EKF Planning

References

- Implementation
  - https://www.youtube.com/watch?v=7HVPjkWOrLE
- Theory
  1. https://www.youtube.com/watch?v=RZd6XDx5VXo
  2. https://www.youtube.com/watch?v=BUW2OdAtzBw
  3. https://www.youtube.com/watch?v=hQUkiC5o0JI

  I should follow the order of introduction in the videos above to explain everything in my thesis: 1. Introduce the basics about the sensors themselves. 2. Introduce how attitude can be derived from them. 3. Introduce basic filtering. 4. Introduce complementary filters. 5. Introduce Kalman filtering. 6. Explain implementation details. 7. Explain performance. 8. Explain caveats.

Pre-Processing

1st Order IIR LP Filtering

(done for all sensor measurements to remove high-frequency noise)

$$y_n = \alpha \cdot y_{n-1} + (1-\alpha) \cdot x_n f_c = \frac{1}{2\pi \cdot \alpha \cdot T_s}$$

(a good value for α *is* $0.01 \lessapprox \alpha \lessapprox 0.1$*, with lower* α *for higher* f *measurement)*

Consider if we also want to LP filter the orientation quaternion measurement.

IMU Calibration

Basic approach is to take offset measurements when not moving, then:

$$x_{\text{calib}} = x_{\text{raw}} + c_{\text{bias}}$$

This c_{bias} should be an average of many static measurements taken over time.

If we have a rig that can max out the range of the accelerometer; then we may do a full linear calibration:

$$x_{\text{calib}} = m_{\text{scale}} \cdot x_{\text{raw}} + c_{\text{bias}}$$

For our application we can apply this calibration to the IMU linear acceleration measurements and potentially the intial orientation quaternion if the system starts at a 0 orientation.

Frame Transformations

Before we pass any measurements from sensors to our EKF, we need to transform them into the global frame. We can do this through quaternions and rotation matrices given from the orientation quaternion output (IMU does its own 9DOF sensor fusion to output attitude quaternion).

It's important to double check all of the orientations are correctly transformed to the global frame before debugging EKF output.

Variables

State:

$$\underline{\hat{x}}_n = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

(estimated values of the system's state, at a given time.)

Error Covariance:

(uncertainty in the current state prediction)

$$P = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_z^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{v_x}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{v_y}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{v_z}^2 \end{bmatrix} \Rightarrow \begin{bmatrix} \sigma_x^2 \\ \sigma_y^2 \\ \sigma_z^2 \\ \sigma_{v_x}^2 \\ \sigma_{v_y}^2 \\ \sigma_{v_z}^2 \end{bmatrix}$$

(assuming that there is no noise-correlation, all diagonals becomes a 1x6 vector) (typically small non-zero elements, 0.1-0.01 range)

Process Noise Covariance:

(uncertainty in process model, aka. due to IMU noise)

$$Q = \begin{bmatrix} \sigma_{a_x}^2 & 0 & 0 \\ 0 & \sigma_{a_y}^2 & 0 \\ 0 & 0 & \sigma_{a_z}^2 \end{bmatrix} \Rightarrow \begin{bmatrix} \sigma_{a_x}^2 \\ \sigma_{a_y}^2 \\ \sigma_{a_z}^2 \end{bmatrix}$$

(assuming that there is no noise-correlation, all diagonals becomes a 1x3 vector)

(this needs to be emperically tuned to make EKF estimates stable and convergent, might want to look at datasheet and measure this value for baseline)

Measurement Noise Covariance:

(uncertainty in measurements, aka. optical flow noise)

$$R = \begin{bmatrix} \sigma_{\Delta_x}^2 & 0 & 0 \\ 0 & \sigma_{\Delta_y}^2 & 0 \\ 0 & 0 & \sigma_{\Delta_z}^2 \end{bmatrix} \Rightarrow \begin{bmatrix} \sigma_{\Delta_x}^2 \\ \sigma_{\Delta_y}^2 \\ \sigma_{\Delta_z}^2 \end{bmatrix}$$

(assuming that there is no noise-correlation, all diagonals becomes a 1x3 vector)

(this needs to be emperically tuned to make EKF estimates stable and convergent, might want to look at datasheet and measure this value for baseline)

Steps

1. **Initialize** $\underline{\hat{x}}_0, P, Q, R$ and T.

