NASA/TM—2017-219695

Pointing System Simulation Toolbox With Application to a Balloon Mission Simulator

Rosana M. Maringolo Baldraco, Eliot D. Aretskin-Hariton, and Aaron J. Swank
Glenn Research Center, Cleveland, Ohio

October 2017

NASA/TM—2017-219695

Pointing System Simulation Toolbox With Application to a Balloon Mission Simulator

Rosana M. Maringolo Baldraco, Eliot D. Aretskin-Hariton, and Aaron J. Swank
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

October 2017

Acknowledgments

This development of the Python toolbox for attitude determination simulation was funded by the Gondola for High Altitude Planetary Science (GHAPS) project. The authors would like to thank Calvin Robinson and George L. Thomas for their time in reviewing and debugging, Dean Schrage for helping with Ballon Mission Simulator integration, and Jim Lanzi for his help with Kalman filter implementation.

Pointing System Simulation Toolbox With Application to a Balloon Mission Simulator

Rosana M. Maringolo Baldraco, Eliot D. Aretskin-Hariton, and Aaron J. Swank
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

The development of attitude estimation and pointing-control algorithms is necessary in order to achieve high-fidelity modeling for a Balloon Mission Simulator (BMS). A pointing system simulation toolbox was developed to enable this. The toolbox consists of a star-tracker (ST) and Inertial Measurement Unit (IMU) signal generator, a UDP (User Datagram Protocol) communication file (bridge), and an indirect-multiplicative extended Kalman filter (imEKF). This document describes the Python toolbox developed and the results of its implementation in the imEKF.

1 Introduction

The use of telescopes for planetary science observations is an emerging use of high altitude balloons. Instead of using complicated adaptive optics, the balloon allows the observatory to be located in about 98% vacuum and the optical observations are less affected by the atmosphere. In addition, scientific observations from balloons may be as much as $1/30^{th}$ of the cost associated with similar space-based observations [1]. Lastly, the instrumentation is normally retrieved after every mission, allowing for multiple flights.

A Balloon Mission Simulator (BMS) has been developed[1]. This simulator will take advantage of the Pointing System Simulation Toolbox (PSST) presented in this document to add high-fidelity pointing simulation. This toolbox has the capability of emulating different star tracker (ST) and Inertial Measurement Unit (IMU) combinations, allowing projects to select the best one for their application. The PSST was developed in Python and consists of a true signal generator, a UDP (User Datagram Protocol) communication bridge, a sensor signal simulation that adds errors to the signal, and an attitude indirect-multiplicative extended Kalman filter (imEKF).

A true signal generator creates sensor signals that are sent over UDP bridge to the sensor signal simulation. The sensor signal simulation adds hardware errors (bias) and noise and calculates IMU Euler angles and angle rates. This information is then sent to the imEKF for processing, via another UDP bridge. Typical IMUs generates angles and angle rates, and STs outputs attitude quaternions. For this reason, the sensor signal simulation gives out a quaternion, Euler angles, and angle rates. The imEKF processes these signals and generates an error in the attitude estimate (will be explained in more details in Section 4). The final output of the filter, however, is an estimated quaternion. The attitude estimation is then sent to the BMS via UDP bridge for pointing correction. Figure 1 shows the diagram of the entire system for illustration.

[1]LynxCAT, unpublished work developed by Dean Schrage (Zin Technologies/NASA GRC).

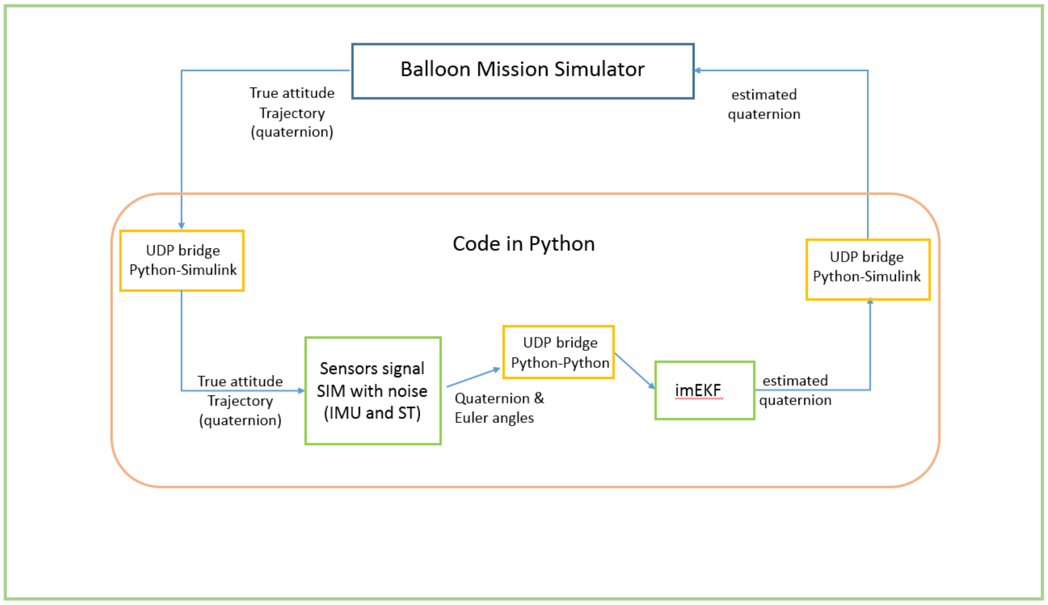

Balloon Mission Simulator
True attitude Trajectory (quaternion)
estimated quaternion
Code in Python
UDP bridge Python-Simulink
UDP bridge Python-Simulink
True attitude Trajectory (quaternion)
Sensors signal SIM with noise (IMU and ST)
UDP bridge Python-Python
Quaternion & Euler angles
imEKF
estimated quaternion

Figure 1: Diagram of data flow.

This document gives an overview of the PSST. Section 2 describes the true signal generation and the sensed signal simulation. Section 3 describes the bridge code. Section 4 describes the imEKF mathematical model. Results of the implementation are presented in Section 5.3.

