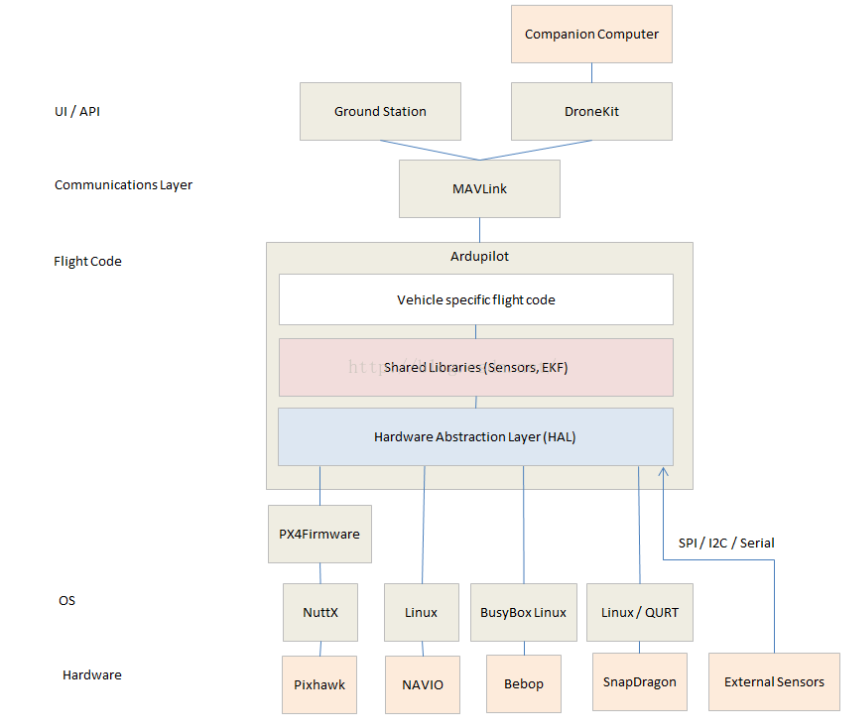
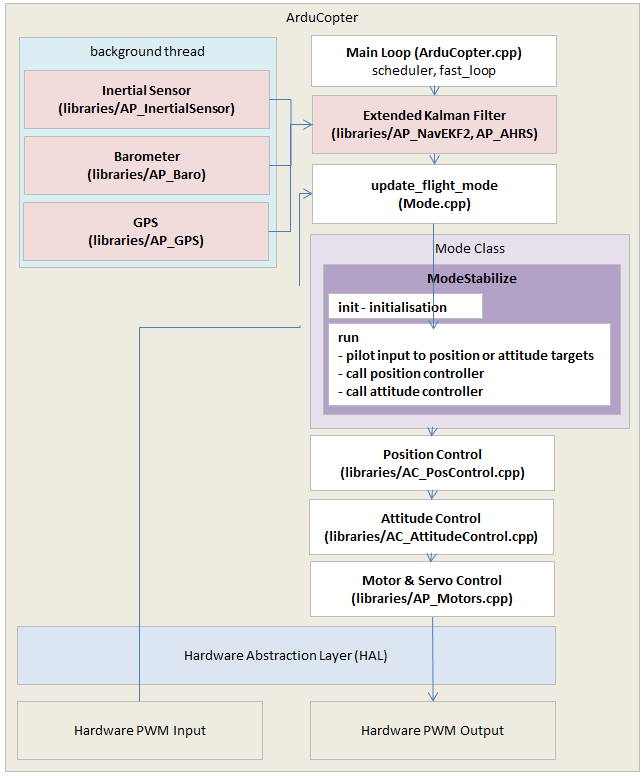
ardupilot整体结构框图：



导航部分框图：



2）Libraries介绍

（1）核心库

AP\_AHRS：采用DCM（方向余弦矩阵方法）或EKF（扩展卡尔曼滤波方法）预估飞行器姿态。

AP\_Common：所有执行文件（sketch格式，arduino IDE的文件）和其他库都需要的基础核心库。

AP\_Math：包含了许多数学函数，特别对于矢量运算。

AC\_PID：PID控制器库。

AP\_InertialNav：扩展带有gps和气压计数据的惯性导航库。

AC\_AttitudeControl：姿态控制相关库。

AP\_WPNav：航点相关的导航库。

AP\_Motors：多旋翼和传统直升机混合的电机库。

RC\_Channel：更多的关于从APM\_RC的PWM输入/输出数据转换到内部通用单位的库，比如角度。

AP\_HAL，AP\_HAL\_AVR，AP\_HAL\_PX4:硬件抽象层库，提供给其他高级控制代码一致的接口，而不必担心底层不同的硬件。AP\_HAL\_PX4：GPIO、I2C、UART、RCinput/output、scheduler、semaphores、storage。

（2）传感器相关库

AP\_InertialSensor：读取陀螺仪和加速度计数据，并向主程序执行标准程序和提供标准单位数据（deg/s，m/s）。

AP\_RangerFinder：声呐和红外测距传感器的交互库

AP\_Baro：气压计相关库

AP\_GPS：GPS相关库

AP\_Compass：三轴罗盘相关库

AP\_OpticalFlow：光流传感器相关库

（3）其他库

AP\_Mount，AP\_Camera, AP\_Relay：相机安装控制库，相机快门控制库

AP\_Mission： 从eeprom（电可擦只读存储器）存储/读取飞行指令相关库

AP\_Buffer：惯性导航时所用到的一个简单的堆栈（FIFO，先进先出）缓冲区

AP\_AccelCal、AP\_Declination、AP\_RCMapper、AP\_RPM、AP\_RSSI

AP\_ADC：Analog to Digital

APM\_Control： pitch/roll/yaw controller

DataFlash：flash memory

GCS\_Console/GCS\_MAVLink：地面站通信、飞行日志

所用EKF介绍：

与DCM相比，EKF优势在于通过融合所有可用的测量，能够拒绝某些带有明显误差的测量值，不易受到单个传感器故障的影响。EKF另一优势可以估计误差量，另外还可以利用光流、激光测距等其他传感器辅助测量。KF能够使用不同误差和不同状态之间的相关性来校准被测量状态之外的状态，如GPS测量的位置可以用来校正位置、速度、姿态和陀螺漂移。校正量是由状态误差和测量误差假定的比率确定的。如果KF认为预测的位置准确，那GPS测量的校正量变小，反之变大。GPS的精度由EKF\_POSNE\_NOISE控制，其值过大将导致滤波器认为GPS不准确。