   1.1. $\underline{\hat{x}}_0 = \begin{bmatrix} x & y & z & v_x & v_y & v_z \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$

   2.2. $P = \begin{bmatrix} \sigma_x^2 & \sigma_y^2 & \sigma_z^2 & \sigma_{v_x}^2 & \sigma_{v_y}^2 & \sigma_{v_z}^2 \end{bmatrix}^T = \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \end{bmatrix}^T$

   2.3. $Q = \begin{bmatrix} \sigma_{a_x}^2 & \sigma_{a_y}^2 & \sigma_{a_z}^2 \end{bmatrix}^T = \begin{bmatrix} 0.001 & 0.001 & 0.001 \end{bmatrix}^T$

   2.4. $R = \begin{bmatrix} \sigma_{\Delta_x}^2 & \sigma_{\Delta_y}^2 & \sigma_{\Delta_z}^2 \end{bmatrix}^T = \begin{bmatrix} 0.01 & 0.01 & 0.01 \end{bmatrix}^T$

2. **Predict** from IMU lin. accel. at 100Hz, $\Delta_t = 0.01$s.

   Inputs:

   - Body-frame linear acceleration $\mathbf{a_{imu}}$ (gravity-compensated)
   - Orientation quaternion $\mathbf{q}$

   2.1. Rotate acceleration to global frame

$$\mathbf{a_g} = R(\mathbf{q}) \cdot \mathbf{a_{imu}} = \underline{u}$$

   2.2. Update the current state prediction

$$\underline{\hat{x}}_{n+1} = \underline{\hat{x}}_n + \Delta_t \cdot f(\underline{\hat{x}}_n, \underline{u}, \Delta_t) \because f = \begin{bmatrix} v_x + \frac{1}{2}\underline{u}_x\Delta_t \\ v_y + \frac{1}{2}\underline{u}_y\Delta_t \\ v_z + \frac{1}{2}\underline{u}_z\Delta_t \\ \underline{u}_x \\ \underline{u}_y \\ \underline{u}_z \end{bmatrix} \therefore \underline{\hat{x}}_{n+1} = \begin{bmatrix} x + v_x \cdot \Delta_t + \frac{1}{2}\underline{u}_x \cdot \Delta_t^2 \\ y + v_y \cdot \Delta_t + \frac{1}{2}\underline{u}_y \cdot \Delta_t^2 \\ z + v_z \cdot \Delta_t + \frac{1}{2}\underline{u}_z \cdot \Delta_t^2 \\ v_x + \underline{u}_x \cdot \Delta_t \\ v_y + \underline{u}_y \cdot \Delta_t \\ v_z + \underline{u}_z \cdot \Delta_t \end{bmatrix}$$

(updating based on Euler integration of acceleration, aka. dead-reckoning)

   2.3. Update the error covariance

$$P_{n+1} = P_n + \Delta_t \cdot (A \cdot P_n + P_n \cdot A^T + Q)$$

(where A is the Jacobian of $f(\underline{\hat{x}}_n, \underline{u}, \Delta_t)$ or in other words:)