2 True Signal Generator and Sensor Signal Simulation

Typically, the quaternion desired values are calculated in the Balloon Mission Simulator. This is a high-fidelity calculation of the expected pointing quaternion as a function of the position and dynamics of the balloon. This toolbox includes a quaternion generator that uses simplistic models for testing purposes only. Noise and measurement errors are added to the quaternions based on the characteristics of the sensors chosen. The file `ST_true_signal_UDP.py` generates the true signal. The file `sensors_signal_UDP.py` generates both the ST and the IMU signals with noise and errors.

The true signal values will come in a packet containing 40 bytes of data, as shown in Figure 3. The size of the data accounts for a quaternion (4 doubles = 32 bytes) and one double for the time stamp associated with the quaternion (8 bytes). The time stamp is in POSIX time[2] format.

The received data packet needs to be unpacked. After unpacking the data, it becomes a type *tuple*[3] with 5 components. The data needs to be extracted into

[2]Unix time (also known as POSIX time or epoch time) is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds. (Wikipedia)

[3]A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use

its many variables. In the case of the true attitude trajectory values data packet, the data will be extracted into two variables, the quaternion and the time stamp associated with it. It is important to note that the quaternions in the Python simulation are treated as right-handed (the scalar term is the 1^{st} term). The right-hand quaternion format was chosen to be consistent with the quaternion library (Reference [2]) utilized in the code.

Once the data is extracted and errors/noise are added, the data is packed in a variable and sent to the imEKF. The imEKF library we are using requires quaternions in left hand format, so we flip the quaternions into that format before processing data.

2.1 Star Tracker signal simulation

The ST signal simulation includes Gaussian white noise to the attitude quaternion true values coming from the MATLAB simulation. The method applied to include these errors is described below.

1- Generate roll, boresight, and cross-boresight Gaussian white noise by using a random number generator and multiplying it by each axis. The multiplication creates an error quaternion every time step. In the sensors simulation code the error quaternions are named q_{roll} for roll errors, q_{xb1} for boresight errors, and q_{xb2} for cross-boresight errors.

2- In addition to random errors (noise), very small bias errors are accounted for as well. In the code, the bias is generated by transforming the bias angles given in the hardware specs into bias quaternions, by means of an Euler-to-quaternion transformation matrix M (Eq.1). In the code, the given bias angles are named $thet$(for θ), phe(for ϕ), and cy(for ψ).

$$M = \begin{Bmatrix} \cos(\phi/2)*\cos(\theta/2)*\cos(\psi/2)+\sin(\phi/2)*\sin(\theta/2)*\sin(\psi/2) \\ \sin(\phi/2)*\cos(\theta/2)*\cos(\psi/2)-\cos(\phi/2)*\sin(\theta/2)*\sin(\psi/2) \\ \cos(\phi/2)*\sin(\theta/2)*\cos(\phi/2)+\sin(\phi/2)*\cos(\theta/2)*\sin(\psi/2) \\ \cos(\phi/2)*\cos(\theta/2)*\sin(\psi/2)-\sin(\phi/2)*\sin(\theta/2)*\cos(\psi/2) \end{Bmatrix} \tag{1}$$

3- Lastly, all errors are included to the true values by means of quaternion multiplications. Quaternion multiplications are the same as rotations. The small random rotations cause a jitter effect in the pointing quaternion, which is expected in real hardware.

2.2 IMU signal simulation

An Inertial Measurement Unit (IMU) typically outputs rotation rates (gyroscope) and/or Euler angles. This simulation outputs both rotation rates and Euler angles. In order to simulate the signal, the sensors generator code takes in the true quaternion values from MATLAB, calculates the change in quaternion between two

parentheses, whereas lists use square brackets (www.tutorialspoint.com).

consecutive quaternions (dq), the rotation angle between them (*turning angle*), and the time elapsed between the quaternion rotations. The change in quaternion (dq) is then transformed into Euler angles (yaw, pitch, and roll). The process to calculate the IMU angles and angle rates, as well as the addition of errors and noise to it will be described in this subsection.

2.2.1 Euler angles and angle rates calculation with errors and noise:

1- For every two consecutive true quaternions a change (quaternion rotation dq) is calculated by dividing the new quaternion ($q(t)$) by the previous quaternion ($q(t-1)$), or $qnew/qold$ in the code. This is the same as subtraction, in quaternion algebra.

2- The Euler angles can be calculated after dq's are obtained. An algorithm based on Trawny [3] Eq.106 was developed, as described below:

$$q_L = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$$

where q_L is the local frame that the rotations are expressed in,

$$Q = \begin{Bmatrix} q_L(4) & -q_L(3) & q_L(2) \\ q_L(3) & q_L(4) & -q_L(1) \\ -q_L(2) & q_L(1) & q_L(4) \\ -q_L(1) & -q_L(2) & -q_L(3) \end{Bmatrix} \tag{2}$$

and Q is the transformation matrix between a rotation quaternion and its correspondent Euler angles, where the angles are obtained by:

$$\begin{pmatrix} \Phi \\ \theta \\ \Psi \end{pmatrix} = 2 * Q^{\intercal} * dq \tag{3}$$

3- The errors are generated and included in the Euler angles.

4- The time elapsed between two quaternions calculation is simply a subtraction of the two consecutive time stamps associated with each quaternion.

5- The angle rates can then be calculated:

$$\begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} d\Phi/dt \\ d\theta/dt \\ d\Psi/dt \end{pmatrix}$$

where Φ, θ, and Ψ are the Euler angles with errors already included.

The IMU signal being sent to the imEKF will only contain the Euler angles for processing, because the imEKF is currently set to operate with angles and not angular rates. This can be changed as needed.

2.3 Data packet to the imEKF

At the end of the sensors signal simulation code, all information that needs to be sent to the imEKF will be concatenated in a list to form the data packet. The items being sent to the Kalman filter are listed below. They are placed in the following order inside the data packet:

1- The quaternion (with errors), in left-hand format (scalar term as the last term to be consistent with the imEKF algorithm). The quaternion contains 4 doubles.

