

Lacal_position_estimator

1. 运行过程是什么？

看程序之前先要大概理解卡尔曼算法的原理和实现过程

预测值有高斯噪声，测量值也有高斯噪声，这2个噪声相互独立，单独的利用任何一个都不能很好的得到真实值，所以在2者之间有个信赖度的问题，应该相信谁更多些，这也就是卡尔曼算法的核心，这个信赖度就是卡尔曼增益。

卡尔曼增益通过测量值和真实值之间的协方差最小时确定的，由此求这个协方差偏导为0时的系数，这个系数就是卡尔曼增益，这样就能很好的融合预测值和测量值。

并且推导卡尔曼增益的时候，发现协方差是可以递归的，由此只要刚开始指定初始协方差就可以源源不断的求出卡尔曼增益和新的协方差，从而不断的跟新真实值。

首先要计算预测值、 $P(k|k-1)$ 协方差矩阵。

$$\begin{aligned}\hat{x}_k^- &= A\hat{x}_{k-1} + Bu_{k-1} \\ P_k^- &= AP_{k-1}A^T + Q\end{aligned}$$

有了这两个就能计算卡尔曼增益K，再然后得到估计值

$$\begin{aligned}K_k &= P_k^- H^T (HP_k^- H^T + R)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)\end{aligned}$$

最后还要计算 $P(k|k)$ 协方差矩阵，为下次递推做准备。

$$P_k = (I - K_k H)P_k^-$$

```
int local_position_estimator_thread_main(int argc, char *argv[])
```

```
{
    warnx("starting");
    using namespace control;
    BlockLocalPositionEstimator est;//构造函数
    thread_running = true;
    while (!thread_should_exit) { //不断循环
        est.update();//数据计算更新
    }
    warnx("exiting.");
    thread_running = false;
    return 0;
}
```

进入 BlockLocalPositionEstimator est;构造函数完成矩阵初始化(构造函数是赋初始值的函数)

```
BlockLocalPositionEstimator::BlockLocalPositionEstimator() :
```

```
.....
    _x(), _u(), _P()
```

```

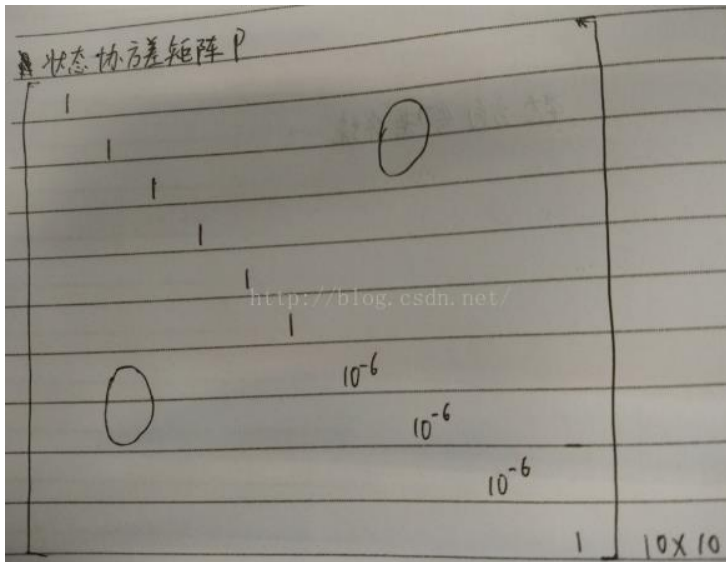
{
    .....
    // initialize P, x, u
    initP();
    _x.setZero();
    _u.setZero();
    _flowX = 0;
    _flowY = 0;
    .....
    updateParams();
}
进入 initP();