$$A = \text{Jacobian}(f, \underline{\hat{x}}) = \frac{\partial f(\underline{\hat{x}}_n, \underline{u}, \Delta_t)}{\partial \underline{\hat{x}}_n} = \begin{bmatrix} \frac{\partial \dot{x}}{\partial x} & \frac{\partial \dot{x}}{\partial y} & \frac{\partial \dot{x}}{\partial z} & \frac{\partial \dot{x}}{\partial v_x} & \frac{\partial \dot{x}}{\partial v_x} & \frac{\partial \dot{x}}{\partial v_z} \\ \frac{\partial \dot{y}}{\partial x} & \frac{\partial \dot{y}}{\partial y} & \frac{\partial \dot{y}}{\partial z} & \frac{\partial \dot{y}}{\partial v_x} & \frac{\partial \dot{y}}{\partial v_x} & \frac{\partial \dot{y}}{\partial v_z} \\ \frac{\partial \dot{z}}{\partial x} & \frac{\partial \dot{z}}{\partial y} & \frac{\partial \dot{z}}{\partial z} & \frac{\partial \dot{z}}{\partial v_x} & \frac{\partial \dot{z}}{\partial v_x} & \frac{\partial \dot{z}}{\partial v_z} \\ \frac{\partial \dot{v_x}}{\partial x} & \frac{\partial \dot{v_x}}{\partial y} & \frac{\partial \dot{v_x}}{\partial z} & \frac{\partial \dot{v_x}}{\partial v_x} & \frac{\partial \dot{v_x}}{\partial v_x} & \frac{\partial \dot{v_x}}{\partial v_z} \\ \frac{\partial \dot{v_y}}{\partial x} & \frac{\partial \dot{v_y}}{\partial y} & \frac{\partial \dot{v_y}}{\partial z} & \frac{\partial \dot{v_y}}{\partial v_x} & \frac{\partial \dot{v_y}}{\partial v_x} & \frac{\partial \dot{v_y}}{\partial v_z} \\ \frac{\partial \dot{v_z}}{\partial x} & \frac{\partial \dot{v_z}}{\partial y} & \frac{\partial \dot{v_z}}{\partial z} & \frac{\partial \dot{v_z}}{\partial v_x} & \frac{\partial \dot{v_z}}{\partial v_x} & \frac{\partial \dot{v_z}}{\partial v_z} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0_{3\times3} & I_3 \\ 0_{3\times3} & 0_{3\times3} \end{bmatrix}$$

(the Jacobian can be pre-computed using CAS system and then hardcoded for solve, using syms functions in Octave/MATLAB)

   Pay attention here that also, Q is a 3×3 matrix, but it needs to be projected onto the state dimensions. We need to expand Q using the process model to account for how acceleration noise affects position and velocity. For this we can define a **State Transition Noise Matrix** G:

$$G = \begin{bmatrix} \frac{1}{2}\Delta_t^2 & 0 & 0 \\ 0 & \frac{1}{2}\Delta_t^2 & 0 \\ 0 & 0 & \frac{1}{2}\Delta_t^2 \\ \Delta_t & 0 & 0 \\ 0 & \Delta_t & 0 \\ 0 & 0 & \Delta_t \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\Delta_t^2 \cdot I_3 \\ \Delta_t \cdot I_3 \end{bmatrix}$$

   Then we can perform the expansion:

$$Q_{6\times6} = G \cdot Q \cdot G^T = Q_d$$

   Which turns Q into a 6×6 matrix that we can use; we can call this expanded Q by the name of Q_d.

(P always increases in this step, Q is process noise which adds uncertainty.)

Note here! That Q_d is a full non-diagonal 6×6 matrix. This is because the cross-terms represent correlations between position and velocity uncertainties caused by the same acceleration noise. Therefor, we have to use a full matrix and cannot use a vector representation of diagonal matrix for Q/Q_d.

Since the update step which is supposed to reduce the covariance P only reduces the diagonal terms. This may cause **divergence** on the cross-diagonal terms which goes out of control. There are some methods to fix this issue:

- Option 1: Zero out cross-diagonals, by doing $P = P \cdot I_6$. This ignores the cross-correlation and may cause estimation divergence.
- Option 2: Symmetrization, by symmetrizing P we can ensure it stays positive and semi-definite, as the update step reduces the upper cross-diagonals. We can use that to reduce both. After the update step, we either copy the top cross-diagonal to the bottom cross-diagonal, or we average across the diagonal: $P = \frac{P+P'}{2}$.
- Option 3: Square Root EKF.
  - Instead of working with P directly, we work with S which is such that $P = SS^T$.
  - Initialise with $S_0 = \sqrt{P_0}$.
  - Prediction step formulas become:

$$S_{n+1} = \sqrt{A} \cdot S_n$$

This needs checking

  - Update step formulas remain the same, just replace all P for S_{n+1}.