2- The IMU time elapsed between two IMU signals (dt). It is used inside the imEKF for calculations. The variable dt is a double.

3- The ST time stamp associated with each ST quaternion generated in the file. The ST time stamp is a double.

4- The IMU time stamp associated with each IMU angle generated in the file. The IMU time stamp is a double.

5- The Euler angles, in the *yaw*, *pitch*, and *roll* order. The Euler angles contains 3 doubles.

The pack function sets the packet to contain 10 doubles (80 bytes) to account for all items above. The remote port is currently set to 20000, which is the local port of the imEKF file.

3 UDP Bridge

UDP stands for User Datagram Protocol and enables inter processes communication. There are two types of bridges: one is a Python-Python bridge that allows data exchange between two Python processes. The other is a Python-Simulink bridge and allows data exchange between Python and Simulink.

The bridge code consists of two parts: one part receives data, and the other one sends it. The communication link can be unidirectional (sends or receive data) or bidirectional (sends and receives data). Listing 1 shows a generic UDP bridge.

```
import socket
import sys
import time
import errno
from struct import pack
from struct import unpack

# UDP setup
host = "localhost"
LocalPort = 18000
RemPort = 25000
sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM, socket.
    IPPROTO_UDP)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((host,LocalPort))

```

```
16 while True:
17 t = time.time() #POSIX time
18
19 # attempt to read data
20 try:
21 dataRecv, addr = sock.recvfrom(1024)
22 except IOError as e: # some errors are ok, like #11 means no data
       available
23 err = e.args[0]
24 if err == errno.EAGAIN or err == errno.EWOULDBLOCK:
25 # no data available
26 dataRecv = '' # set the return value as an empty string
27 else: # really bad error occurred
28 # a real error occurred
29 print(e)
30 sys.exit(1)
31 else :
32 if dataRecv!='':
33 dataRecv = unpack('ddddd', dataRecv) #unpack the data string into a
       tuple
```
Listing 1: Excerpt from the UDP bridge generic code.

3.1 Python-Python Bridge Settings

This section describes how we modify the bridge to send and receive data in the PSST. The names inside the parenthesis are the names of the actual variables in the code.

> The host number (host): The host is the IP address of the machine where the process is running. In the Python UDP bridge code, it can either be set to 'localhost' or the IP address number. In the MATLAB 2010a UDP receive block the input needs to be the actual IP address ('localhost' will not work).

> Remote port (RemPort): this is the port number of the process where you are sending the data to. It is also the local port of the receiving process. It is an arbitrary number between 0 and 65536, however, port numbers between 0 and 1024 are reserved for privileged services and designated as well-known ports.

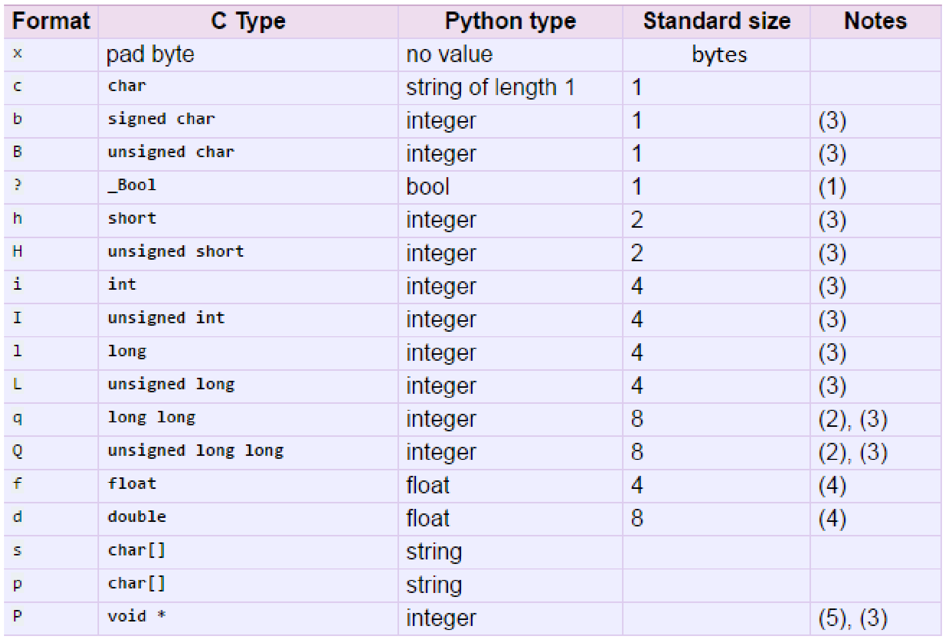
> Local port (LocalPort): this is the port number assigned to the process receiving data. It is also the remote port of the sending file.

> Buffer size: Inside the receive code, the sock.recvfrom (Listing 1, line 21) function has an input parameter, which is the buffer size to receive data. A default value for the buffer size is 1024 bytes.

> The data packet to be sent (packsend): this is the concatenated list of all data to be sent. All information needs to be included in one Python list object. Every item inside the `packsend` list has to be represented by a data type inside the pack function, as described next.

> Data type and size inside the pack function: the data size and type being sent needs to be established inside the pack function. A list of data types in Python can be found in Figure 2.

> Data type and size inside the unpack function: similar to the 'pack' function, the data size and type being received needs to be established inside the unpack function. One needs to know this information in order to receive the packets inside the file. In the example file, the unpack function is set to receive 40 bytes of data (5 doubles, 4 for the quaternion, and 1 for the time stamp associated with it). Please refer to Listing 1, line 33 of the code for illustration. Figure 3 shows the data packet bytes distribution.



Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value	bytes	
`c`	`char`	string of length 1	1	
`b`	`signed char`	integer	1	(3)
`B`	`unsigned char`	integer	1	(3)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	integer	2	(3)
`H`	`unsigned short`	integer	2	(3)
`i`	`int`	integer	4	(3)
`I`	`unsigned int`	integer	4	(3)
`l`	`long`	integer	4	(3)
`L`	`unsigned long`	integer	4	(3)
`q`	`long long`	integer	8	(2), (3)
`Q`	`unsigned long long`	integer	8	(2), (3)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`s`	`char[]`	string		
`p`	`char[]`	string		
`P`	`void *`	integer		(5), (3)

Figure 2: Python binary codes

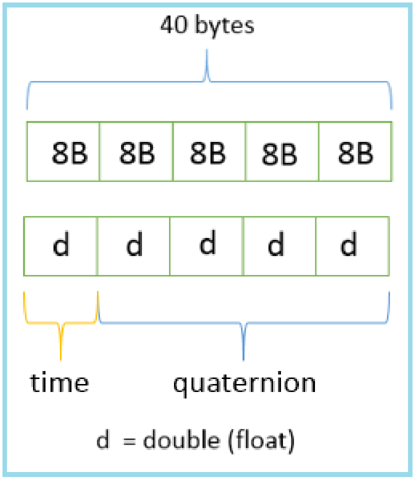


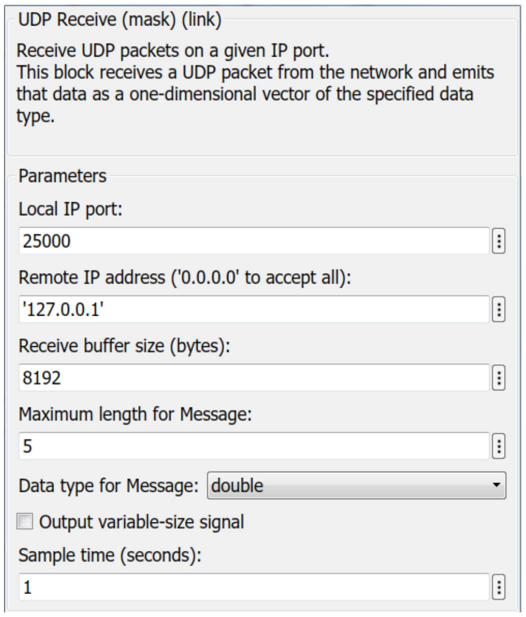
Figure 3: Data packet structure

3.2 Python-Simulink Bridge Settings

The Python-Simulink UDP bridge was developed to enable communication between the BMS and the PSST. The Simulink side of the bridge was originally designed in MATLAB 2010a to match the BMS version. However, it is possible to use this bridge with other MATLAB versions. This subsection describes the settings for both the `send` and the `receive` blocks of the bridge.

3.2.1 The UDP Receive Block

The UDP block to receive data in MATLAB is similar to the UDP receive code in Python. It consists of an IP port (set to 25000, as mentioned earlier in this document), a buffer size (currently set to 8192 bytes), the remote IP address (IP address of the machine where the data is being sent to - if same computer, then the remote IP will be the same for the host and the target files), and the size of the packet being received (here it is the number of items, not the number of bytes. In this case, 8 items - 4 components for the quaternion estimation and 4 components for the error quaternion). See Figure 4(a).


UDP Receive (mask) (link)

Receive UDP packets on a given IP port.
This block receives a UDP packet from the network and emits that data as a one-dimensional vector of the specified data type.

Parameters

Local IP port: 25000

Remote IP address ('0.0.0.0' to accept all): '127.0.0.1'

Receive buffer size (bytes): 8192

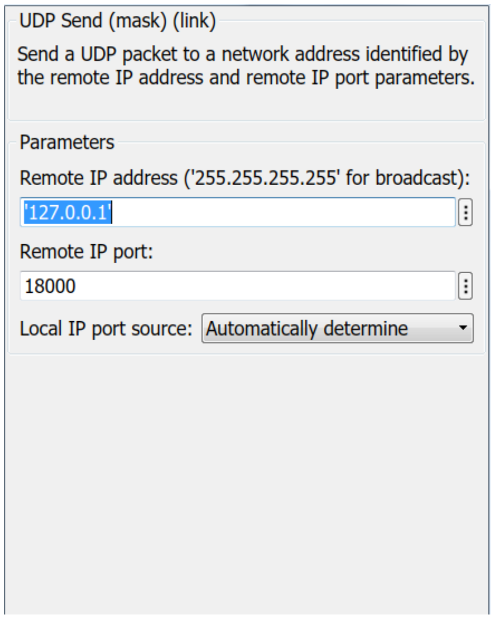
Maximum length for Message: 5

Data type for Message: double

☐ Output variable-size signal

Sample time (seconds): 1

(a) **MATLAB UDP Receive block**


UDP Send (mask) (link)

Send a UDP packet to a network address identified by the remote IP address and remote IP port parameters.

Parameters

Remote IP address ('255.255.255.255' for broadcast): '127.0.0.1'

Remote IP port: 18000

Local IP port source: Automatically determine

(b) **MATLAB UDP Send block**

Figure 4: **UDP *Send* and *Receive* Simulink blocks for illustration.**

3.2.2 The UDP Send block

The UDP Send block is also similar to the UDP send code in Python and is described in Figure 4(b). Its inputs are the IP address of the machine being used and the remote port, which is the local port of the file where the data is being sent to (set to 21000). In this case, the data will be sent to the sensor signal simulation in Python once the system is fully integrated. Currently, the `ST_true_signal.py` file in Python is replacing this part of the code. The UDP Send block in MATLAB is just a communication block that will be integrated with the true attitude trajectory simulation in MATLAB system once it is completed. This block does not required specification of the data size.

3.2.3 The Soft Real Time Block

The *Soft Real Time* block regulates the data flow speed in the simulation. It is currently set to operate at 100 Hz, which is the IMU frequency (highest sensor frequency). The *RealTime* folder needs to be included in the MATLAB path in order for the *Soft Real Time* block to work. It will show as a 'bad link' when MATLAB file if the *RealTime* folder has not been included in the path.