程序中所用EKF的状态和量测见附录：

参数调整：

AHRS\_EKF\_USE：设为1启动滤波器；设为0使用传统算法；

EKF\_ABIAS\_PNOISE：控制天向加计零偏误差估计的增长速度，增大会使加计bias估计更快更嘈杂。

EKF\_EAS\_NOISE：罗盘测量噪声中的RMS值，增大它可减少测量的权重；

EKF\_FALLBACK：设置为1的话，当数据有问题时，会回退到DCM；

EKF\_GBIAS\_PNOISE：该噪声控制陀螺零偏状态误差的增长，增加它会使估计更快更嘈杂。

EKF\_GLITCH\_ACCEL：滤波器预测的值域GPS测量的值之间的水平加速度最大差异值。设置过低，将丢弃有用的GPS数据；设置过高，GPS故障会导致位置发生较大偏差；

EKF\_GLITCH\_RAD：在故障逻辑未被激活且未利用GPS量测进行偏移补偿之前，一步预测的水平位置和GPS的水平位置之间最大差值，防止位置发生较大跳跃。

EKF\_GPS\_TYPE：0 使用3D速度；1使用2D速度；2 不使用速度；

EKF\_GYRO\_PNOISE：除偏置外其他陀螺测量误差导致的估计误差增长，增大会减少对陀螺测量的信任，更多信任其他测量；

EKF\_POSNE\_NOISE：GPS水平测量噪声的RMS值。增大，则GPS噪声较大，且水平GPS测量权值较小；如果设置过小，即信任GPS量测，则滤波器不断对GPS量测中噪声作出反应，导致持续的快速的姿态和位置波动；设置过大，惯性器件误差的导致的滤波器位置速度漂移未被修正，过度漂移；

EKF\_POS\_GATE：用于GPS量测的检查，GPS量测大于某值将被舍弃；减小，将导致正确的测量值被舍弃；增大，将引入过大误差；

EKF\_RNG\_GATE：在地面上EKF失效之前，测量与一步预测的最大偏差；

EKF\_VELD\_NOISE：GPS天向速度的RMS值。增大，则滤波器认为GPS天向速度噪声大，天向速度权值较小。参数设置过小，相信GPS天向速度，滤波器会对天向速度噪声敏感，会上下抖动。

EKF\_VELNE\_NOISE：GPS水平速度RMS值。同上，俯仰、横滚对噪声比较敏感。

EKF1

TimeMS - time in msec from startup

Roll - Roll angle (deg)

Pitch - Pitch angle (deg)

Yaw - Yaw angle (deg)

VN,VE,VD - North,East,Down velocities (m/s)

PN,PE,PD - North,East,Down positions (m) relative to where the vehicle was armed

GX,GY,GZ - X,Y,Z Gyro biases (deg/min)

随着传感器温度升高和工作温度升高，低成本的MEMS陀螺零偏会发生明显变化。

EKF2

Ratio：在存在两个IMU情况下，IMU1加计数据的百分比

AZ1bias - Z accelerometer bias for IMU1 (cm/s:sup:2)

AZ2bias - Z accelerometer bias for IMU2 (cm/s:sup:2)

VWN,VWE - North and East wind velocity (m/s). A positive value means the wind is moving in the direction of that axis, eg a positive North wind velocity is blowing from the South.

MN,ME,MD - North, East, Down earth magnetic field strength (sensor units). If you are flying quickly, or are at low speed with EKF\_MAG\_CAL enabled, these will slowly change during flight as the filter ‘learns’ the earth’s magnetic field.

MX,MY,MZ - X, Y, Z body magnetic field biases (sensor units). If you are flying quickly, or are at low speed with EKF\_MAG\_CAL enabled, these will slowly change during flight as the filter ‘learns’ the earth’s magnetic field. These have the same meaning as the compass offsets, but are the opposite sign (eg in the following figure MX stabilises at a value of +35, indicating that a COMPASS\_OFS\_X value of -35 should be used.

EKF3：

EKF4：

EKF的估计：

它是AP\_NavEKF2库中的24状态扩展卡尔曼滤波器，用于估计以下状态。模型见附录2：

态度（四元数）

速度（北，东，下）

位置（北，东，下）

陀螺偏置偏移（X，Y，Z）

陀螺比例因子（X，Y，Z）

Z accel偏差

地球磁场（北，东，下）

体磁场（X，Y，Z）

风速（北，东）

它可以为每个IMU运行单独的EKF2，从而更有可能从IMU故障中恢复

如果有故障，它可以在磁力计之间切换

它可以估计陀螺比例因子，可以提高高速率机动时的准确度

EKF2不是直接估计四元数，而是误差旋转矢量，然后修正四元数。在大角度时此种方法更好，避免了四元素线性化带来的较大误差；数学计算和代码得到进一步优化；增加了基于固定偏差的磁罗盘航向角融合方法，适用于地面或者存在磁干扰而无法使用三轴六维的磁融合技术；The new EKF runs on a delayed time horizon so that when measurements are fused, they are done so using the measurement, filter states and covariance matrix from the same point in time. The legacy EKF uses a simpler method where delayed measurements are fused using delayed states but the covariance matrix is from the current time which reduces accuracy.。

使用EKF2

通过设置EK2\_ENABLE = 1启用新的EKF.EKF2现在将并行运行并记录等，但控制回路不会使用它。

要在控制循环中使用它，请设置AHRS\_EKF\_TYPE = 2

多个IMU的使用由EK2\_IMU\_MASK参数控制：

要仅使用IMU1，请设置EK2\_IMU\_MASK = 1（这是默认值）

要仅使用IMU2，请设置EK2\_IMU\_MASK = 2

要使用IMU1和IMU2运行双EKF2，请设置EK2\_IMU\_MASK = 3并通过设置EKF\_ENABLE = 0关闭传统EKF。设置参数后，需要重新启动。

而用加速度计积分做位置的话。AHRS是不现实的（1分钟就能飘出几十米，而且是成二次方的速度递增）。AHRS通常要结合GPS和气压计做位置，

附录1：

% State vector:

% quaternions (q0, q1, q2, q3)

% Velocity - m/sec (North, East, Down)

% Position - m (North, East, Down)

% Delta Angle bias - rad (X,Y,Z)

% Delta Velocity bias - m/s (Z)

% Wind Vector - m/sec (North,East)

% Earth Magnetic Field Vector - milligauss (North, East, Down)

% Body Magnetic Field Vector - milligauss (X,Y,Z)

% Observations:

% NED velocity - m/s

% NED position - m

% True airspeed - m/s

% XYZ magnetic flux - milligauss

% XY line of sight angular rate measurements from a downwards looking optical flow sensor