3. **Update** from OF disp. at 10Hz, $\Delta_t = 0.1$s.

Inputs:

- OF 2D delta-position $\Delta_{x_{OF}}, \Delta_{y_{OF}}$
- Position estimate 10 states ago $\underline{\hat{x}}_{n-9}$
- Orientation quaternion **q**

3.1. Compute the Kalman gain

$$K = P \cdot C^T \cdot (C \cdot P \cdot C^T + R)^{-1}$$

(for our particular case, K is a 3×6 matrix.)

(in this step we need to calculate the determinant of the matrix in brackets first in order to calculate the inverse.)

(where C is the Jacobian of $h(\underline{\hat{x}}_n, \underline{\hat{x}}_{n-9})$ or in other words:)

$$C = \text{Jacobian}(h, \underline{\hat{x}}) = \frac{\partial h(\underline{\hat{x}}_n, \underline{\hat{x}}_{n-9})}{\partial \underline{\hat{x}}_n} = \begin{bmatrix} \frac{\partial \dot{x}}{\partial x} & \frac{\partial \dot{x}}{\partial y} & \frac{\partial \dot{x}}{\partial z} & \frac{\partial \dot{x}}{\partial v_x} & \frac{\partial \dot{x}}{\partial v_x} & \frac{\partial \dot{x}}{\partial v_z} \\ \frac{\partial \dot{y}}{\partial x} & \frac{\partial \dot{y}}{\partial y} & \frac{\partial \dot{y}}{\partial z} & \frac{\partial \dot{y}}{\partial v_x} & \frac{\partial \dot{y}}{\partial v_x} & \frac{\partial \dot{y}}{\partial v_z} \\ \frac{\partial \dot{z}}{\partial x} & \frac{\partial \dot{z}}{\partial y} & \frac{\partial \dot{z}}{\partial z} & \frac{\partial \dot{z}}{\partial v_x} & \frac{\partial \dot{z}}{\partial v_x} & \frac{\partial \dot{z}}{\partial v_z} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} I_3 & 0_{3\times 3} \end{bmatrix}$$

3.2. Update the current state

$$\underline{\hat{x}}_{n+1} = \underline{\hat{x}}_n + K \cdot (\underline{y} - h(\underline{\hat{x}}_n, \underline{\hat{x}}_{n-9}))$$

(where $\underline{y}$ is the current position delta measurement (from OF), and h is the prediction of the current position delta (from $\underline{\hat{x}}_n$, $\underline{\hat{x}}_{n-9}$); therefore $(\underline{y} - h)$ forms the innovation.)

$$\underline{y} = R(\mathbf{q}) \cdot \begin{bmatrix} \Delta_{x_{OF}} \\ \Delta_{y_{OF}} \\ 0 \end{bmatrix} = \begin{bmatrix} \Delta_{x_{OF}} \\ \Delta_{y_{OF}} \\ \Delta_{z_{OF}} \end{bmatrix}$$

(the optical flow measurements are projected into 3D using the current orientation (camera frame to global frame))

$$h(\underline{\hat{x}}_n, \underline{\hat{x}}_{n-9}) = \underline{\hat{x}}_n - \underline{\hat{x}}_{n-9} = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} - \begin{bmatrix} x_{n-9} \\ y_{n-9} \\ z_{n-9} \end{bmatrix} = \begin{bmatrix} x_n - x_{n-9} \\ y_n - y_{n-9} \\ z_n - z_{n-9} \end{bmatrix} = \begin{bmatrix} \underline{\Delta_x} \\ \underline{\Delta_y} \\ \underline{\Delta_z} \end{bmatrix}$$

(this delta prediction is taken by keeping a buffer of the last 10 position values, so that we can easily take a delta from the states' history)

(here we need to check the dimensions of K and be sure that in this step, we only update the position in the state $\underline{\hat{x}}_{n+1}$ and not the velocities; since OF gives us calculated average velocity, but not instantaneous velocity like the IMU, which is the only thing that makes sense to put into the current state vector)

3.3. Update the error covariance

$$P_{n+1} = (I - K \cdot C) \cdot P_n$$

(where I is the identity matrix (I_6 in this case))

(and C is the Jacobian of h, as computed in the Kalman gain step)

(P always decreases in this step.)