```

```

1. Matrix<float, n_x, n_x> _P; // state covariance matrix
2.
3. enum {X_x = 0, X_y, X_z, X_vx, X_vy, X_vz, X_bx, X_by, X_bz, X_tz, n_x};
4. enum {U_ax = 0, U_ay, U_az, n_u};
5. enum {Y_baro_z = 0, n_y_baro};
6. enum {Y_lidar_z = 0, n_y_lidar};
7. enum {Y_flow_x = 0, Y_flow_y, n_y_flow};
8. enum {Y_sonar_z = 0, n_y_sonar};
9. enum {Y_gps_x = 0, Y_gps_y, Y_gps_z, Y_gps_vx, Y_gps_vy, Y_gps_vz, n_y_gps};
10. enum {Y_vision_x = 0, Y_vision_y, Y_vision_z, n_y_vision};
11. enum {Y_mocap_x = 0, Y_mocap_y, Y_mocap_z, n_y_mocap};
12. enum {POLL_FLOW, POLL_SENSORS, POLL_PARAM, n_poll};
13.
14. void BlockLocalPositionEstimator::initP()
15. {
16.     _P.setZero();
17.     _P(X_x, X_x) = 1;
18.     _P(X_y, X_y) = 1;
19.     _P(X_z, X_z) = 1;
20.     _P(X_vx, X_vx) = 1;
21.     _P(X_vy, X_vy) = 1;
22.     _P(X_vz, X_vz) = 1;
23.     _P(X_bx, X_bx) = 1e-6;
24.     _P(X_by, X_by) = 1e-6;
25.     _P(X_bz, X_bz) = 1e-6;
26.     _P(X_tz, X_tz) = 1;
27. }

```



进入循环 `est.update()`

```
void BlockLocalPositionEstimator::update()
{
    .....
    // get new data
    updateSubscriptions();//获取传感器数据
    .....
    predict();//一步预测，也就是模型预测
    if (gpsUpdated) {
        if (!_gpsInitialized) {
            gpsInit();
        } else {
            gpsCorrect();//修正函数
        }
    }
    if (baroUpdated) {
        if (!_baroInitialized) {
            baroInit();
        } else {
            baroCorrect();//修正函数
        }
    }
    if (lidarUpdated) {
        if (!_lidarInitialized) {
            lidarInit();

        } else {
            lidarCorrect();//修正函数
        }
    }
    if (sonarUpdated) {
        if (!_sonarInitialized) {
            sonarInit();
        } else {
            sonarCorrect();//修正函数
        }
    }
}
```

```

    }
}
if (flowUpdated) {
    if (!_flowInitialized) {
        flowInit();
    } else {
        perf_begin(_loop_perf); // TODO
        flowCorrect(); // 矫正函数
        //perf_count(_interval_perf);
        perf_end(_loop_perf);
    }
}
if (visionUpdated) {
    if (!_visionInitialized) {
        visionInit();
    } else {
        visionCorrect(); // 矫正函数
    }
}
if (mocapUpdated) {
    if (!_mocapInitialized) {
        mocapInit();
    } else {
        mocapCorrect(); // 矫正函数
    }
}
if (_altHomeInitialized) {
    // update all publications if possible
    publishLocalPos(); // 发布主题
    publishEstimatorStatus();
    if (_canEstimateXY) {
        publishGlobalPos();
    }
}
.....
}
进入 updateSubscriptions();

```

```

1. virtual void updateSubscriptions()
2. {
3.     Block::updateSubscriptions();
4.     if (getChildren().getHead() != NULL) { updateChildSubscriptions(); }
5. }

```

进入 Block::updateSubscriptions();

```

1. void Block::updateSubscriptions()
2. {
3.     uORB::SubscriptionNode *sub = getSubscriptions().getHead();

```

```

4.     int count = 0;
5.     while (sub != NULL) {
6.         if (count++ > maxSubscriptionsPerBlock) {
7.             char name[blockNameLengthMax];
8.             getName(name, blockNameLengthMax);
9.             printf("exceeded max subscriptions for block: %s\n", name);
10.            break;
11.        }
12.        sub->update();
13.        sub = sub->getSibling();
14.    }
15. }

```

进入 sub->update();

这里只会跟到 `Firmware/src/modules/uORB/Subscription.hpp` virtual void update() = 0;然后在相对应的.cpp 里面找

```

1. void SubscriptionBase::update(void *data)
2. {
3.     if (updated()) {
4.         int ret = orb_copy(_meta, _handle, data);
5.         if (ret != PX4_OK) { warnx("orb copy failed"); }
6.     }
7. }

```

于是通过 while (sub != NULL) {}循环，将订阅的主题都 copy 下来了

接下来要回到 est.update()大循环中查看 predict()

```

void BlockLocalPositionEstimator::predict()
{
    // if can't update anything, don't propagate
    // state or covariance
    if (!_canEstimateXY && !_canEstimateZ) { return; }
    //输入向量_u
    if (_integrate.get() && _sub_att.get().R_valid) {
        Matrix3f R_att(_sub_att.get().R);
        Vector3f a(_sub_sensor.get().accelerometer_m_s2);
        _u = R_att * a;
        _u(U_az) += 9.81f; // add g
    } else {
        _u = Vector3f(0, 0, 0);
    }
}