4 Kalman Filter

An indirect-multiplicative extended Kalman filter (imEKF) is being utilized within the PSST. An extended Kalman filter is capable of predicting errors for non-linear systems. In the EKF, the current estimate (i.e., conditional mean) is used for the nominal state estimate, so that $\bar{x}(t) = \hat{x}(t)$ [4]. As a side note, a Kalman filter that estimates the full state of the system is denoted as a *direct* Kalman filter [5].

An indirect EKF gives out pointing estimation errors (the state is the error in the estimated attitude, not the attitude directly). In this imEKF, the state is the error in the Gibbs vector.

In order to get an output that is the estimated quaternion instead of the attitude error, the attitude error needs to be added to the estimated quaternion at the end, which is what is currently being calculated inside the imEKF.

The multiplicative part of the filter means that the attitude is being represented as a quaternion product (see Eq.138 in [3]).

The filter can operate with data files or hardware data, and it is to the discretion of the user which one to choose or how to implement it. Implementation of the data inside the filter will be presented in Section 4.6.

In the current configuration, the imEKF will process the simulated signals from the IMU and ST, received via the UDP bridge (emulating hardware data). The imEKF processes left-hand format quaternions. For this reason, the sensors signal simulation file in Python already reformats the quaternions when forming the data packet to be sent to the imEKF. The output of the imEKF is the estimated attitude pointing quaternion that predicts where the telescope is pointing at. A data packet is then formed to be sent back to the BMS.

4.1 General Equations

The imEKF is primarily an implementation of [3], [6], and [7]. This subsection is an overview of the mathematics of this imEKF for documentation purposes. All equations and associated comments from next paragraph until Subsection 4.5 were extracted from an unpublished document.[4] The imEKF is still under revision for release, but it has been validated against WASP (Wallops Arc Second Pointer) data and results can be found in [8]. The performance of the filter is out of the scope of this document, however, the results of the integration of the Python simulated data with the filter is presented in Section 5.3

The unit quaternion is the primary means for parameterizing the attitude solution. A quaternion giving the orientation of the body frame (B) relative to an inertial frame (I) is defined in terms of its vector and scalar components as a function of the Euler axis of rotation ($\mathbf{e}$) and rotation angle (ϕ):

$$q_{B/I} = \begin{bmatrix} \mathbf{q} \\ q_4 \end{bmatrix} = \begin{bmatrix} \mathbf{e} \cdot sin(\phi/2) \\ cos(\phi/2) \end{bmatrix}$$

A "right-to-left" quaternion multiplication is adopted:

$$p \otimes q = \begin{bmatrix} p_4\mathbf{q} + q_4\mathbf{p} - \mathbf{p} \times \mathbf{q} \\ p_4 q_4 - \mathbf{p} \cdot \mathbf{q} \end{bmatrix}$$

[4]WASP project files/*Development of Extended Kalman Filter for WASP*) by Heatwole and Lanzi, December 19, 2011.

The kinematic equations describing the propagation of body-rate (${}^I\omega^B$) into body attitude are as follows:

$$\dot{q}_{B/I} = \frac{1}{2}\begin{bmatrix} {}^I\omega^B \\ 0 \end{bmatrix} \otimes q_{B/I}$$

The rate sensor is modeled by the following equation:

$$\tilde{\boldsymbol{\omega}} = \boldsymbol{\omega} + \boldsymbol{b}_\omega - \boldsymbol{\nu}_\omega$$

where $\tilde{\boldsymbol{\omega}}$ is the modeled output of the rate sensor, which consists of the true rate ($\boldsymbol{\omega}$), corrupted by a bias term ($\boldsymbol{b}_\omega$) and a Gaussian white noise term ($\boldsymbol{\nu}_\omega$).

The rate estimation is defined by:

$$\hat{\boldsymbol{\omega}} = \tilde{\boldsymbol{\omega}} - \hat{\boldsymbol{b}}_\omega$$

and the rate estimation error is defined by:

$$\delta\boldsymbol{\omega} \equiv \boldsymbol{\omega} - \hat{\boldsymbol{\omega}}$$

Similarly, the bias estimation error is defined as:

$$\delta\boldsymbol{b}_\omega \equiv \boldsymbol{b}_\omega - \hat{\boldsymbol{b}}_\omega$$

4.2 Filter Formulation

The propagation of the state error in discrete form can be written as a matrix:

$$\delta X_k = \Phi_{k-1}\delta X_{k-1} + \Gamma_{k-1}\boldsymbol{\nu}_{k-1}$$

where Φ is the state transition matrix, Γ is the system noise transition matrix, and ν is the discretized system noise. For the EKF it is assumed that the corrected error state is equal to zero after each Kalman update.

The pre-updated state covariance is propagated as:

$$P_k^- = \Phi_k P_{k-1}\Phi_k^\intercal + \Gamma_k Q_{k-1}\Gamma_k^\intercal$$

where P_k^- is the propagated covariance at step k and Q is the system noise covariance matrix.

The residual is calculated upon new measurement:

$$\delta\boldsymbol{z}_k = \boldsymbol{h}_{obs}(X_k) - \boldsymbol{h}(\hat{X}_k)$$

The Kalman gain is calculated:

$$K_k = P_k^- H_k^\intercal [H_k P_k^- H_k^\intercal + R]^{-1}$$

where H_k is the Jacobian of $\boldsymbol{h}(\boldsymbol{x})$, a matrix relating the error states and the measurement residuals, and R is the measurement noise covariance matrix.

