Flight Computer for Amateur Rocketry

By Ibrahima Barry, ECE '23

Introduction

This paper will explore the canonical Kalman filter algorithm and derive the update and state space equations using mean squared error. Firstly, we begin by providing the necessary mathematical details in probability and linear algebra required to understand the technical details of the algorithm. Then we discuss the algorithm in detail and give some motivation for why one should care about it (Kalman fltering). Finally, we discuss the algorithm, Extended Kalman Filtering, as it used in our senior capstone project. We will also discuss some future work that could be done with the project.

Mathematical Pre-requisites

Mean Squared Error

When we have a signal from a sensor or any other source, we can represent it using the equation (1), where y_k is the observed signal, x_k is the information signal and n_k is the noise signal. The goal is to estimate the information signal.

$$y_k = a_k x_k + n_k \ (1)$$

To evaluate the performance of our estimate, we define the error between the estimated signal and the true signal using equation (2). The error function f_e measures the difference between the estimated signal $\hat{x}_k$ and the true signal x_k.

$$f_e(e_k) = f_e(x_k - \hat{x}_k) \ (2)$$

If we assume that the error function f is a positive, monotonically increasing function, we can use the squared error function (3), which squares the difference between the estimated signal and the true signal.

$$f_e(e_k) = (x_k - \hat{x}_k)^2 \ (3)$$

Finally, we can measure the error over time by taking the expected value of the error function, which is called the loss function L. The loss function L tells us on average how far off our estimated signal is from the true signal.

$$L = E\big(f_e(e_k)\big) \ (4)$$

Maximum Likelihood

In order to find the best estimate of the information signal from a given signal, we can use a technique called maximum likelihood statistics. This means we want to find the filter that maximizes the probability of getting the given signal. Assuming the noise in the signal is distributed in a Gaussian way, we can calculate the probability of getting the signal by using a normalization constant and an exponential formula. The optimal filter is the one that minimizes the mean squared error, which means it provides the best estimate of the information signal.

Assuming Gaussian noise:

$$P\big(Y_k|\hat{X}_k\big) = ke^{-\left(\frac{(y_k - a_k\hat{X})^2}{2\sigma_k^2}\right)} (6)$$

Where K is a normalization constant. The maximum likelihood is given by the product over k.

Derivation Of the Kalman Filter (KF)

We won't give a full derivation via mean squared error here but 6th element of the references goes into the details. Here is a high level description of the algorithm – I will also present some of the results here. The Kalman Filter algorithm is used to estimate the state of a variable based on observations from that variable. The state of the variable is modeled as a linear equation with a noise component. Observations from the variable are made through a second linear equation that also has a noise component. The mean squared error (MSE) of the estimate can be minimized by modeling the noise as a Gaussian distribution. The Kalman Filter uses the MSE to provide an optimal filter. The Kalman Filter algorithm updates the estimate of the state using the Kalman gain and the innovation. The error covariance of the estimate is updated using the Kalman gain and the prediction covariance. The prediction covariance is a function of the measurement noise and the state transition matrix. The Kalman gain is computed using the prior estimate, the prior error covariance, and the prediction covariance. The update equation for the error covariance is a function of the Kalman gain and the prior error covariance. Figure 1. Shows a high-level diagram as well.

The Kalman innovation is as follows:

$$i_k = z_k - H\hat{x}_k \quad (7)$$

Where z_k is true measurement of x at time step k. H is the connection between the state vector x_k and the measurement vector.

The Kalman Gain Equation K is as follows:

$$K_k = P'_k H^T (HP'_k H^T + R)^{-1} \quad (8)$$

Where P is the covariance matrix.

In the Future we would like to explore a fast Kalman filtering algorithm (CKMS recursion) which reduces the computational complexity by using different state propagation equations. In the next section we explore the practical applications of KF via extended Kalman filtering (EKF).

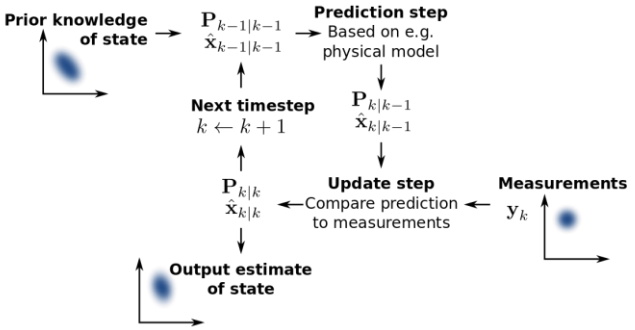

Figure 2. Kalman Filter Flow (img src: https://upload.wikimedia.org/wikipedia/commons/thumb/a/a5/Basic_concept_of_Kalman_filtering.svg/1200px-Basic_concept_of_Kalman_filtering.svg.png)

Kalman Filtering for Our Flight Computer

In figure 2 we give code that implements the extended Kalman filtering (EKF) algorithm for our project. The purpose of the EKF is to estimate the state of the system, which includes the position and velocity vectors, as well as the orientation of the vehicle in space. The EKF takes measurements from the rocket's inertial measurement unit (IMU), which consists of accelerometers and gyroscopes, as well as from external sensors such as GPS and barometers.