```

输入向量 u

$$\begin{bmatrix} \quad \end{bmatrix}_{3 \times 3} \cdot \begin{bmatrix} \quad \end{bmatrix}_{1 \times 3} + \begin{bmatrix} 0 \\ 0 \\ 9.81 \end{bmatrix} = \begin{bmatrix} \quad \end{bmatrix}_{1 \times 3} + 9.81$$

旋转矩阵 R accelerometer - m - sz 向量

vehicle_attitude_s sensor_combined_s

//动态矩阵 A

```
// dynamics matrix
Matrix<float, n_x, n_x> A; // state dynamics matrix
A.setZero();
// derivative of position is velocity
A(X_x, X_vx) = 1;
A(X_y, X_vy) = 1;
A(X_z, X_vz) = 1;
// derivative of velocity is accelerometer acceleration
// (in input matrix) - bias (in body frame)
Matrix3f R_att(_sub_att.get().R);
A(X_vx, X_bx) = -R_att(0, 0);
A(X_vx, X_by) = -R_att(0, 1);
A(X_vx, X_bz) = -R_att(0, 2);
A(X_vy, X_bx) = -R_att(1, 0);
A(X_vy, X_by) = -R_att(1, 1);
A(X_vy, X_bz) = -R_att(1, 2);
A(X_vz, X_bx) = -R_att(2, 0);
A(X_vz, X_by) = -R_att(2, 1);
A(X_vz, X_bz) = -R_att(2, 2);
```

动态矩阵 A

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -R_{att}(0,0) & -R_{att}(0,1) & -R_{att}(0,2) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -R_{att}(1,0) & -R_{att}(1,1) & -R_{att}(1,2) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -R_{att}(2,0) & -R_{att}(2,1) & -R_{att}(2,2) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{10 \times 10}$$

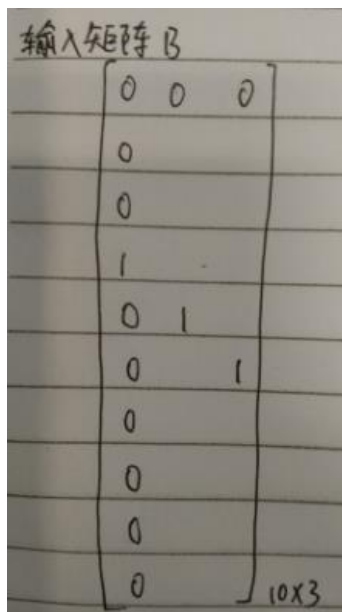
//输入矩阵 B

```
// input matrix
Matrix<float, n_x, n_u> B; // input matrix
```

```

B.setZero();
B(X_vx, U_ax) = 1;
B(X_vy, U_ay) = 1;
B(X_vz, U_az) = 1;

```



//输入噪声协方差矩阵 R

// input noise covariance matrix

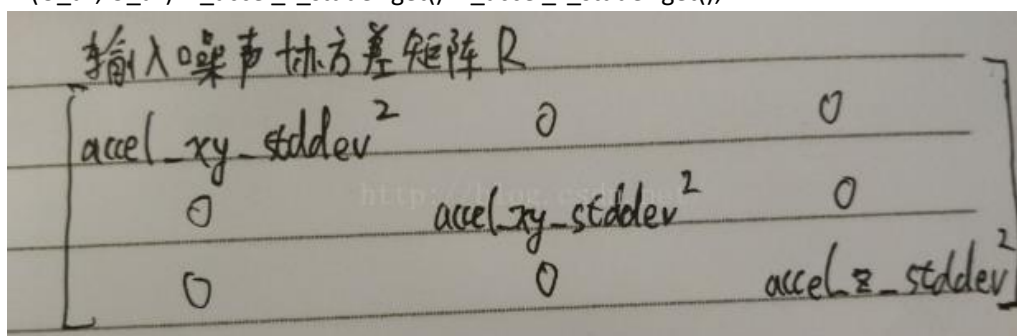
```
Matrix<float, n_u, n_u> R;
```

```
R.setZero();
```

```
R(U_ax, U_ax) = _accel_xy_stddev.get() * _accel_xy_stddev.get();
```

```
R(U_ay, U_ay) = _accel_xy_stddev.get() * _accel_xy_stddev.get();
```

```
R(U_az, U_az) = _accel_z_stddev.get() * _accel_z_stddev.get();
```



