N} \sim (0, 60^2)\frac{^\circ}{s}$

Table 13: Additional parameters for a test of robustness with respect to the standard deviation of the measurement error

The following graph is averaged over 100 runs.

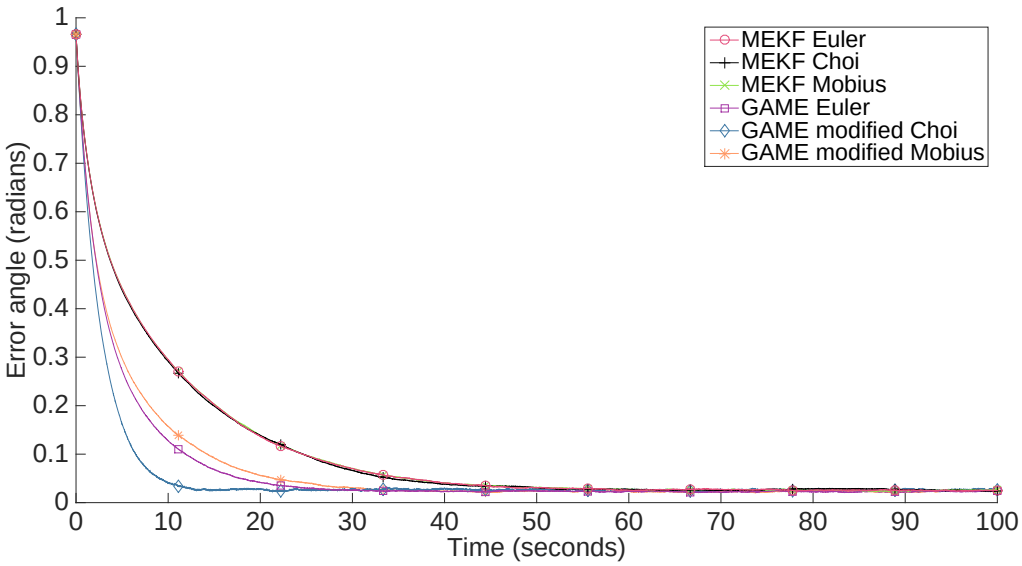


Figure 8: Resultant observer error for the MEKF and GAME filter, under conditions listed in tables 11 and 13

Compared to the results under typical conditions for UAVs, the convergence rate is extremely slow. Interestingly, the GAME filter implemented with the modified Choi's method is now converging to the same noise floor as the other methods.