The filter state vector is partitioned as follows:

$$\delta \boldsymbol{x} = \begin{bmatrix} \delta \boldsymbol{a} \\ \delta \boldsymbol{b}_\omega \end{bmatrix}$$

where a is the modified Gibb's vector, described as:

$$a(q) = \frac{2\boldsymbol{q}}{q_4}$$

The bias is then updated:

$$\hat{\boldsymbol{b}}^+_{\omega,k} = \hat{\boldsymbol{b}}^-_{\omega,k} + \delta \hat{\boldsymbol{b}}_{\omega,k}$$

The attitude states are updated:

$$q^+_{B/I,k} = q_{B/\hat{B}}(\delta \boldsymbol{a}_k) \otimes q^-_{\hat{B}/I,k}$$

The State Transition Matrix (Φ) is time variant and is obtained by taking the Jacobian of the continuous-time system differential equations:

$$A(t_k) = \begin{bmatrix} -\hat{\boldsymbol{\omega}}(t)\times & -I_3 \\ 0 & 0 \end{bmatrix}$$

where $\omega(t)\times$ is the skew-symmetric matrix operator. The State Transition matrix (Φ) is given by:

$$\Phi(t_k, t_{k-1}) \equiv \Phi_{k,k-1} = e^{A\Delta t} = I + A\Delta t + \frac{(A\Delta t)^2}{2!} = \frac{(A\Delta t)^3}{3!} + ...$$

which can be approximated to second order:

$$\Phi_{k,k-1} = \begin{bmatrix} I_3 - (\hat{\boldsymbol{\omega}}_k) \times \Delta t & -\Delta t \cdot I_3 \\ 0 & I_3 \end{bmatrix}$$

The process noise matrix (Q) is given by:

$$Q_k = \begin{bmatrix} (\sigma^2_\omega \tau \cdot I_3 + (\hat{\boldsymbol{\omega}}\times)(\hat{\boldsymbol{\omega}}\times)^\intercal \sigma^2_\omega \frac{\tau^3}{3} + \sigma^2_b \frac{\tau^3}{3} \cdot I_3) & -\sigma^2_b \frac{\tau^2}{2} \cdot I_3 \\ -\sigma^2_b \frac{\tau^2}{2} \cdot I_3 & \sigma^2_b \tau \cdot I_3 \end{bmatrix}$$

where τ is the time interval between updates and σ^2_i are the respective variances. The $\hat{\boldsymbol{\omega}}\times$ term is not used in the Kalman filter presented in this work.

4.3 Attitude Propagation

The attitude is propagated via the corrected $\Delta\hat{\theta}$ measurements from the IMU:

$$\Delta\hat{\boldsymbol{\theta}}_k = \Delta\tilde{\boldsymbol{\theta}}_k - \hat{\boldsymbol{b}}_g \Delta t$$

where $\Delta\tilde{\boldsymbol{\theta}}$ is the IMU measurement and $\Delta t = t_k - t_{k-1}$.

To propagate the quaternion estimate one time step:

$$\hat{\boldsymbol{q}}_{B/I,k+1} = \left[\cos\left(\frac{\|\Delta\hat{\theta}_k\|}{2}\right) I_{4\times4} + \frac{1}{\|\Delta\hat{\theta}_k\|}\sin\left(\frac{\|\Delta\hat{\theta}_k\|}{2}\right)\Delta\Theta_k\right]\hat{\boldsymbol{q}}_{B/I,k}$$

where $\Delta\Theta_k$ is given by:

$$\Delta\Theta_k = \begin{bmatrix} 0 & \Delta\theta_k(3) & -\Delta\theta_k(2) & \Delta\theta_k(1) \\ -\Delta\theta_k(3) & 0 & \Delta\theta_k(1) & \Delta\theta_k(2) \\ \Delta\theta_k(2) & -\Delta\theta_k(1) & 0 & \Delta\theta_k(3) \\ -\Delta\theta_k(1) & -\Delta\theta_k(2) & -\Delta\theta_k(3) & 0 \end{bmatrix}$$

4.4 Covariance Propagation

The covariance can be propagated at slower rate than the attitude and the rate of IMU output if the attitude is slowly changing. It is assumed that ω is small and the attitude is slowly changing over the elapsed time:

$$P^-_{k+1} = \Phi_{k+1,k} P_k \Phi^\intercal_{k+1,k} + Q_{k+1}$$

4.5 State Update and Reset

The estimated covariance at time t is used to find the Kalman gains. The Kalman gains are found as:

$$K_k = P^-_k H^\intercal_k [H_k P^-_k H^\intercal_k + R_k]^{-1}$$

where H and R are defined for the measurement being processed. The correction to the error states at time t is given by:

$$\delta\hat{\boldsymbol{x}}_k = K_k \delta\boldsymbol{z}_k$$

The covariance is then updated by using Joseph's form and enforcing symmetry:

$$P^+_k = (I - K_k H_k) P^-_k (I - K_k H_k)^\intercal + K_k R_k K^\intercal_k$$

$$P_{sym} = \frac{1}{2}(P + P^\intercal)$$

4.6 Sensor Signal Simulation Integration

The imEKF was developed for the Integrated Radio and Optical Communication (IROC) project ([8], [9]). It has the capability of processing either hardware data or a data file. It can also process either delta Euler angles (change in angles) or angle rates IMU inputs. Currently, the filter is set to process delta angles, instead of angle rates, and it is configured to read simulated hardware values from a data file. The system has been setup and tested with real hardware on the IMU side as well. The ST has been simulated.

The Python sensors signal simulation file emulates hardware data, with one packet being sent at a time to the imEKF. Since the filter was set up to work in the 'data file mode,' some minor modifications were required in order to implement the incoming UDP Python data in the imEKF code. Nothing of the imEKF algorithm was modified, other than the type of input data and the way the code is receiving the signal.

The original filter is still under revision and not ready for distribution, however, its results have been validated with flight data from WASP (Wallops Arc Second Pointer) mission.[8] After the implementation of the sensor signal simulation data in the filter, results were similar to the original setup. The filter modifications are detailed in the following subsections.

5 Results

The sensor signal simulation file was tested and validated after integration with the imEKF. The communication bridges in Python and MATLAB 2010a were successful. The communication link was established and validated.

5.1 The Python Setup for the UDP bridge validation:

Table 1 shows a summary of the testing setup.