% range to terrain measurements

% Time varying parameters:

% XYZ delta angle measurements in body axes - rad

% XYZ delta velocity measurements in body axes - m/sec

clear all;

%% define symbolic variables and constants

syms dax day daz real % IMU delta angle measurements in body axes - rad

syms dvx dvy dvz real % IMU delta velocity measurements in body axes - m/sec

syms q0 q1 q2 q3 real % quaternions defining attitude of body axes relative to local NED

syms vn ve vd real % NED velocity - m/sec

syms pn pe pd real % NED position - m

syms dax\_b day\_b daz\_b real % delta angle bias - rad

syms dvz\_b real % delta velocity bias - m/sec

syms dt real % IMU time step - sec

syms gn ge gd real % NED gravity vector - m/sec^2

syms daxCov dayCov dazCov dvxCov dvyCov dvzCov real; % IMU delta angle and delta velocity measurement variances

syms vwn vwe real; % NE wind velocity - m/sec

syms magX magY magZ real; % XYZ body fixed magnetic field measurements - milligauss

syms magN magE magD real; % NED earth fixed magnetic field components - milligauss

syms R\_VN R\_VE R\_VD real % variances for NED velocity measurements - (m/sec)^2

syms R\_PN R\_PE R\_PD real % variances for NED position measurements - m^2

syms R\_TAS real % variance for true airspeed measurement - (m/sec)^2

syms R\_MAG real % variance for magnetic flux measurements - milligauss^2

syms R\_LOS real % variance of LOS angular rate mesurements (rad/sec)^2

syms R\_RNG real % variance of laser range finder observations

syms ptd real % location of terrain in D axis

syms BCXinv BCYinv real % inverse of ballistic coefficient for wind relative movement along the x and y body axes

syms rho real % air density (kg/m^3)

syms R\_ACC real % variance of accelerometer measurements (m/s^2)^2

syms Kacc real % ratio of horizontal acceleration to top speed for a multirotor

syms decl real; % earth magnetic field declination from true north

syms R\_MAGS real; % variance for magnetic deviation measurement

%% define the process equations

% Define the state vector & number of states

stateVector = [q0;q1;q2;q3;vn;ve;vd;pn;pe;pd;dax\_b;day\_b;daz\_b;dvz\_b;vwn;vwe;magN;magE;magD;magX;magY;magZ];

nStates=numel(stateVector);

% define the measured Delta angle and delta velocity vectors

da = [dax; day; daz];

dv = [dvx; dvy; dvz];

% define the delta angle and delta velocity bias errors

da\_b = [dax\_b; day\_b; daz\_b];

dv\_b = [0; 0; dvz\_b];

% derive the body to nav direction cosine matrix

Tbn = Quat2Tbn([q0,q1,q2,q3]);

% define the bias corrected delta angles and velocities

dAngCor = da - da\_b;

dVelCor = dv - dv\_b;

% define the quaternion rotation vector

quat = [q0;q1;q2;q3];

% define the attitude update equations

delQuat = [1;

0.5\*dAngCor(1);

0.5\*dAngCor(2);

0.5\*dAngCor(3);

];

qNew = QuatMult(quat,delQuat);

% define the velocity update equations

vNew = [vn;ve;vd] + [gn;ge;gd]\*dt + Tbn\*dVelCor;

% define the position update equations

pNew = [pn;pe;pd] + [vn;ve;vd]\*dt;

% define the IMU bias error update equations

dabNew = [dax\_b; day\_b; daz\_b];

dvbNew = dvz\_b;

% define the wind velocity update equations

vwnNew = vwn;

vweNew = vwe;

% define the earth magnetic field update equations

magNnew = magN;

magEnew = magE;

magDnew = magD;

% define the body magnetic field update equations

magXnew = magX;

magYnew = magY;

magZnew = magZ;

% Define the process equations output vector

processEqns = [qNew;vNew;pNew;dabNew;dvbNew;vwnNew;vweNew;magNnew;magEnew;magDnew;magXnew;magYnew;magZnew];

%% derive the state transition matrix

% derive the state transition matrix

F = jacobian(processEqns, stateVector);

[F,SF]=OptimiseAlgebra(F,'SF');

%% derive the covariance prediction equation

% This reduces the number of floating point operations by a factor of 6 or

% more compared to using the standard matrix operations in code

% Define the control (disturbance) vector. Error growth in the inertial

% solution is assumed to be driven by 'noise' in the delta angles and

% velocities, after bias effects have been removed. This is OK becasue we

% have sensor bias accounted for in the state equations.

distVector = [da;dv];

% derive the control(disturbance) influence matrix

G = jacobian(processEqns, distVector);

[G,SG]=OptimiseAlgebra(G,'SG');

% derive the state error matrix

imuNoise = diag([daxCov dayCov dazCov dvxCov dvyCov dvzCov]);

Q = G\*imuNoise\*transpose(G);

[Q,SQ]=OptimiseAlgebra(Q,'SQ');

% define a symbolic covariance matrix using strings to represent

% '\_l\_' to represent '( '

% '\_c\_' to represent ,

% '\_r\_' to represent ')'

% these can be substituted later to create executable code

for rowIndex = 1:nStates

for colIndex = 1:nStates

eval(['syms OP\_l\_',num2str(rowIndex),'\_c\_',num2str(colIndex), '\_r\_ real']);

eval(['P(',num2str(rowIndex),',',num2str(colIndex), ') = OP\_l\_',num2str(rowIndex),'\_c\_',num2str(colIndex),'\_r\_;']);

end

end

% Derive the predicted covariance matrix using the standard equation

PP = F\*P\*transpose(F) + Q;

% Collect common expressions to optimise processing

[PP,SPP]=OptimiseAlgebra(PP,'SPP');

%% derive equations for sequential fusion of velocity and position measurements

H\_VN= jacobian(vn,stateVector); % measurement Jacobian

K\_VN = (P\*transpose(H\_VN))/(H\_VN\*P\*transpose(H\_VN) + R\_VN);

H\_VE= jacobian(ve,stateVector); % measurement Jacobian

K\_VE = (P\*transpose(H\_VE))/(H\_VE\*P\*transpose(H\_VE) + R\_VE);

H\_VD= jacobian(vd,stateVector); % measurement Jacobian

K\_VD = (P\*transpose(H\_VD))/(H\_VD\*P\*transpose(H\_VD) + R\_VD);

H\_PN= jacobian(pn,stateVector); % measurement Jacobian

K\_PN = (P\*transpose(H\_PN))/(H\_PN\*P\*transpose(H\_PN) + R\_PN);