//系统过程噪声矩阵 Q

// process noise power matrix

```
Matrix<float, n_x, n_x> Q;
```

```
Q.setZero();
```

```
float pn_p_sq = _pn_p_noise_density.get() * _pn_p_noise_density.get();
```

```
float pn_v_sq = _pn_v_noise_density.get() * _pn_v_noise_density.get();
```

```
Q(X_x, X_x) = pn_p_sq;
```

```
Q(X_y, X_y) = pn_p_sq;
```

```
Q(X_z, X_z) = pn_p_sq;
```

```
Q(X_vx, X_vx) = pn_v_sq;
```

```
Q(X_vy, X_vy) = pn_v_sq;
```

```
Q(X_vz, X_vz) = pn_v_sq;
```

// technically, the noise is in the body frame,

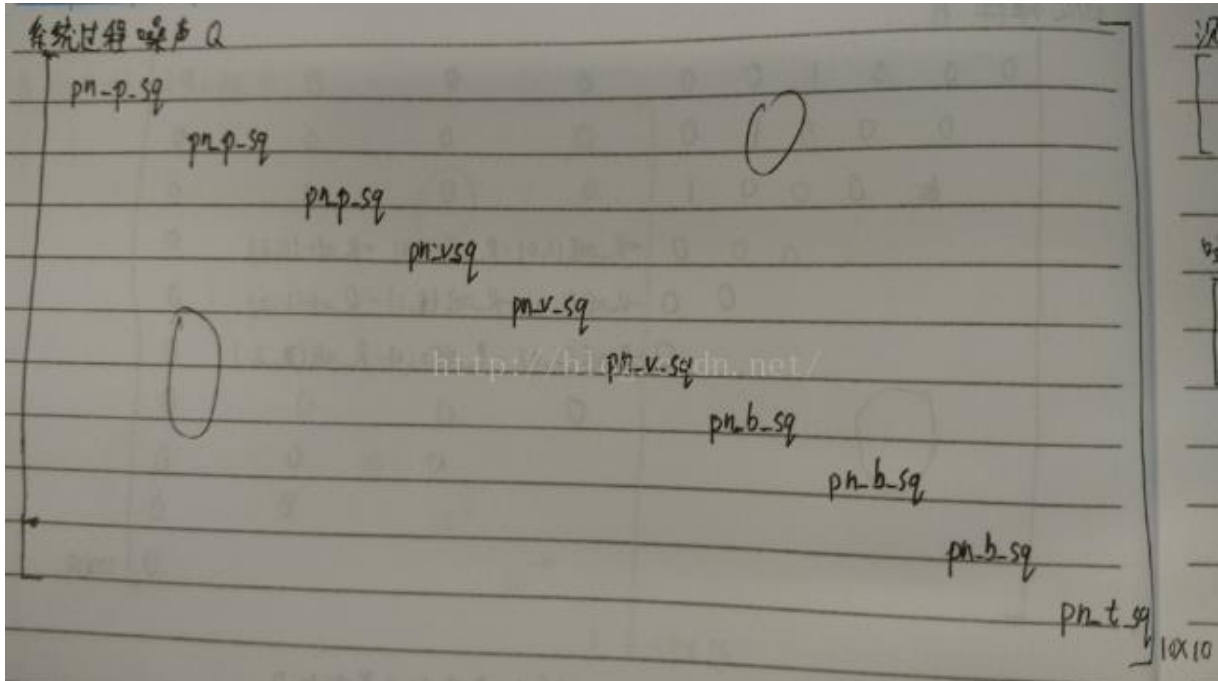
// but the components are all the same, so

// ignoring for now


```

float pn_b_sq = _pn_b_noise_density.get() * _pn_b_noise_density.get();
Q(X_bx, X_bx) = pn_b_sq;
Q(X_by, X_by) = pn_b_sq;
Q(X_bz, X_bz) = pn_b_sq;
// terrain random walk noise
float pn_t_sq = _pn_t_noise_density.get() * _pn_t_noise_density.get();
Q(X_tz, X_tz) = pn_t_sq;

```



//连续时间的卡尔曼滤波器预测值

// continuous time kalman filter prediction

```
Vector<float, n_x> dx = (A * _x + B * _u) * getDt();
```

//上一时刻状态推导这一时刻状态线性系统的状态

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1}$$

.....