- `ST_true_signal_UDP.py` file sets remote port (RemPort) to 21000 (sensor signal simulation file).
- The sensor signal simulation file (`sensors_signal_UDP.py`) sets its RemPort to 20000 (the imEKF file) and its local port (LocalPort) to 21000.
- The packet inside the sensor signal simulation file (`dataSend`) packs a list that contains a quaternion, a time elapsed information, the star tracker time stamp, the IMU time stamp, and the IMU output angles, in this order. The packet size is 80 bytes.
- The imEKF file (`attest_ekf_UDP.py`) receives the data packets at port 20000 (LocalPort) and sends out estimation quaternion and error to remote port 25000 (MATLAB simulation file).
- All time information is POSIX time, double precision.
- Time intervals for transmission are set to the IMU frequency (100 Hz).

Python Validation Test Setup				
File name	**Data received**	**Data sent**	**Data size**	**Parameters**
`ST_true_signal_UDP`	none	$[t_{st},\, \bar{q}_t,\, q_t]$	40B	RP = 21000
`sensors_signal_UDP`	$[t_{st},\, \bar{q}_t,\, q_t]$	$[\bar{q}_d,\, q_d,\, dt,\, t_{st},\, t_{imu},\, \phi,\, \theta,\, \psi]$	80B	RP = 20000
				LP = 21000
`attest_ekf_UDP`	$[\bar{q}_d,\, q_d,\, dt,\, t_{st},\, t_{imu},\, \phi,\, \theta,\, \psi]$	$[d\bar{q},\, dq,\, d\phi,\, d\theta,\, d\psi]$	64B	RP = 25000

Table 1: Test conditions for the UDP bridge code validation

where 'RP' and 'LP' stand for *remote* and *local* ports; the subscripts 'st', 't', and 'd' stand for *star tracker*, *true values*, and *dirty* (data with errors and noise); $\bar{q}$

and q represent the vector part of the quaternion and its scalar part, respectively; ϕ, θ, and ψ are the Euler angles *yaw*, *pitch*, and *roll*, respectively; and finally, d_i's are the errors.

5.2 MATLAB Setup for the UDP bridge validation:

- Local port was set to 25000.
- Remote port was set to 21000 to send info to the sensor signal simulation in Python and close the communication loop. Currently, the file `ST_true_signal_UDP.py` is replacing the true attitude trajectory signal coming from MATLAB because the integration with the simulation in MATLAB has not yet been done.
- The *Soft Real Time* block establishes the data flow speed. It is currently set to 100 Hz, which is the IMU frequency.

5.3 Sensors signal simulation validation results

Figure 5 shows the quaternion error converging to [0,0,0,1], in left-hand quaternion format (the scalar term is the last).The total simulation time was 2000 seconds. Figure 6 shows the quaternion errors converted to Euler angles format, which is a more intuitive way of observing the errors. The angles were obtained by using a quaternion-Euler angles transformation matrix. Figure 7 shows that it takes about 3 seconds for the scalar term error to converge to 1, while the other 3 terms take around 40 seconds to converge to zero, which is roughly within the range of 0.1% to 2% of the total simulation time. It was noticed, however, that the settling time varies according to simulation time. The filter tends to give much smaller settling times as the simulation time increases.

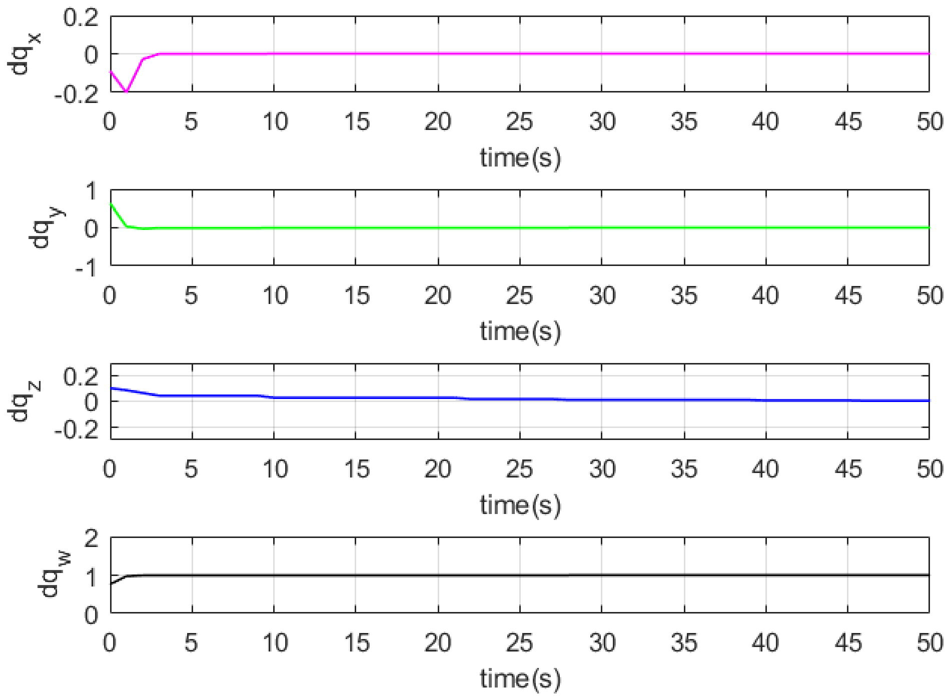


Figure 5: Quaternion error estimation by the imEKF.