The EKF class has two main methods: ***predict*** and ***update.*** The ***predict*** method takes in an IMU reading and the time step since the last reading, and uses this information to predict the new state of the system. The ***update*** method takes in measurements from the GPS and/or barometer and uses them to correct the predicted state.

The EKF class is initialized with the initial state vector x0 and the initial quaternion q0_e2b, which represents the orientation of the rocket. The class also includes methods for initializing the EKF matrices and for computing the measurement update equations.

```
"""

@author: zrummler

PURPOSE: Implements Extended Kalman Filtering for
our Flight Computer

OBJECT: EKF(x, q, P, Q, R, f, F, h, H)
```

```
METHODS: EKF.predict(), EKF.update(z)

SEE BELOW FOR MORE DOCUMENTATION

"""

import scipy

import numpy as np

np.set_printoptions(linewidth=200)

import strapdown as sd

import quaternions as qt

import earth_model as em


class EKF:

    """

    Extended Kalman Filter Implementation

    How to initialize:

        - Initialize 9-element state vector, x

        - Initialize 4-element global quaternion, q_e2b

        - Create a class with ekf = EKF(x, q_e2b)

    How to run:

        while data collection

            accel, gyro, dt = get next IMU reading

            ekf.predict(accel, gyro, dt)

            if gps or barometer ready

                lla = get next GPS data

                ekf.update(lla)

    See predict() and update() for information on running the filter

    """

    def __init__(self, x0, q0_e2b):

        """

        Initializes the EKF object

        Arguments:

            - x0: (9,1) or (9,) initial state vector [r_ecef, v_ecef, roll_error, pitch_error, yaw_error]

            - q0: initial best estimate of quaternion, 4 x 1,

        """

        self.x = x0.flatten()

        self.q_e2b = q0_e2b

        self.P, self.Q = init_ekf_matrices(x0, q0_e2b)


    def predict(self, z_imu, dt):

        """

        EKF state prediction - run this when you have a new IMU reading

        Arguments:

            - z_imu: (6,1) or (6,) IMU reading [accel_x, accel_y, accel_z, gyro_x, gyro_y, gyro_z]

            - dt: time step since last reading, seconds

        Returns:

            - None

        Notes:

            - Requires an initialized EKF object

        """

        # predict state estimate

        self.x, self.q_e2b, phi = f(self.x.flatten(), self.q_e2b, z_imu, dt)

        # predict state covariance

        self.P = phi @ self.P @ phi.T + self.Q

    def update(self, z_gps, z_baro=None, sigma_gps=15, sigma_baro=0.1):

        """
```

```
        EKF measurement update - run this when you have a new GPS or Barometer measurement

        Arguments:

            - z_gps: (3,1) or (3,) gps measurement vector [lat, long, alt]

            - z_baro: (3,1) or (3,) barometer measurement vector [alt1, alt2, alt3]

            - sigma_gps: standard deviation of GPS readings

            - sigma_baro: standard deviation of barmometer readings

        Returns:

            - None

        Notes:

            - Requires an initialized EKF object
        """

        # do not update if no new measurement

        if (z_gps is None) and (z_baro is None):

            return

        # update with both if new measurements from both

        elif (z_gps is not None) and (z_baro is not None):

            # compute nu, H, R for both

            z_gps_ecef = em.lla2ecef(z_gps)

            nu_gps, H_gps, R_gps = get_position_measurement(self.x, z_gps_ecef, sigma_gps)  # GPS measurement

            nu_baro, H_baro, R_baro = get_altitude_measurement(self.x, z_baro, sigma_baro)

            # use vstack and blockdiag to combine nu, H, and R as needed

            nu = np.vstack((nu_gps, nu_baro))

            H = np.vstack((H_gps, H_baro))

            R = scipy.linalg.block_diag(R_gps, R_baro)

        # update with GPS if new measurement from GPS only

        elif z_gps is not None:

            # compute nu, H, R for GPS

            z_gps_ecef = em.lla2ecef(z_gps)

            nu, H, R = get_position_measurement(self.x, z_gps_ecef, sigma_gps)  # GPS measurement



        # update with barometer if new measurement from baroemter only

        elif z_baro is not None:

            # compute nu, H, R for barometer

            nu, H, R = get_altitude_measurement(self.x, z_baro, sigma_baro)

            #raise NotImplementedError('Barometer measurement not yet implemented')



        # generic EKF update equations

        S = H @ self.P @ H.T + R  # innovation covariance

        K = self.P @ H.T @ np.linalg.inv(S)  # Kalman gain

        self.x = self.x.reshape(-1, 1) + K @ nu  # update state vector

        IKH = np.eye(self.x.shape[0]) - K.dot(H)  # intermediate variable

        self.P = IKH.dot(self.P).dot(IKH.T) + K.dot(R).dot(K.T)  # update state covariance (9 x 9)

        self.x = self.x.flatten()  # ensure x is 1D



        # Reset the attitude state.  Move attitude correction from x to q
```

```
        q_error = qt.deltaAngleToDeltaQuat(-self.x[6:9])

        self.q_e2b = qt.quatMultiply(q_error, self.q_e2b).flatten()

        self.x[6:9] = 0  # reset attitude error