H\_PE= jacobian(pe,stateVector); % measurement Jacobian

K\_PE = (P\*transpose(H\_PE))/(H\_PE\*P\*transpose(H\_PE) + R\_PE);

H\_PD= jacobian(pd,stateVector); % measurement Jacobian

K\_PD = (P\*transpose(H\_PD))/(H\_PD\*P\*transpose(H\_PD) + R\_PD);

% combine into a single H and K matrix (note these matrices cannot be used

% for a single step fusion, so each row|column must be used in a separate

% fusion step

H\_VP = [H\_VN;H\_VE;H\_VD;H\_PN;H\_PE;H\_PD];

clear H\_VN H\_VE H\_VD H\_PN H\_PE H\_PD

K\_VP = [K\_VN,K\_VE,K\_VD,K\_PN,K\_PE,K\_PD];

clear K\_VN K\_VE K\_VD K\_PN K\_PE K\_PD

[K\_VP,SK\_VP]=OptimiseAlgebra(K\_VP,'SK\_VP');

%% derive equations for fusion of true airspeed measurements

VtasPred = sqrt((vn-vwn)^2 + (ve-vwe)^2 + vd^2); % predicted measurement

H\_TAS = jacobian(VtasPred,stateVector); % measurement Jacobian

[H\_TAS,SH\_TAS]=OptimiseAlgebra(H\_TAS,'SH\_TAS'); % optimise processing

K\_TAS = (P\*transpose(H\_TAS))/(H\_TAS\*P\*transpose(H\_TAS) + R\_TAS);[K\_TAS,SK\_TAS]=OptimiseAlgebra(K\_TAS,'SK\_TAS'); % Kalman gain vector

%% derive equations for fusion of magnetic field measurement

magMeas = transpose(Tbn)\*[magN;magE;magD] + [magX;magY;magZ]; % predicted measurement

H\_MAG = jacobian(magMeas,stateVector); % measurement Jacobian

[H\_MAG,SH\_MAG]=OptimiseAlgebra(H\_MAG,'SH\_MAG');

K\_MX = (P\*transpose(H\_MAG(1,:)))/(H\_MAG(1,:)\*P\*transpose(H\_MAG(1,:)) + R\_MAG); % Kalman gain vector

[K\_MX,SK\_MX]=OptimiseAlgebra(K\_MX,'SK\_MX');

K\_MY = (P\*transpose(H\_MAG(2,:)))/(H\_MAG(2,:)\*P\*transpose(H\_MAG(2,:)) + R\_MAG); % Kalman gain vector

[K\_MY,SK\_MY]=OptimiseAlgebra(K\_MY,'SK\_MY');

K\_MZ = (P\*transpose(H\_MAG(3,:)))/(H\_MAG(3,:)\*P\*transpose(H\_MAG(3,:)) + R\_MAG); % Kalman gain vector

[K\_MZ,SK\_MZ]=OptimiseAlgebra(K\_MZ,'SK\_MZ');

%% derive equations for sequential fusion of optical flow measurements

% calculate range from plane to centre of sensor fov assuming flat earth

% and camera axes aligned with body axes

range = ((ptd - pd)/Tbn(3,3));

% calculate relative velocity in body frame

relVelBody = transpose(Tbn)\*[vn;ve;vd];

% divide by range to get predicted angular LOS rates relative to X and Y

% axes. Note these are body angular rate motion compensated optical flow rates

losRateX = +relVelBody(2)/range;

losRateY = -relVelBody(1)/range;

H\_LOS = jacobian([losRateX;losRateY],stateVector); % measurement Jacobian

[H\_LOS,SH\_LOS]=OptimiseAlgebra(H\_LOS,'SH\_LOS');

% combine into a single K matrix to enable common expressions to be found

% note this matrix cannot be used in a single step fusion

K\_LOSX = (P\*transpose(H\_LOS(1,:)))/(H\_LOS(1,:)\*P\*transpose(H\_LOS(1,:)) + R\_LOS); % Kalman gain vector

K\_LOSY = (P\*transpose(H\_LOS(2,:)))/(H\_LOS(2,:)\*P\*transpose(H\_LOS(2,:)) + R\_LOS); % Kalman gain vector

K\_LOS = [K\_LOSX,K\_LOSY];

simplify(K\_LOS);

[K\_LOS,SK\_LOS]=OptimiseAlgebra(K\_LOS,'SK\_LOS');

%% derive equations for fusion of laser range finder measurement

% calculate range from plane to centre of sensor fov assuming flat earth

% and sensor aligned with Z body axis

range = ((ptd - pd)/Tbn(3,3));

H\_RNG = jacobian(range,stateVector); % measurement Jacobian

[H\_RNG,SH\_RNG]=OptimiseAlgebra(H\_RNG,'SH\_RNG');

% calculate the Kalman gain matrix and optimise algebra

K\_RNG = (P\*transpose(H\_RNG))/(H\_RNG\*P\*transpose(H\_RNG) + R\_RNG);

[K\_RNG,SK\_RNG]=OptimiseAlgebra(K\_RNG,'SK\_RNG');

%% derive equations for fusion of lateral body acceleration (multirotors only)

% use relationship between airspeed along the X and Y body axis and the

% drag to predict the lateral acceleration for a multirotor vehicle type

% where propulsion forces are generated primarily along the Z body axis

vrel = transpose(Tbn)\*[(vn-vwn);(ve-vwe);vd]; % predicted wind relative velocity

% calculate drag assuming flight along axis in positive direction

% sign change will be looked after in implementation rather than by adding

% sign functions to symbolic derivation which genererates output with dirac

% functions

% accXpred = -0.5\*rho\*vrel(1)\*vrel(1)\*BCXinv; % predicted acceleration measured along X body axis

% accYpred = -0.5\*rho\*vrel(2)\*vrel(2)\*BCYinv; % predicted acceleration measured along Y body axis

% Use a simple viscous drag model for the linear estimator equations

% Use the the derivative from speed to acceleration averaged across the

% speed range

% The nonlinear equation will be used to calculate the predicted

% measurement in implementation

accXpred = -Kacc\*vrel(1); % predicted acceleration measured along X body axis

accYpred = -Kacc\*vrel(2); % predicted acceleration measured along Y body axis

% Derive observation Jacobian and Kalman gain matrix for X accel fusion

H\_ACCX = jacobian(accXpred,stateVector); % measurement Jacobian

[H\_ACCX,SH\_ACCX]=OptimiseAlgebra(H\_ACCX,'SH\_ACCX'); % optimise processing

K\_ACCX = (P\*transpose(H\_ACCX))/(H\_ACCX\*P\*transpose(H\_ACCX) + R\_ACC);