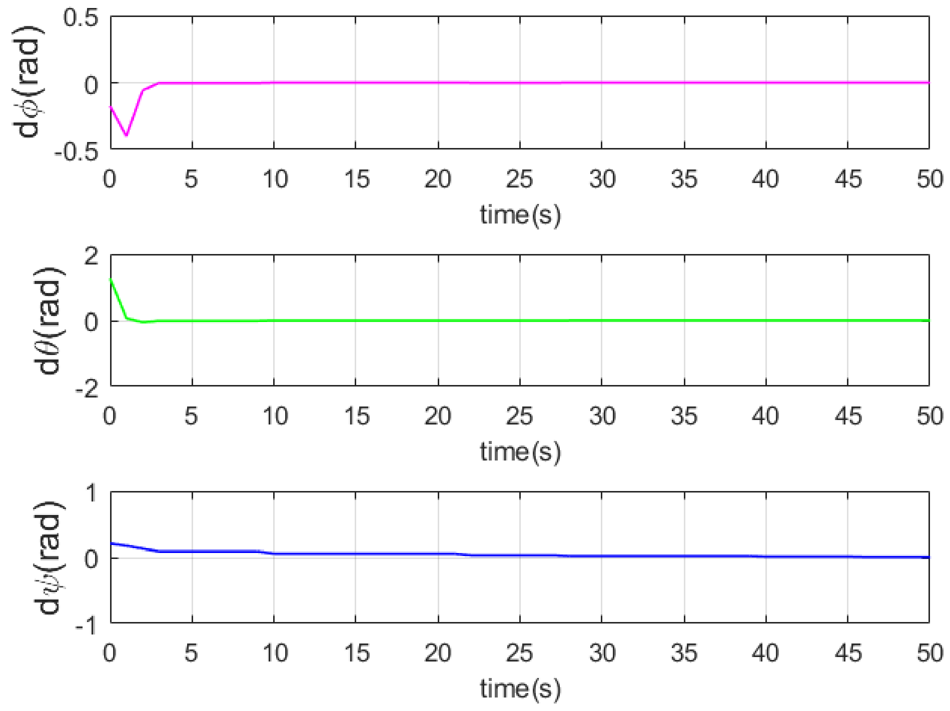


Figure 6: Euler angles error convergence.

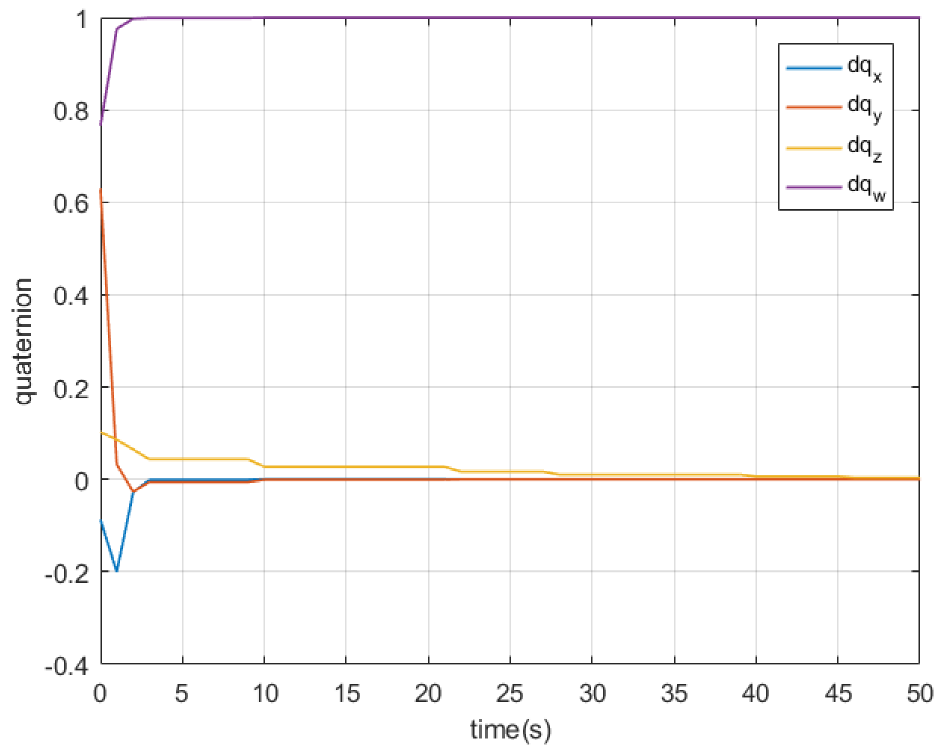


Figure 7: Quaternion error converging time.

6 Conclusion

A Pointing System Simulation Toolbox (PSST) was developed to enable a high-fidelity for a Balloon Mission Simulator (BMS). The PSST consists of a star tracker and inertial measurement unit signal generator, a communication bridge, and an indirect-multiplicative extended Kalman filter (imEKF). Step-by-step instructions on how to operate the PSST were given. The PSST performance was verified to be similar to the standalone imEKF results. The PSST is ready for integration with the BMS.

References

1. Dankanich, John W. et al. “Planetary Balloon-Based Science Platform Evaluation and Program Implementation.” NASA/TM—2016-218870, 2016.

2. Boyle, M. (2017), GitHub repository, https://github.com/moble/quaternion.

3. Trawny N, Roumeliotis SI (2005) “Indirect Kalman Filter for 3D Attitude Estimation.” Technical Report 2005-002, University of Minnesota, Department of Computer Science and Engineering, MARS Lab.

4. Crassidis, J. L., Junkins, J. L. (2012). “Optimal Estimation of Dynamic Systems.” Boca Raton, FL: CRC Press.

5. Maybeck, P. S. (1979). “Stochastic models, estimation, and control.” (Vol. 1). Academic press.

6. Titterton, D., Weston, J. L. (2004). “Strapdown Inertial Navigation Technology.” London, UK: The Institution of Engineering and Technology.

7. Markley, F. L. “Attitude Error Representations for Kalman Filtering.” *Journal of Guidance, control, and dynamics*, Vol.26, No. 2, 2003, pp.282-284.

8. Swank, A. J. et al. “Beaconless Pointing for Deep-Space Optical Communication.” *34th International Communications Satellite Systems Conference*, AIAA, Cleveland, OH, 2016.

9. Raible, D. et al. “On the Physical Realizability of Hybrid RF and Optical Communications Platforms for Deep Space Applications.” *32nd Inernational Communications Satellite Systems Conference*, AIAA, San Diego, CA, 2014.