//更新-x 状态矩阵, -P 协方差矩阵

// propagate

x += dx;// 这个 dx 就是通过模型推导

P += (A * P + P * A.transpose() +

B * R * B.transpose() + Q) * getDt();

$$P_k^- = AP_{k-1}A^T + Q$$

}

再回到 est.update()大循环中查看 Correct(), 比如 flowCorrect();

```
void BlockLocalPositionEstimator::flowCorrect()
```

```
{
```

```
// measure flow
```

```
Vector<float, n_y_flow> y;
```

```
if (flowMeasure(y) != OK) { return; }//获取光流数据
```

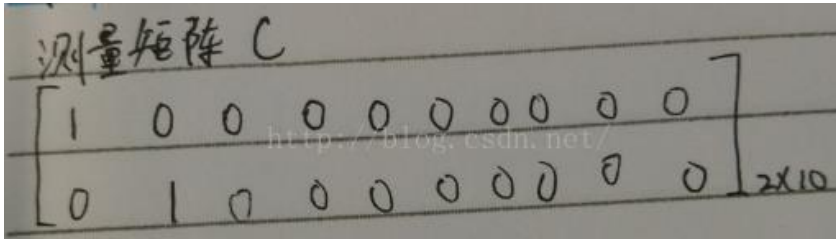
//光流测量矩阵 C

// flow measurement matrix and noise matrix

```
Matrix<float, n_y_flow, n_x> C;
```

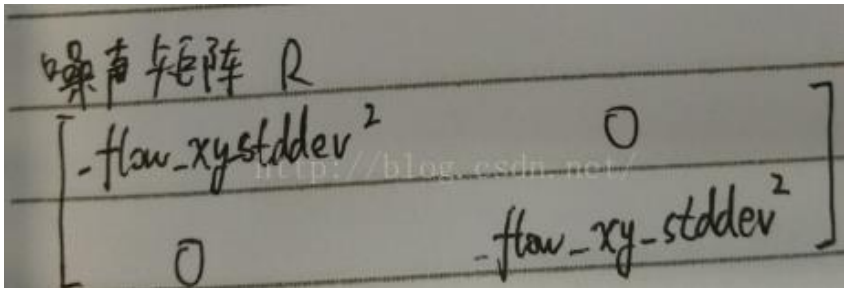


```
C.setZero();
C(Y_flow_x, X_x) = 1;
C(Y_flow_y, X_y) = 1;
```



//噪声矩阵 R

```
Matrix<float, n_y_flow, n_y_flow> R;
R.setZero();
R(Y_flow_x, Y_flow_x) =
    _flow_xy_stddev.get() * _flow_xy_stddev.get();
R(Y_flow_y, Y_flow_y) =
    _flow_xy_stddev.get() * _flow_xy_stddev.get();
```



//剩余向量 r

```
// residual
Vector<float, 2> r = y - C * _x; // _x 来自上面的 sonarCorrect() 等
```

//剩余协方差(逆)

```
// residual covariance, (inverse)
Matrix<float, n_y_flow, n_y_flow> S_I =
    inv<float, n_y_flow>(C * _P * C.transpose() + R); // _P 来自上面的 sonarCorrect() 等
```

//故障检测

```
// fault detection
float beta = (r.transpose() * (S_I * r))(0, 0);
if (beta > BETA_TABLE[n_y_flow]) {
    if (_flowFault < FAULT_MINOR) {
        //mavlink_and_console_log_info(&mavlink_log_pub, "[lpe] flow fault, beta %5.2f", double(beta));
        _flowFault = FAULT_MINOR;
    }
} else if (_flowFault) {
    _flowFault = FAULT_NONE;
    //mavlink_and_console_log_info(&mavlink_log_pub, "[lpe] flow OK");
}
}
```

//光流矫正

```
if (_flowFault < fault_lvl_disable) {
    Matrix<float, n_x, n_y_flow> K =
        _P * C.transpose() * S_I; //卡尔曼增益
    _x += K * r; //更新_x, 状态向量
    // _x = _x(predict 求得) + K(卡尔曼增益) * r(测量与上一状态最优值的误差)
```

