Appendix

EKF C code (Verified on real time platform - ARM cortex M4):

```c
#include "math.h"
#include "stdio.h"
#include "stdlib.h"

char buffM[100];
void multiply_square(double A[7][7], double B[7][7], double C[7][7]);
void multiply_rect1(double A[6][7], double B[7][7], double C[6][7]);
void multiply_rect2(double A[6][7], double B[7][6], double C[6][6]);
void multiply_rect3(double A[7][7], double B[7][6], double C[7][6]);
void multiply_rect4(double A[7][6], double B[6][6], double C[7][6]);
void multiply_rect5(double A[7][6], double B[6], double C[7]);
void multiply_rect6(double A[7][6], double B[6][7], double C[7][7]);
void invNxN();
extern void USART_PutString(uint8_t * str);

/*--- External variables ---*/
extern volatile double  P_Extern[7][7], x_Extern[7], Euler_angles_Extern[3];
extern volatile int16_t IMU_Data[9];
extern volatile double Temp_Extern[9];
volatile double x_inv[36]= {0};
volatile double y[36]= {0};

void EKF()
{

/*--- Variables used only in this function ---*/

double acc_scale[3] = {  0.951965F, 0.958475F, 0.96408F };
double acc_bias[3] = {  0.4331F,0.666F,1.672F };
double mag_bias[3] = {  -0.216F,-0.12085F,-0.42655F };
double a[3], w[3], m[3];
double ax, ay, az, wx, wy, wz, mx, my, mz, z_norm_am[6] = {0};
double q0, q1, q2, q3, qnorm, wxb, wyb, wzb;
double dt = 0.02;
int i,j; // used for loops - important always initialize to zero before using
double F[7][7] = {0};
```

```
double F_Trans[7][7] = {0};
double H[6][7] = {0};
double H_Trans[7][6] = {0};
double P_Local_Temp[6][7] = {0};
double P_Local[7][7] = {0};
double P_Temp[7][7] = {0};
double P_new[7][7] = {0};
double Q[7] = {0,0,0,0,0.0000008,0.0000008,0.0000008};  /*  Process covariance   */
double R[6] = {100.765,100.765,100.765,10000.45,10000.45,10000.45}; /*  Noise covariance
    */
double Rot_qtn[3][3] = {0};
double mat_sum = 0;
double Rm_mat[3] = {0};
double bx = 0, bz = 0;
double h_mes[6] = {0};
double y_res[6] = {0};
double S_cov[6][6] = {0};
double S_Temp[6][6] = {0};
double S_cov_inv[6][6] = {0};
double K_Gain[7][6] = {0};
double K_Temp[7][6] = {0};
double Del_x[7] = {0};

/*--- Last time step data (Prev state info) ---*/

   q0  = x_Extern[0];
   q1  = x_Extern[1];
   q2  = x_Extern[2];
   q3  = x_Extern[3];
   wxb = x_Extern[4];
     wyb = x_Extern[5];
     wzb = x_Extern[6];

/*--- Raw data units convertion and scaling ---*/

  for (i = 0; i < 3; i++) {
      a[i] = IMU_Data[i]*0.004F*9.81F*acc_scale[i] + acc_bias[i] ; //in m/s2
      w[i] = (IMU_Data[i+3] / 14.375F)*0.0174532925; //deg to rad
      m[i] = IMU_Data [i+6] * 4.35F * 0.001F + mag_bias[i]; // in gauss
```

```
    }

  ax = a[0]; ay = a[1]; az = a[2];