> We might have an issue here as P from the prediction step is *non-diagonal* due to the acceleration noice cross-corelation between the position and acceleration. This means that the cross-diagonal terms never reduce, as this P update step can only reduce the covariance along the diagonal; and further more only on the diagonal x,y,z as OF does not input velocity data. See the highlight above for some possible solutions.

Implementation Draft

```
class KalmanFilter {
    public:
        // Constructor, Initialise (1)
        KalmanFilter(float P_init, float Q_init, float R_init, float alpha_imu, float alpha_of);
        // Predict step (2)
        void Predict(float a_imu[3], float orin_q[4]);
        // Update step (3)
        void Update(float of_delta[2], float orin_q[4]);

        // State
        float x[6];

    private:
        // IIR filter alphas
        const float alpha_imu;
        const float alpha_of;
        // IIR filters
        IIR1stOrderVecF imuF(alpha_imu, 3);
        IIR1stOrderVecF ofF(alpha_of, 2);
        //? see if we need to add a filter for orin_q

        // State history buffer for positions only
        float x_pos_hist[10][3];

        // Error covariance matrix
```

```
        float P[6][6];
        // Process noise covariance matrix
        float Q[3][3];
        float G[6][3];
        float Qd[6][6];
        // Measurement noise covariance matrix
        float R[3][3];

        // Kalman gain
        float K[6][3];
        // Jacobian of f
        const float A[6][6];
        // Jacobian of h
        const float C[3][6];
};

class IIR1stOrderVecF {
    public:
        // Constructor
        IIR1stOrderVecF(float alpha, size_t dim):alpha(alpha),dim(dim) {
            n_prev = make(float, dim);
        };
        // Process
        float filter(float* n) {
            if (!first) {
                for (size_t i=0; i<dim; i++) {
                    n_prev[i] = alpha*n_prev[i] + (1-alpha)*n[i];
                }
            } else {
                first = false;
                for (size_t i=0; i<dim; i++) {
                    n_prev[i] = n[i];
                }
            }
            return n_prev;
        }

        // Change alpha
        void setAlpha(float a) {
            alpha = a;
        }

        // Get cutoff f. util
        float getCFreq(float Ts) {
            return float(1)/(2*math.Pi*alpha*Ts);
        }

    private:
        // Vector dimensions
        size_t dim;

        // Mixing alpha
        float alpha;
        // Last state
```

```
        float* n_prev;

        // First input flag
        bool first = true;
};

class Quaternion {
    public:
        // State
        float x, y, z, w;

        // Constructor
        Quaternion(float x,y,z,w):x(x),y(y),z(z),w(w){};
        Quaternion(float i_x,i_y,i_z) {
            fromEuler(i_x,i_y,i_z, &x,&y,&z,&w);
        }

        // Rotation
        void rotate(Quaternion* r, Quaternion* o_r) {
            // Rotate quaternion r by this quaternion
            // and output to o_r

        }
        void rotate(float vec[3], float* o_vec) {
            // Rotate vector vec by this quaternion
            // and output to o_vec as vector

        }

        void rotationMatrix(float* r) {
            // Return this quaternion
            // as a 3x3 rotation matrix

        }

        void rotateSelf(Quaternion* r) {
            // Rotate this quaternion by
            // another quaternion
            // (quaternion multiply)

        }

        void rotateSelf(float vec[3]) {
            // Rotate this quaternion by
            // a vector
            // (quaternion multiply)
            Quaternion rot(vec[0],vec[1],vec[2]);
            rotateSelf(&rot);
        }

        // Normalise
        void normalise() {
            // Normalise self to unit quaternion
```

```
        }

        // Conversion
        void toEuler(float* x,y,z) {
            // Convert this quaternion to euler vec
            // quaternion -> euler

        }

    private:
        //

        void fromEuler(float x,y,z, float* o_x,o_y,o_z, o_w) {
            // euler -> quaternion

        }
};
```