[K\_ACCX,SK\_ACCX]=OptimiseAlgebra(K\_ACCX,'SK\_ACCX'); % Kalman gain vector

% Derive observation Jacobian and Kalman gain matrix for Y accel fusion

H\_ACCY = jacobian(accYpred,stateVector); % measurement Jacobian

[H\_ACCY,SH\_ACCY]=OptimiseAlgebra(H\_ACCY,'SH\_ACCY'); % optimise processing

K\_ACCY = (P\*transpose(H\_ACCY))/(H\_ACCY\*P\*transpose(H\_ACCY) + R\_ACC);

[K\_ACCY,SK\_ACCY]=OptimiseAlgebra(K\_ACCY,'SK\_ACCY'); % Kalman gain vector

%% derive equations for fusion of magnetic deviation measurement

% measured field in earth frame

magMeas = transpose(Tbn)\*[magN;magE;magD] + [magX;magY;magZ]; % predicted measurement

% measured field in body frame

magMeasNED = Tbn\*magMeas;

% the predicted measurement is the angle wrt magnetic north of the horizontal

% component of the measured field

angMeas = tan(magMeasNED(2)/magMeasNED(1)) - decl;

H\_MAGS = jacobian(angMeas,stateVector); % measurement Jacobian

H\_MAGS = simplify(H\_MAGS);

[H\_MAGS,SH\_MAGS]=OptimiseAlgebra(H\_MAGS,'SH\_MAGS');

K\_MAGS = (P\*transpose(H\_MAGS))/(H\_MAGS\*P\*transpose(H\_MAGS) + R\_MAGS);

[K\_MAGS,SK\_MAGS]=OptimiseAlgebra(K\_MAGS,'SK\_MAGS'); % Kalman gain vector

%% Save output and convert to m and c code fragments

nStates = 22;

fileName = strcat('SymbolicOutput',int2str(nStates),'.mat');

save(fileName);

SaveScriptCode(nStates);

ConvertToM(nStates);

ConvertToC(nStates);

附录2：

% State vector:

% error rotation vector in body frame (X,Y,Z)

% Velocity - m/sec (North, East, Down)

% Position - m (North, East, Down)

% Delta Angle bias - rad (X,Y,Z)

% Delta Angle scale factor (X,Y,Z)

% Delta Velocity bias - m/s (Z)

% Earth Magnetic Field Vector - (North, East, Down)

% Body Magnetic Field Vector - (X,Y,Z)

% Wind Vector - m/sec (North,East)

% Observations:

% NED velocity - m/s

% NED position - m

% True airspeed - m/s

% angle of sideslip - rad

% XYZ magnetic flux

% Time varying parameters:

% XYZ delta angle measurements in body axes - rad

% XYZ delta velocity measurements in body axes - m/sec

%% define symbolic variables and constants

clear all;

reset(symengine);

syms dax day daz 'real' % IMU delta angle measurements in body axes - rad

syms dvx dvy dvz 'real' % IMU delta velocity measurements in body axes - m/sec

syms q0 q1 q2 q3 'real' % quaternions defining attitude of body axes relative to local NED

syms vn ve vd 'real' % NED velocity - m/sec

syms pn pe pd 'real' % NED position - m

syms dax\_b day\_b daz\_b 'real' % delta angle bias - rad

syms dax\_s day\_s daz\_s 'real' % delta angle scale factor

syms dvz\_b dvy\_b dvz\_b 'real' % delta velocity bias - m/sec

syms dt 'real' % IMU time step - sec

syms gravity 'real' % gravity - m/sec^2

syms daxNoise dayNoise dazNoise dvxNoise dvyNoise dvzNoise 'real'; % IMU delta angle and delta velocity measurement noise

syms vwn vwe 'real'; % NE wind velocity - m/sec

syms magX magY magZ 'real'; % XYZ body fixed magnetic field measurements - milligauss

syms magN magE magD 'real'; % NED earth fixed magnetic field components - milligauss

syms R\_VN R\_VE R\_VD 'real' % variances for NED velocity measurements - (m/sec)^2

syms R\_PN R\_PE R\_PD 'real' % variances for NED position measurements - m^2

syms R\_TAS 'real' % variance for true airspeed measurement - (m/sec)^2

syms R\_MAG 'real' % variance for magnetic flux measurements - milligauss^2

syms R\_BETA 'real' % variance of sidelsip measurements rad^2

syms R\_LOS 'real' % variance of LOS angular rate mesurements (rad/sec)^2

syms ptd 'real' % location of terrain in D axis

syms rotErrX rotErrY rotErrZ 'real'; % error rotation vector in body frame

syms decl 'real'; % earth magnetic field declination from true north

syms R\_DECL R\_YAW 'real'; % variance of declination or yaw angle observation

syms BCXinv BCYinv 'real' % inverse of ballistic coefficient for wind relative movement along the x and y body axes

syms rho 'real' % air density (kg/m^3)

syms R\_ACC 'real' % variance of accelerometer measurements (m/s^2)^2

syms Kaccx Kaccy 'real' % derivative of X and Y body specific forces wrt componenent of true airspeed along each axis (1/s)

%% define the state prediction equations

% define the measured Delta angle and delta velocity vectors

dAngMeas = [dax; day; daz];

dVelMeas = [dvx; dvy; dvz];

% define the IMU bias errors and scale factor

dAngBias = [dax\_b; day\_b; daz\_b];

dAngScale = [dax\_s; day\_s; daz\_s];

dVelBias = [0;0;dvz\_b];

% define the quaternion rotation vector for the state estimate

estQuat = [q0;q1;q2;q3];

% define the attitude error rotation vector, where error = truth - estimate

errRotVec = [rotErrX;rotErrY;rotErrZ];

% define the attitude error quaternion using a first order linearisation

errQuat = [1;0.5\*errRotVec];

% Define the truth quaternion as the estimate + error

truthQuat = QuatMult(estQuat, errQuat);

% derive the truth body to nav direction cosine matrix

Tbn = Quat2Tbn(truthQuat);

% define the truth delta angle

% ignore coning compensation as these effects are negligible in terms of

% covariance growth for our application and grade of sensor

dAngTruth = dAngMeas.\*dAngScale - dAngBias - [daxNoise;dayNoise;dazNoise];

% define the attitude update equations

% use a first order expansion of rotation to calculate the quaternion increment

% acceptable for propagation of covariances

deltaQuat = [1;

0.5\*dAngTruth(1);

0.5\*dAngTruth(2);

0.5\*dAngTruth(3);