def f(x, q_e2b, z_imu, dt):
    """
    This function updates the state vector and global quaternion via IMU strapdown.

    It also updates the state transiction matrix (9 x 9)

    Arguments:

        - x: (9,1) or (9,) state vector, [pos_x, ... vel_x, ... roll_error, ...]

        - q_eqb: (4,1) or (4,) global quaternion [q_scalar, qi, qj, qk]


    Returns:

        - x_new: updated state vector

        - q_new: updated global quaternion

        - phi: (9,9) updated state propagation matrix
    """

    r_ecef, v_ecef = x[0:3], x[3:6]  # extract ECEF states for convenience

    # grab next IMU reading

    accel, gyro = z_imu[0:3], z_imu[3:6]

    dV_b_imu = accel * dt

    dTh_b_imu = gyro * dt

    # Run the IMU strapdown, get predictions including attitude (q_e2b_new)

    r_ecef_new, v_ecef_new, q_e2b_new = sd.strapdown(r_ecef, v_ecef, q_e2b, dV_b_imu, dTh_b_imu, dt)

    # Update state matrix

    x_new = np.concatenate((r_ecef_new, v_ecef_new, np.zeros(r_ecef.shape)))

    # compute linearized state transition matrix

    phi = compute_state_transition_matrix(dt, x, q_e2b, accel, gyro)

    return x_new, q_e2b_new, phi


# Credit: Tyler Klein

def get_altitude_measurement(x, alt_meas: np.ndarray, sigma: float = 5.0):
    """
    Gets an altitude measurement and the accompanying measurement Jacobian. The altitude is expected to be measure in Height Above the Ellipsoid (HAE) which

    may not be the most useful coordinate frame. This was not used in the software and thus was never modified.

    Parameters

    ----------

    x : (N,) ndarray

        state vector

    alt_meas : (M,)

        measured altitude in HAE [m]

    sigma : float

        measurement standard deviation [m] (Default: 5)

    Returns

    -------

    nu : (M,1)

        measurement innovation vector
```

```
    H : (M,N) ndarray
        measurement partial matrix
    R : (M,M)
        measurement variance
    """
    lla = em.ecef2lla(x[0:3])  # convert to LLA in [rad, rad, m (HAE)]
    M = alt_meas.shape[0]
    #print(M)
    H = np.zeros((M, x.shape[0]))  # measurement partial
    # Populate H matrix
    #H[:, 0] = np.cos(lla[1]) * np.cos(lla[0])
    #H[:, 1] = np.sin(lla[1]) * np.cos(lla[0])
    #H[:, 2] = np.sin(lla[0])
    # Populate H matrix
    J = em.lla_jacobian(x[0:3])
    H[:, 0:3] = J[2,:]
    nu = (alt_meas - lla[2]).reshape(M,1)
    R = sigma ** 2 * np.eye(M)
    return nu, H, R



# Credit: Tyler Klein
def get_position_measurement(x, z, sigma=15):
    """
    Gets an absolute position measurement in the ECEF frame
    Parameters
    ----------
    x : (N,) or (N,1) ndarray
        state vector where x[0:3] is the ECEF position in [meters]
    z : (3,) ndarray,
        measured ECEF position [meters]
    sigma : float, default=15
        measurement uncertainty [m] (Default: 15)
    Returns
    -------
    nu : (3,1) ndarray
        measurement innovation vector [meters]
    H : (3,N) ndarray
        measurement partial matrix
    R : (3,3) ndarray
        measurement covariance matrix
    """
    if np.ndim(x) == 2:
        x = x[:, 0]  # reduce dimension
        nu = (z - x[:3]).reshape(3, 1)  # measurement innovation
        R = sigma * sigma * np.eye(3)  # measurement covariance matrix
        H = np.zeros((3, x.shape[0]))
        H[:3, :3] = np.eye(3)
        return nu, H, R


def compute_state_transition_matrix(dt, x, q, accel, gyro):
    """
    This function constructs the 9 x 9 state transition matrix
```

```
    Arguments:
        dt: timestep [seconds]
        x: state vector, 9 x 9, [pos_x, pos_y, pos_x, vel_x, ... ]
        q: best quaternion estimate, 4 x 1, [qs, qi, qj, qk]
        accel: IMU acceleration, 3 x 1, [accel_x, accel_y, accel_z], m/s^2
        gyro: IMU angular rotation, 3 x 1, [gyro_x, gyro_y, gyro_z], rad/sec
    Returns:
        F: a 9 x 9 matrix
    """
    F = np.zeros((9, 9))
    # unpack position
    r_ecef = x[:3]
    # determine rotation matrix and such
    T_b2i = np.linalg.inv(qt.quat2dcm(q))
    # determine cross of omega
    omega_cross = skew(em.omega)
    # Compute each 3 x 3 submatrix ... Credit: Tyler's email
    F[0:3, 3:6] = np.eye(3)  # drdv
    F[3:6, 0:3] = em.grav_gradient(r_ecef) - omega_cross.dot(omega_cross)  # dvdr
    F[3:6, 3:6] = -2 * omega_cross  # dvdv
    F[3:6, 6:9] = -T_b2i.dot(skew(accel))  # dvdo
    F[6:9, 6:9] = -skew(gyro)  # dodo
    F = np.eye(9) + F * dt
    return F
```

```
def skew(M):
    """
    Computes the skew-symmetric matrix of a 3-element vector
    Arguments:
        - M: 3 x 1 vector
    Returns:
        - M x: 3 x 3 skew-symmetric matrix
    """
    return np.cross(np.eye(3), M)


def init_ekf_matrices(x, q):
    """
    Initializes the P, Q, R, and F matrices
    Arguments:
        - x: state vector, 9 x 9, [pos_x, pos_y, pos_z, vel_x, vel_y, vel_z, roll_error, pitch_error, yaw_error]
        - q: best quaternion estimate, 4 x 1, [qs, qi, qj, qk]
    Returns:
        - P: 9 x 9
        - Q: 9 x 9
    """
    # P: predicted covariance matrix, 9 x 9, can be random (reflects initial uncertainty)
    P = np.eye(9) * 0.1  # does not matter what this is
    # Q: process noise matrix, 9 x 9, I * 0.001
    Q = np.eye(9) * 0.001
    return P, Q
```