  wx = w[0]; wy = w[1]; wz = w[2];
  mx = m[0]; my = m[1]; mz = m[2];

#define DO_PREDICT
#define DO_UPDATE

#ifdef DO_PREDICT
/*--- Quaternion state propagation ---*/

   q0 = q0+ 0.02F *0.5F * ((-q1 * (w[0] - wxb) - q2 * (w[1] - wyb)) - q3 * (w[2] - wzb));
   q1 = q1+ 0.02F *0.5F * (( q0 * (w[0] - wxb) - q3 * (w[1] - wyb)) + q2 * (w[2] - wzb));
   q2 = q2+ 0.02F *0.5F * (( q3 * (w[0] - wxb) + q0 * (w[1] - wyb)) - q1 * (w[2] - wzb));
   q3 = q3+ 0.02F *0.5F * ((-q2 * (w[0] - wxb) + q1 * (w[1] - wyb)) + q0 * (w[2] - wzb));

/*--- Normalize quaternion ---*/

   qnorm = (double)sqrt(((q0 * q0 + q1 * q1) + q2 * q2) + q3 * q3);
   q0 = q0 / qnorm;
   q1 = q1 / qnorm;
   q2 = q2 / qnorm;
   q3 = q3 / qnorm;
   x_Extern[0] = q0;
   x_Extern[1] = q1;
   x_Extern[2] = q2;
   x_Extern[3] = q3;

#endif

#ifdef DO_UPDATE
/*--- Populate F (state) Jacobian ---*/

   F[0][0] = 1.0F;
   F[0][1] = -(dt * 0.5F) * (wx - wxb);
   F[0][2] = -(dt * 0.5F) * (wy - wyb);
   F[0][3] = -(dt * 0.5F) * (wz - wzb);
   F[0][4] = 0.5F*q1*dt;
```

```
    F[0][5] = 0.5F*q2*dt;
    F[0][6] = 0.5F*q3*dt;
    F[1][0] =  (dt * 0.5F) * (wx - wxb);
    F[1][1] =  1.0F;
    F[1][2] =  (dt * 0.5F) * (wz - wzb);
    F[1][3] = -(dt * 0.5F) * (wy - wyb);
    F[1][4] = -0.5F*q0*dt;
    F[1][5] =  0.5F*q3*dt;
    F[1][6] = -0.5F*q2*dt;
    F[2][0] =  (dt * 0.5F) * (wy - wyb);
    F[2][1] = -(dt * 0.5F) * (wz - wzb);
    F[2][2] =  1;
    F[2][3] =  (dt * 0.5F) * (wx - wxb);
    F[2][4] = -0.5F*q3*dt;
    F[2][5] = -0.5F*q0*dt;
    F[2][6] =  0.5F*q1*dt;
    F[3][0] =  (dt * 0.5F) * (wz - wzb);
    F[3][1] =  (dt * 0.5F) * (wy - wyb);
    F[3][2] = -(dt * 0.5F) * (wx - wxb);
    F[3][3] =  1;
    F[3][4] =  0.5F*q2*dt;
    F[3][5] = -0.5F*q1*dt;
    F[3][6] = -0.5F*q0*dt;

    F[4][4] = 1;
    F[5][5] = 1;
    F[6][6] = 1;

/*--- Predicted covariance estimate (P = FPF'+Q)---*/

   for (i = 0; i < 7; i++){
      for (j = 0; j < 7; j++){
        F_Trans[i][j] = F[j][i];
        P_Local[i][j] = P_Extern[i][j];
      }
    }

   multiply_square(F,P_Local,P_Local_Temp);
   multiply_square(P_Local_Temp,F_Trans,P_Local);
```

```
  for (i = 0; i < 7; i++){
      P_Local[i][i] = P_Local[i][i] + Q[i];
  }

/*--- Normalize accelerometer and magnetometer measurements ---*/

  qnorm = (double)sqrt(ax * ax + ay * ay + az * az);
  z_norm_am[0] = ax / qnorm;
  z_norm_am[1] = ay / qnorm;
  z_norm_am[2] = az / qnorm;

  qnorm = (double)sqrt(mx * mx + my * my + mz * mz);
  z_norm_am[3] = mx / qnorm;
  z_norm_am[4] = my / qnorm;
  z_norm_am[5] = mz / qnorm;

/*--- Build quaternion rotation matrix ---*/

  Rot_qtn[0][0] = 1-2*q2*q2-2*q3*q3;
  Rot_qtn[0][1] = 2*(q1*q2-q0*q3);
  Rot_qtn[0][2] = 2*(q1*q3+q0*q2);