];

truthQuatNew = QuatMult(truthQuat,deltaQuat);

% calculate the updated attitude error quaternion with respect to the previous estimate

errQuatNew = QuatDivide(truthQuatNew,estQuat);

% change to a rotaton vector - this is the error rotation vector updated state

errRotNew = 2 \* [errQuatNew(2);errQuatNew(3);errQuatNew(4)];

% Define the truth delta velocity -ignore sculling and transport rate

% corrections as these negligible are in terms of covariance growth for our

% application and grade of sensor

dVelTruth = dVelMeas - dVelBias - [dvxNoise;dvyNoise;dvzNoise];

% define the velocity update equations

% ignore coriolis terms for linearisation purposes

vNew = [vn;ve;vd] + [0;0;gravity]\*dt + Tbn\*dVelTruth;

% define the position update equations

pNew = [pn;pe;pd] + [vn;ve;vd]\*dt;

% define the IMU error update equations

dabNew = [dax\_b; day\_b; daz\_b];

dasNew = [dax\_s; day\_s; daz\_s];

dvbNew = dvz\_b;

% define the wind velocity update equations

vwnNew = vwn;

vweNew = vwe;

% define the earth magnetic field update equations

magNnew = magN;

magEnew = magE;

magDnew = magD;

% define the body magnetic field update equations

magXnew = magX;

magYnew = magY;

magZnew = magZ;

% Define the state vector & number of states

stateVector = [errRotVec;vn;ve;vd;pn;pe;pd;dax\_b;day\_b;daz\_b;dax\_s;day\_s;daz\_s;dvz\_b;magN;magE;magD;magX;magY;magZ;vwn;vwe];

nStates=numel(stateVector);

% Define vector of process equations

newStateVector = [errRotNew;vNew;pNew;dabNew;dasNew;dvbNew;magNnew;magEnew;magDnew;magXnew;magYnew;magZnew;vwnNew;vweNew];

% derive the state transition matrix

F = jacobian(newStateVector, stateVector);

% set the rotation error states to zero

F = subs(F, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

[F,SF]=OptimiseAlgebra(F,'SF');

% define a symbolic covariance matrix using strings to represent

% '\_l\_' to represent '( '

% '\_c\_' to represent ,

% '\_r\_' to represent ')'

% these can be substituted later to create executable code

for rowIndex = 1:nStates

for colIndex = 1:nStates

eval(['syms OP\_l\_',num2str(rowIndex),'\_c\_',num2str(colIndex), '\_r\_ real']);

eval(['P(',num2str(rowIndex),',',num2str(colIndex), ') = OP\_l\_',num2str(rowIndex),'\_c\_',num2str(colIndex),'\_r\_;']);

end

end

save 'StatePrediction.mat';

%% derive the covariance prediction equations

% This reduces the number of floating point operations by a factor of 6 or

% more compared to using the standard matrix operations in code

% Define the control (disturbance) vector. Error growth in the inertial

% solution is assumed to be driven by 'noise' in the delta angles and

% velocities, after bias effects have been removed. This is OK becasue we

% have sensor bias accounted for in the state equations.

distVector = [daxNoise;dayNoise;dazNoise;dvxNoise;dvyNoise;dvzNoise];

% derive the control(disturbance) influence matrix

G = jacobian(newStateVector, distVector);

G = subs(G, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

[G,SG]=OptimiseAlgebra(G,'SG');

% derive the state error matrix

distMatrix = diag(distVector.^2);

Q = G\*distMatrix\*transpose(G);

[Q,SQ]=OptimiseAlgebra(Q,'SQ');

% remove the disturbance noise from the process equations as it is only

% needed when calculating the disturbance influence matrix

vNew = subs(vNew,{'daxNoise','dayNoise','dazNoise','dvxNoise','dvyNoise','dvzNoise'}, {0,0,0,0,0,0});

errRotNew = subs(errRotNew,{'daxNoise','dayNoise','dazNoise','dvxNoise','dvyNoise','dvzNoise'}, {0,0,0,0,0,0});

% Derive the predicted covariance matrix using the standard equation

PP = F\*P\*transpose(F) + Q;

% Collect common expressions to optimise processing

[PP,SPP]=OptimiseAlgebra(PP,'SPP');

save('StateAndCovariancePrediction.mat');

clear all;

reset(symengine);

%% derive equations for fusion of true airspeed measurements

load('StatePrediction.mat');

VtasPred = sqrt((vn-vwn)^2 + (ve-vwe)^2 + vd^2); % predicted measurement

H\_TAS = jacobian(VtasPred,stateVector); % measurement Jacobian

H\_TAS = subs(H\_TAS, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

[H\_TAS,SH\_TAS]=OptimiseAlgebra(H\_TAS,'SH\_TAS'); % optimise processing

K\_TAS = (P\*transpose(H\_TAS))/(H\_TAS\*P\*transpose(H\_TAS) + R\_TAS);

[K\_TAS,SK\_TAS]=OptimiseAlgebra(K\_TAS,'SK\_TAS'); % Kalman gain vector

% save equations and reset workspace

save('Airspeed.mat','SH\_TAS','H\_TAS','SK\_TAS','K\_TAS');

clear all;

reset(symengine);

%% derive equations for fusion of angle of sideslip measurements

load('StatePrediction.mat');

% calculate wind relative velocities in nav frame and rotate into body frame

Vbw = Tbn'\*[(vn-vwn);(ve-vwe);vd];

% calculate predicted angle of sideslip using small angle assumption

BetaPred = Vbw(2)/Vbw(1);

H\_BETA = jacobian(BetaPred,stateVector); % measurement Jacobian

H\_BETA = subs(H\_BETA, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

[H\_BETA,SH\_BETA]=OptimiseAlgebra(H\_BETA,'SH\_BETA'); % optimise processing

K\_BETA = (P\*transpose(H\_BETA))/(H\_BETA\*P\*transpose(H\_BETA) + R\_BETA);[K\_BETA,SK\_BETA]=OptimiseAlgebra(K\_BETA,'SK\_BETA'); % Kalman gain vector

% save equations and reset workspace

save('Sideslip.mat','SH\_BETA','H\_BETA','SK\_BETA','K\_BETA');

clear all;

reset(symengine);

%% derive equations for fusion of magnetic field measurement

load('StatePrediction.mat');

magMeas = transpose(Tbn)\*[magN;magE;magD] + [magX;magY;magZ]; % predicted measurement

H\_MAG = jacobian(magMeas,stateVector); % measurement Jacobian

H\_MAG = subs(H\_MAG, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

[H\_MAG,SH\_MAG]=OptimiseAlgebra(H\_MAG,'SH\_MAG');