Figure 2. EKF for flight computer

Conclusion

And we are done with the derivation. In the Future we would like to explore a fast Kalman filtering algorithm (CKMS recursion) which reduces the computational complexity by using different state propagation equations. We would also like to explore the practical applications of KF via extended Kalman filtering (EKF) and unscented Kalman filtering (UKF).

References

1. Jiao, Jiantao. “Lecture 24: CKMS Recursion - University of California, Berkeley.” *Lecture 24: CKMS Recursion*, UC Berkeley, 28 Apr. 2020, https://people.eecs.berkeley.edu/~jiantao/225a2020spring/scribe/EECS225A_Lecture_24.pdf.

2. Becker, Alex. “Online Kalman Filter Tutorial.” *Kalman Filter Tutorial*, https://www.kalmanfilter.net/default.aspx.

3. Lacey, Tony. “Chapter Utorial: The Kalman Filter - Massachusetts Institute of Technology.” Tutorial: The Kalman Filter, MIT, https://web.mit.edu/kirtley/kirtley/binlustuff/literature/control/Kalman%20filter.pdf.

4. “Kalman Filter.” Wikipedia, Wikimedia Foundation, 15 Mar. 2023, https://en.wikipedia.org/wiki/Kalman_filter.

5. Hendeby , Gustaf. “Optimal Filtering 2004 Lecture 8 — Methods.” Fast Algorithms, https://people.isy.liu.se/rt/fredrik/edu/optfilt/methods4.pdf.

6. Barry, Ibrahima. Kalman Filtering Derivation via Mean Squared Error. Tufts University Senior Design, 15 Jan. 2023.

7. source code: https://github.com/barrycoder123/flightCPUforAmatRoc