  Rot_qtn[1][0] = 2*(q1*q2+q0*q3);
  Rot_qtn[1][1] = 1-2*q1*q1-2*q3*q3;
  Rot_qtn[1][2] = 2*(q2*q3-q0*q1);
    Rot_qtn[2][0] = 2*(q1*q3-q0*q2);
  Rot_qtn[2][1] = 2*(q2*q3+q0*q1);
  Rot_qtn[2][2] = 1-2*q1*q1-2*q2*q2;

/*--- Rotate magnetic vector into reference frame ---*/

  for (i = 0; i < 3; i++){
    mat_sum= 0;
    for (j = 0; j < 3; j++){
    mat_sum = mat_sum +Rot_qtn[i][j]*z_norm_am[j+3];
    }
    Rm_mat[i] = mat_sum;
    }
```

```c
  bx = sqrt(Rm_mat[0]*Rm_mat[0] + Rm_mat[1]*Rm_mat[1]);
  bz = Rm_mat[2];

  h_mes[0] = -2*(q1*q3 - q0*q2);
  h_mes[1] = -2*(q2*q3 + q0*q1);
  h_mes[2] = -(1-2*q1*q1-2*q2*q2);
  h_mes[3] =  bx*(1-2*q2*q2-2*q3*q3) + 2*bz*(q1*q3 - q0*q2);
  h_mes[4] =  2*bx*(q1*q2 - q0*q3)   + 2*bz*(q2*q3 + q0*q1);
  h_mes[5] =  2*bx*(q1*q3 + q0*q2)   + bz*(1-2*q1*q1-2*q2*q2);

/*--- Measurement residual ---*/

  for (i = 0; i < 6; i++){
   y_res[i] = z_norm_am[i] - h_mes[i];
  }

/*--- Populate H (measurement) Jacobian ---*/

  H[0][0] =  2*q2;
  H[0][1] = -2*q3;
  H[0][2] =  2*q0;
  H[0][3] = -2*q1;
  H[0][4] =  0;
  H[0][5] =  0;
  H[0][6] =  0;
  H[1][0] = -2*q1;
  H[1][1] = -2*q0;
  H[1][2] = -2*q3;
  H[1][3] = -2*q2;
  H[1][4] =  0;
  H[1][5] =  0;
  H[1][6] =  0;
  H[2][0] =  0;
  H[2][1] =  4*q1;
  H[2][2] =  4*q2;
  H[2][3] =  0;
  H[2][4] =  0;
  H[2][5] =  0;
  H[2][6] =  0;
```

```
H[3][0] = -2*bz*q2;
H[3][1] =  2*bz*q3;
H[3][2] = -4*q2*bx - 2*bz*q0;
H[3][3] = -4*bx*q3 + 2*bz*q1;
H[3][4] =  0;
H[3][5] =  0;
H[3][6] =  0;
H[4][0] = -2*bx*q3 + 2*bz*q1;
H[4][1] =  2*bx*q2 + 2*bz*q0;
H[4][2] =  2*bx*q1 + 2*bz*q3;
H[4][3] = -2*bx*q0 + 2*bz*q2;
H[4][4] =  0;
H[4][5] =  0;
H[4][6] =  0;
H[5][0] =  2*bx*q2;
H[5][1] =  2*bx*q3 - 4*q1*bz;
H[5][2] =  2*bx*q0 - 4*bz*q2;
H[5][3] =  2*bx*q1;
H[5][4] =  0;
H[5][5] =  0;
H[5][6] =  0;

 for (i = 0; i < 7; i++){
     for (j = 0; j < 6; j++){
      H_Trans[i][j] = H[j][i];
    }
 }

/*--- Residual covariance ---*/

 multiply_rect1(H,P_Local,P_Local_Temp);
 multiply_rect2(P_Local_Temp,H_Trans,S_cov);

 for (i = 0; i < 6; i++){
   S_cov[i][i] = S_cov[i][i]+R[i];
 }
 for (i = 0; i < 6; i++){
        for (j = 0; j < 6;j++){
            x_inv[i*6+j] = S_cov[i][j];
```

```
        }
    }

 invNxN();

  for (i = 0; i < 6; i++){
     for (j = 0; j < 6;j++){
      S_cov_inv[i][j] = y[i*6+j];
     }
  }

/*--- Kalman gain matrix computation ---*/

  multiply_rect3(P_Local,H_Trans,K_Temp);
  multiply_rect4(K_Temp,S_cov_inv,K_Gain);

/*--- Update state estimate ---*/

  multiply_rect5(K_Gain,y_res,Del_x);

  for(i=0;i<7;i++){
     x_Extern[i] = x_Extern[i] + Del_x[i];
  }

/*--- Normalize quaternion (Updated state) ---*/