K\_MX = (P\*transpose(H\_MAG(1,:)))/(H\_MAG(1,:)\*P\*transpose(H\_MAG(1,:)) + R\_MAG); % Kalman gain vector

[K\_MX,SK\_MX]=OptimiseAlgebra(K\_MX,'SK\_MX');

K\_MY = (P\*transpose(H\_MAG(2,:)))/(H\_MAG(2,:)\*P\*transpose(H\_MAG(2,:)) + R\_MAG); % Kalman gain vector

[K\_MY,SK\_MY]=OptimiseAlgebra(K\_MY,'SK\_MY');

K\_MZ = (P\*transpose(H\_MAG(3,:)))/(H\_MAG(3,:)\*P\*transpose(H\_MAG(3,:)) + R\_MAG); % Kalman gain vector

[K\_MZ,SK\_MZ]=OptimiseAlgebra(K\_MZ,'SK\_MZ');

% save equations and reset workspace

save('Magnetometer.mat','SH\_MAG','H\_MAG','SK\_MX','K\_MX','SK\_MY','K\_MY','SK\_MZ','K\_MZ');

clear all;

reset(symengine);

%% derive equations for sequential fusion of optical flow measurements - method 1

load('StatePrediction.mat');

% calculate range from plane to centre of sensor fov assuming flat earth

% and camera axes aligned with body axes

range = ((ptd - pd)/Tbn(3,3));

% calculate relative velocity in body frame

relVelBody = transpose(Tbn)\*[vn;ve;vd];

% divide by range to get predicted angular LOS rates relative to X and Y

% axes. Note these are body angular rate motion compensated optical flow rates

losRateX = +relVelBody(2)/range;

losRateY = -relVelBody(1)/range;

% calculte the observation Jacobian

H\_LOS = jacobian([losRateX;losRateY],stateVector); % measurement Jacobian

H\_LOS = subs(H\_LOS, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_LOS = simplify(H\_LOS);

% recursively simplify the equations

[H\_LOS,SH\_LOS] = OptimiseAlgebra(H\_LOS,'SH\_LOS');

% combine into a single K matrix to enable common expressions to be found

% note this matrix cannot be used in a single step fusion

K\_LOSX = (P\*transpose(H\_LOS(1,:)))/(H\_LOS(1,:)\*P\*transpose(H\_LOS(1,:)) + R\_LOS); % Kalman gain vector

K\_LOSX = subs(K\_LOSX, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

K\_LOSY = (P\*transpose(H\_LOS(2,:)))/(H\_LOS(2,:)\*P\*transpose(H\_LOS(2,:)) + R\_LOS); % Kalman gain vector

K\_LOSY = subs(K\_LOSY, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

K\_LOS = [K\_LOSX,K\_LOSY];

simplify(K\_LOS);

[K\_LOS,SK\_LOS]=OptimiseAlgebra(K\_LOS,'SK\_LOS');

K\_LOSX = K\_LOS(:,1);

K\_LOSY = K\_LOS(:,2);

% save equations and reset workspace

save('OpticalFlow.mat','SH\_LOS','H\_LOS','SK\_LOS','K\_LOSX','K\_LOSY');

clear all;

reset(symengine);

%% derive equations for sequential fusion of optical flow measurements - method 2

load('StatePrediction.mat');

% calculate range from plane to centre of sensor fov assuming flat earth

% and camera axes aligned with body axes

%range = ((ptd - pd)/Tbn(3,3));

syms range 'real';

% calculate relative velocity in body frame

relVelBody = transpose(Tbn)\*[vn;ve;vd];

% divide by range to get predicted angular LOS rates relative to X and Y

% axes. Note these are body angular rate motion compensated optical flow rates

losRateX = +relVelBody(2)/range;

losRateY = -relVelBody(1)/range;

save('temp1.mat','losRateX','losRateY');

% calculate the observation Jacobian for the X axis

H\_LOSX = jacobian(losRateX,stateVector); % measurement Jacobian

H\_LOSX = subs(H\_LOSX, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_LOSX = simplify(H\_LOSX);

save('temp2.mat','H\_LOSX');

ccode(H\_LOSX,'file','H\_LOSX.c');

fix\_c\_code('H\_LOSX.c');

clear all;

reset(symengine);

load('StatePrediction.mat');

load('temp1.mat');

% calculate the observation Jacobian for the Y axis

H\_LOSY = jacobian(losRateY,stateVector); % measurement Jacobian

H\_LOSY = subs(H\_LOSY, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_LOSY = simplify(H\_LOSY);

save('temp3.mat','H\_LOSY');

ccode(H\_LOSY,'file','H\_LOSY.c');

fix\_c\_code('H\_LOSY.c');

clear all;

reset(symengine);

load('StatePrediction.mat');

load('temp1.mat');

load('temp2.mat');

% calculate Kalman gain vector for the X axis

K\_LOSX = (P\*transpose(H\_LOSX))/(H\_LOSX\*P\*transpose(H\_LOSX) + R\_LOS); % Kalman gain vector

K\_LOSX = subs(K\_LOSX, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

K\_LOSX = simplify(K\_LOSX);

ccode(K\_LOSX,'file','K\_LOSX.c');

fix\_c\_code('K\_LOSX.c');

clear all;

reset(symengine);

load('StatePrediction.mat');

load('temp1.mat');

load('temp3.mat');

% calculate Kalman gain vector for the Y axis

K\_LOSY = (P\*transpose(H\_LOSY))/(H\_LOSY\*P\*transpose(H\_LOSY) + R\_LOS); % Kalman gain vector

K\_LOSY = subs(K\_LOSY, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

K\_LOSY = simplify(K\_LOSY);

ccode(K\_LOSY,'file','K\_LOSY.c');

fix\_c\_code('K\_LOSY.c');

% reset workspace

clear all;

reset(symengine);

%% derive equations for fusion of 321 sequence yaw measurement

load('StatePrediction.mat');

% Calculate the yaw (first rotation) angle from the 321 rotation sequence

angMeas = atan(Tbn(2,1)/Tbn(1,1));

H\_YAW321 = jacobian(angMeas,stateVector); % measurement Jacobian

H\_YAW321 = subs(H\_YAW321, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_YAW321 = simplify(H\_YAW321);

ccode(H\_YAW321,'file','calcH\_YAW321.c');

fix\_c\_code('calcH\_YAW321.c');

% reset workspace

clear all;

reset(symengine);

%% derive equations for fusion of 312 sequence yaw measurement

load('StatePrediction.mat');