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$$

```
//K=_P * C.transpose() * S_1;  
//=_P * C.transpose() * (C * _P * C.transpose() + R)  
//=协方差矩阵*光流测量矩阵的转置*(光流测量矩阵*协方差矩阵*光流测量矩阵的转置+噪声矩阵)
```

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1}$$

```
//r=y - C * _x  
//=光流测量矩阵-测量矩阵*上一状态最优值  
P -= K * C * _P; //更新_P, 状态协方差矩阵  
//_P = _P - K * C * _P  
//    = (I - K * C) * _P
```

$$P_k = (I - K_k H) P_k^-$$

```
} else {  
    // reset flow integral to current estimate of position  
    // if a fault occurred  
    _flowX = _x(X_x);  
    _flowY = _x(X_y);  
}  
}
```

到此卡尔曼算法完成!! 其实卡尔曼算法就是以下 5 个公式不断迭代

首先要计算预测值、 $P(k|k-1)$ 协方差矩阵。

$$\begin{aligned}\hat{x}_k^- &= A\hat{x}_{k-1} + Bu_{k-1} \\ P_k^- &= AP_{k-1}A^T + Q\end{aligned}$$

有了这两个就能计算卡尔曼增益K，再然后得到估计值

$$\begin{aligned}K_k &= P_k^- H^T (H P_k^- H^T + R)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)\end{aligned}$$

最后还要计算 $P(k|k)$ 协方差矩阵，为下次递推做准备。

$$P_k = (I - K_k H) P_k^-$$

相同的，高度/GPS，求法是一样的，只是测量矩阵不一样了，噪声等“个性化”东西不一样，处理过程是一样的。
再回到 `est.update()` 大循环中发布状态

```
1. if (_altHomeInitialized) {  
2.     // update all publications if possible  
3.     publishLocalPos();  
4.     publishEstimatorStatus();
```

```

5.     if (_canEstimateXY) {
6.         publishGlobalPos();
7.     }
8. }

```

publishLocalPos()发布这么多

```

1. _pub_lpos.get().timestamp = _timeStamp;
2. _pub_lpos.get().xy_valid = _canEstimateXY;
3. _pub_lpos.get().z_valid = _canEstimateZ;
4. _pub_lpos.get().v_xy_valid = _canEstimateXY;
5. _pub_lpos.get().v_z_valid = _canEstimateZ;
6. _pub_lpos.get().x = _x(X_x);    // north
7. _pub_lpos.get().y = _x(X_y);    // east
8. _pub_lpos.get().z = _x(X_z);    // down
9. _pub_lpos.get().vx = _x(X_vx);  // north
10. _pub_lpos.get().vy = _x(X_vy); // east
11. _pub_lpos.get().vz = _x(X_vz); // down
12. _pub_lpos.get().yaw = _sub_att.get().yaw;
13. _pub_lpos.get().xy_global = _sub_home.get().timestamp != 0; // need home for reference
14. _pub_lpos.get().z_global = _baroInitialized;
15. _pub_lpos.get().ref_timestamp = _sub_home.get().timestamp;
16. _pub_lpos.get().ref_lat = _map_ref.lat_rad * 180 / M_PI;
17. _pub_lpos.get().ref_lon = _map_ref.lon_rad * 180 / M_PI;
18. _pub_lpos.get().ref_alt = _sub_home.get().alt;
19. _pub_lpos.get().dist_bottom = agl();
20. _pub_lpos.get().dist_bottom_rate = -_x(X_vz);
21. _pub_lpos.get().surface_bottom_timestamp = _timeStamp;
22. _pub_lpos.get().dist_bottom_valid = _canEstimateZ;
23. _pub_lpos.get().eph = sqrtf(_P(X_x, X_x) + _P(X_y, X_y));
24. _pub_lpos.get().epv = sqrtf(_P(X_z, X_z));