  qnorm = sqrt(x_Extern[0]*x_Extern[0] + x_Extern[1]*x_Extern[1] + x_Extern[2]*x_Extern[2]
     + x_Extern[3]*x_Extern[3]);
  q0 = x_Extern[0]/qnorm;
  q1 = x_Extern[1]/qnorm;
  q2 = x_Extern[2]/qnorm;
  q3 = x_Extern[3]/qnorm;

 /*--- Normalised state information ---*/

  x_Extern[0] = q0;
  x_Extern[1] = q1;
  x_Extern[2] = q2;
  x_Extern[3] = q3;
```

```

/*--- Update estimate covariance (P_new = (I - K*H)*P;) ---*/

  multiply_rect6(K_Gain,H,P_Temp);

  for(i=0;i<7;i++){
     for(j=0;j<7;j++){
        P_Temp[i][j] = -P_Temp[i][j];
     }
  }

  for(i=0;i<7;i++){
       P_Temp[i][i] = 1 + P_Temp[i][i];
    }

  multiply_square(P_Temp, P_Local, P_new);

 /*--- Storing in External variable ---*/

  for(i=0;i<7;i++){
     for(j=0;j<7;j++){
        P_Extern[i][j] = P_new[i][j];
     }
    }

#endif

/*--- Generating Euler angles ---*/

  Euler_angles_Extern[0] = 57.2958F*atan2(2.0*(q0*q1+q2*q3),1.0-2.0*(q1*q1+q2*q2));
  Euler_angles_Extern[1] = 57.2958F*asin( 2.0*(q0*q2-q1*q3));
  Euler_angles_Extern[2] = 57.2958F*atan2(2.0*(q1*q2+q0*q3),1.0-2.0*(q2*q2+q3*q3));

/*--- Serial port variables ---*/

  sprintf(buffM, "%0.1f\t %0.1f\t %0.1f\t %0.1f\t %0.1f\t %0.1f\t %0.1f\t %0.1f\t %0.1f\t
    %0.2f\t %0.2f\t %0.2f\t",ax,ay,az, 57.2958F*wx,57.2958F*wy,57.2958F*wz,
    Euler_angles_Extern[0],Euler_angles_Extern[1],Euler_angles_Extern[2],x_Extern
    [4]*57.2958F,x_Extern[5]*57.2958F,x_Extern[6]*57.2958F);
```

```c
  USART_PutString((uint8_t *)buffM);

  sprintf(buffM, "\r\n");
  USART_PutString((uint8_t *)buffM);

}

/*--- All functions ---*/

void multiply_square(double A[7][7], double B[7][7], double C[7][7])
{
    int i, j, k;
    for (i = 0; i < 7; i++)
    {
        for (j = 0; j < 7; j++)
        {
            C[i][j] = 0;
            for (k = 0; k < 7; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}

void multiply_rect1(double A[6][7], double B[7][7], double C[6][7])
{
    int i, j, k;
    for(i=0;i<6;i++)
    {
      for(j=0;j<7;j++)
      {
      C[i][j]=0;
      for(k=0;k<7;k++)
      C[i][j]+=A[i][k]*B[k][j];
      }
    }
}
void multiply_rect2(double A[6][7], double B[7][6], double C[6][6])
{
    int i, j, k;
```

```c
    for(i=0;i<6;i++)
    {
      for(j=0;j<6;j++)
      {
      C[i][j]=0;
      for(k=0;k<7;k++)
      C[i][j]+=A[i][k]*B[k][j];
      }
    }
}
void multiply_rect3(double A[7][7], double B[7][6], double C[7][6])
{
    int i, j, k;
    for(i=0;i<7;i++)
    {
      for(j=0;j<6;j++)
      {
      C[i][j]=0;
      for(k=0;k<7;k++)
      C[i][j]+=A[i][k]*B[k][j];
      }
    }
}
void multiply_rect4(double A[7][6], double B[6][6], double C[7][6])
{
    int i, j, k;
    for(i=0;i<7;i++)
    {
      for(j=0;j<6;j++)