% Calculate the yaw (first rotation) angle from an Euler 312 sequence

angMeas = atan(-Tbn(1,2)/Tbn(2,2));

H\_YAW312 = jacobian(angMeas,stateVector); % measurement Jacobianclea

H\_YAW312 = subs(H\_YAW312, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_YAW312 = simplify(H\_YAW312);

ccode(H\_YAW312,'file','calcH\_YAW312.c');

fix\_c\_code('calcH\_YAW312.c');

% reset workspace

clear all;

reset(symengine);

%% derive equations for fusion of declination

load('StatePrediction.mat');

% the predicted measurement is the angle wrt magnetic north of the horizontal

% component of the measured field

angMeas = atan(magE/magN);

H\_MAGD = jacobian(angMeas,stateVector); % measurement Jacobian

H\_MAGD = subs(H\_MAGD, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_MAGD = simplify(H\_MAGD);

K\_MAGD = (P\*transpose(H\_MAGD))/(H\_MAGD\*P\*transpose(H\_MAGD) + R\_DECL);

K\_MAGD = simplify(K\_MAGD);

ccode([K\_MAGD,H\_MAGD'],'file','calcMAGD.c');

fix\_c\_code('calcMAGD.c');

% reset workspace

clear all;

reset(symengine);

%% derive equations for fusion of lateral body acceleration (multirotors only)

load('StatePrediction.mat');

% use relationship between airspeed along the X and Y body axis and the

% drag to predict the lateral acceleration for a multirotor vehicle type

% where propulsion forces are generated primarily along the Z body axis

vrel = transpose(Tbn)\*[(vn-vwn);(ve-vwe);vd]; % predicted wind relative velocity

% calculate drag assuming flight along axis in positive direction

% sign change will be looked after in implementation rather than by adding

% sign functions to symbolic derivation which genererates output with dirac

% functions

% accXpred = -0.5\*rho\*vrel(1)\*vrel(1)\*BCXinv; % predicted acceleration measured along X body axis

% accYpred = -0.5\*rho\*vrel(2)\*vrel(2)\*BCYinv; % predicted acceleration measured along Y body axis

% Use a simple viscous drag model for the linear estimator equations

% Use the the derivative from speed to acceleration averaged across the

% speed range

% The nonlinear equation will be used to calculate the predicted

% measurement in implementation

accXpred = -Kaccx\*vrel(1); % predicted acceleration measured along X body axis

accYpred = -Kaccy\*vrel(2); % predicted acceleration measured along Y body axis

% Derive observation Jacobian and Kalman gain matrix for X accel fusion

H\_ACCX = jacobian(accXpred,stateVector); % measurement Jacobian

H\_ACCX = subs(H\_ACCX, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_ACCX = simplify(H\_ACCX);

[H\_ACCX,SH\_ACCX]=OptimiseAlgebra(H\_ACCX,'SH\_ACCX'); % optimise processing

K\_ACCX = (P\*transpose(H\_ACCX))/(H\_ACCX\*P\*transpose(H\_ACCX) + R\_ACC);

[K\_ACCX,SK\_ACCX]=OptimiseAlgebra(K\_ACCX,'SK\_ACCX'); % Kalman gain vector

% Derive observation Jacobian and Kalman gain matrix for Y accel fusion

H\_ACCY = jacobian(accYpred,stateVector); % measurement Jacobian

H\_ACCY = subs(H\_ACCY, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_ACCY = simplify(H\_ACCY);

[H\_ACCY,SH\_ACCY]=OptimiseAlgebra(H\_ACCY,'SH\_ACCY'); % optimise processing

K\_ACCY = (P\*transpose(H\_ACCY))/(H\_ACCY\*P\*transpose(H\_ACCY) + R\_ACC);

[K\_ACCY,SK\_ACCY]=OptimiseAlgebra(K\_ACCY,'SK\_ACCY'); % Kalman gain vector

% save equations and reset workspace

save('Drag.mat','SH\_ACCX','H\_ACCX','SK\_ACCX','K\_ACCX','SH\_ACCY','H\_ACCY','SK\_ACCY','K\_ACCY');

clear all;

reset(symengine);

%% Derive equations for fusion of range only measurements to a beacon at a known NED position

load('StatePrediction.mat');

syms bcn\_pn bcn\_pe bcn\_pd 'real' % beacon NED position

syms R\_BCN 'real' % observation variance for range measurement

rangePred = sqrt((pn - bcn\_pn)^2 + (pe - bcn\_pe)^2 + (pd - bcn\_pd)^2);

H\_BCN = jacobian(rangePred,stateVector); % measurement Jacobian

H\_BCN = subs(H\_BCN, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_BCN = simplify(H\_BCN);

[H\_BCN,SH\_BCN]=OptimiseAlgebra(H\_BCN,'SH\_BCN'); % optimise processing

K\_BCN = (P\*transpose(H\_BCN))/(H\_BCN\*P\*transpose(H\_BCN) + R\_BCN);

[K\_BCN,SK\_BCN]=OptimiseAlgebra(K\_BCN,'SK\_BCN'); % Kalman gain vector

save('RngBcn.mat','SH\_BCN','H\_BCN','SK\_BCN','K\_BCN');

clear all;

reset(symengine);

%% Derive equations for fusion of range only measurements to a beacon at a known NED position

load('StatePrediction.mat');

syms bcn\_pn bcn\_pe bcn\_pd 'real' % beacon NED position

syms R\_BCN 'real' % observation variance for range measurement

rangePred = sqrt((pn - bcn\_pn)^2 + (pe - bcn\_pe)^2 + (pd - bcn\_pd)^2);

H\_BCN = jacobian(rangePred,stateVector); % measurement Jacobian

H\_BCN = subs(H\_BCN, {'rotErrX', 'rotErrY', 'rotErrZ'}, {0,0,0});

H\_BCN = simplify(H\_BCN);

K\_BCN = (P\*transpose(H\_BCN))/(H\_BCN\*P\*transpose(H\_BCN) + R\_BCN);

ccode([K\_BCN,H\_BCN'],'file','calcBCN.c');

fix\_c\_code('calcBCN.c');

clear all;

reset(symengine);

%% Save output and convert to m and c code fragments

% load equations for predictions and updates

load('StateAndCovariancePrediction.mat');

load('Airspeed.mat');

load('Sideslip.mat');

load('Magnetometer.mat');

load('OpticalFlow.mat');

load('Drag.mat');

load('RngBcn.mat');

fileName = strcat('SymbolicOutput',int2str(nStates),'.mat');

save(fileName);

SaveScriptCode(nStates);

ConvertToM(nStates);

ConvertToC(nStates);