```

publishEstimatorStatus()发布这么多

```

1. _pub_est_status.get().timestamp = _timeStamp;
2. for (int i = 0; i < n_x; i++) {
3.     _pub_est_status.get().states[i] = _x(i);
4.     _pub_est_status.get().covariances[i] = _P(i, i);
5. } _pub_est_status.get().n_states = n_x;
6. _pub_est_status.get().nan_flags = 0;
7. _pub_est_status.get().health_flags =
8.     ((_baroFault > fault_lvl_disable) << SENSOR_BARO)
9.     + ((_gpsFault > fault_lvl_disable) << SENSOR_GPS)
10.     + ((_lidarFault > fault_lvl_disable) << SENSOR_LIDAR)
11.     + ((_flowFault > fault_lvl_disable) << SENSOR_FLOW)
12.     + ((_sonarFault > fault_lvl_disable) << SENSOR_SONAR)
13.     + ((_visionFault > fault_lvl_disable) << SENSOR_VISION)
14.     + ((_mocapFault > fault_lvl_disable) << SENSOR_MOCAP);
15. _pub_est_status.get().timeout_flags =

```

```

16.    (_baroInitialized << SENSOR_BARO)
17.    + (_gpsInitialized << SENSOR_GPS)
18.    + (_flowInitialized << SENSOR_FLOW)
19.    + (_lidarInitialized << SENSOR_LIDAR)
20.    + (_sonarInitialized << SENSOR_SONAR)
21.    + (_visionInitialized << SENSOR_VISION)
22.    + (_mocapInitialized << SENSOR_MOCAP);

```

publishGlobalPos()发布这么多

```

1.  _pub_gpos.get().timestamp = _timeStamp;
2.  _pub_gpos.get().time_utc_usec = _sub_gps.get().time_utc_usec;
3.  _pub_gpos.get().lat = lat;
4.  _pub_gpos.get().lon = lon;
5.  _pub_gpos.get().alt = alt;
6.  _pub_gpos.get().vel_n = _x(X_vx);
7.  _pub_gpos.get().vel_e = _x(X_vy);
8.  _pub_gpos.get().vel_d = _x(X_vz);
9.  _pub_gpos.get().yaw = _sub_att.get().yaw;
10. _pub_gpos.get().eph = sqrtf(_P(X_x, X_x) + _P(X_y, X_y));
11. _pub_gpos.get().epv = sqrtf(_P(X_z, X_z));
12. _pub_gpos.get().terrain_alt = _altHome - _x(X_tz);
13. _pub_gpos.get().terrain_alt_valid = _canEstimateT;
14. _pub_gpos.get().dead_reckoning = !_canEstimateXY && !_xyTimeout;
15. _pub_gpos.get().pressure_alt = _sub_sensor.get().baro_alt_meter[0];

```

2. 对比 inav 和 lpe 算法

inav	卡尔曼
(1)用矫正后的加速度一次积分得到预测速度，二次积分得到预测位移 $x[0] += x[1] * dt + acc * dt * dt / 2.0f$; (位移) $x[1] += acc * dt$; (速度)	根据抽象出来的模型和上一状态最优估算值，预测这一状态 $X(k k-1)=AX(k-1 k-1)+BU(k)$
(2)用传感器得到的数据矫正控制需要使用的速度或者位移 $ewdt = e * w * dt$; (e=测量值-最优估算值，w=权值，dt=时间差) $x[i] += ewdt$; $if(i==0)\{x[1] += w * ewdt;\}$ 综合第(1)步 $x[i]$ =位移预测值+(测量值-最优估算值)*权值 $x[0] = x[1] * dt + acc * dt * dt / 2.0f + e * w * dt$ $x[1] += acc * dt + w * e * w * dt$ (传感器是测位移的) $x[1] += acc * dt + e * w * dt$ (传感器是测速度的)	求最优估算值 $X(k k)=X(k k-1)+kg(k)(Z(k)-HX(k k-1))$
(3)权值的求取 直接给定，在参数列表中，对于GPS和flow根据自身的特性会再乘变化的权值 $w_z_gps_p = params.w_z_gps_p * w_gps_z$ $params.w_xy_flow * w_flow$	卡尔曼增益kg的求取 $c += PX4_R(att.R, j, i) * accel_bias_corr[j]$; $acc_bias[i] += c * params.w_acc_bias * dt$; $sensor.accelerometer_m_s2[0] -= acc_bias[0]$; $acc[i] += PX4_R(att.R, i, j) * sensor.accelerometer_m_s2[j]$; 卡尔曼中这步体现在第(1)步中
(4)加速度矫正 $corr_baro = baro_offset - sensor.bar_alt_meter - z_est[0]$; $accel_bias_corr[2] -= corr_baro * params.w_z_baro * params.w_z_baro$;	

<http://blog.csdn.net/>