      {
      C[i][j]=0;
      for(k=0;k<6;k++)
      C[i][j]+=A[i][k]*B[k][j];
      }
    }
}
void multiply_rect5(double A[7][6], double B[6], double C[7])
{
    int i, j, k;
```

```
    for(i=0;i<7;i++)
    {
      C[i]=0;
      for(j=0;j<6;j++)
      {
      C[i]+=A[i][j]*B[j];
      }
    }
}
void multiply_rect6(double A[7][6], double B[6][7], double C[7][7])
{
    int i, j, k;
    for(i=0;i<7;i++)
    {
      for(j=0;j<7;j++)
      {
      C[i][j]=0;
      for(k=0;k<6;k++)
      C[i][j]+=A[i][k]*B[k][j];
      }
    }
}


void invNxN()
{
  double A[36];
  int32_t i0;
  int8_t ipiv[6];
  int32_t j;
  int32_t c;
  int32_t pipk;
  int32_t ix;
  double smax;
  int32_t k;
  double s;
  int32_t jy;
  int32_t ijA;
  int8_t p[6];
```

```
for (i0 = 0; i0 < 36; i0++) {
  y[i0] = 0.0;
  A[i0] = x_inv[i0];
}

for (i0 = 0; i0 < 6; i0++) {
  ipiv[i0] = (int8_t)(1 + i0);
}

for (j = 0; j < 5; j++) {
  c = j * 7;
  pipk = 0;
  ix = c;
  smax = fabs(A[c]);
  for (k = 2; k <= (6 - j); k++) {
    ix++;
    s = fabs(A[ix]);
    if (s > smax) {
      pipk = k - 1;
      smax = s;
    }
  }

  if (A[c + pipk] != 0.0) {
    if (pipk != 0) {
      ipiv[j] = (int8_t)((j + pipk) + 1);
      ix = j;
      pipk += j;
      for (k = 0; k < 6; k++) {
        smax = A[ix];
        A[ix] = A[pipk];
        A[pipk] = smax;
        ix += 6;
        pipk += 6;
      }
    }

    i0 = (c - j) + 6;
    for (jy = c + 1; (jy + 1) <= i0; jy++) {
```

```
        A[jy] /= A[c];
      }
    }

    pipk = c;
    jy = c + 6;
    for (k = 1; k <= (5 - j); k++) {
      smax = A[jy];
      if (A[jy] != 0.0) {
        ix = c + 1;
        i0 = (pipk - j) + 12;
        for (ijA = 7 + pipk; (ijA + 1) <= i0; ijA++) {
          A[ijA] += A[ix] * (-smax);
          ix++;
        }
      }

      jy += 6;
      pipk += 6;
    }
  }

  for (i0 = 0; i0 < 6; i0++) {
    p[i0] = (int8_t)(1 + i0);
  }

  for (k = 0; k < 5; k++) {
    if (ipiv[k] > (1 + k)) {
      pipk = p[ipiv[k] - 1];
      p[ipiv[k] - 1] = p[k];
      p[k] = (int8_t)pipk;
    }
  }

  for (k = 0; k < 6; k++) {
    y[k + (6 * (p[k] - 1))] = 1.0;
    for (j = k; (j + 1) < 7; j++) {
      if (y[j + (6 * (p[k] - 1))] != 0.0) {
        for (jy = j + 1; (jy + 1) < 7; jy++) {
```

```
            y[jy + (6 * (p[k] - 1))] -= y[j + (6 * (p[k] - 1))] * A[jy + (6 * j)];
          }
        }
      }
    }

    for (j = 0; j < 6; j++) {
      c = 6 * j;
      for (k = 5; k > -1; k += -1) {
        pipk = 6 * k;
        if (y[k + c] != 0.0) {
          y[k + c] /= A[k + pipk];
          for (jy = 0; (jy + 1) <= k; jy++) {
            y[jy + c] -= y[k + c] * A[jy + pipk];
          }
        }
      }
    }
}
```