

Appendix

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

```
1 #include "math.h"
2 #include "stdio.h"
3 #include "stdlib.h"
4
5 char buffM[100];
6 void multiply_square(double A[7][7], double B[7][7], double C[7][7]);
7 void multiply_rect1(double A[6][7], double B[7][7], double C[6][7]);
8 void multiply_rect2(double A[6][7], double B[7][6], double C[6][6]);
9 void multiply_rect3(double A[7][7], double B[7][6], double C[7][6]);
10 void multiply_rect4(double A[7][6], double B[6][6], double C[7][6]);
11 void multiply_rect5(double A[7][6], double B[6], double C[7]);
12 void multiply_rect6(double A[7][6], double B[6][7], double C[7][7]);
13 void invNxN();
14 extern void USART_PutString(uint8_t * str);
15
16 /*--- External variables ---*/
17 extern volatile double P_Extern[7][7], x_Extern[7], Euler_angles_Extern[3];
18 extern volatile int16_t IMU_Data[9];
19 extern volatile double Temp_Extern[9];
20 volatile double x_inv[36]= {0};
21 volatile double y[36]= {0};
22
23 void EKF()
24 {
25
26 /*--- Variables used only in this function ---*/
27
28 double acc_scale[3] = { 0.951965F, 0.958475F, 0.96408F };
29 double acc_bias[3] = { 0.4331F,0.666F,1.672F };
30 double mag_bias[3] = { -0.216F,-0.12085F,-0.42655F };
31 double a[3], w[3], m[3];
32 double ax, ay, az, wx, wy, wz, mx, my, mz, z_norm_am[6] = {0};
33 double q0, q1, q2, q3, qnorm, wxb, wyb, wzb;
34 double dt = 0.02;
35 int i,j; // used for loops – important always initialize to zero before using
36 double F[7][7] = {0};
```

```

37 double F_Trans[7][7] = {0};
38 double H[6][7] = {0};
39 double H_Trans[7][6] = {0};
40 double P_Local_Temp[6][7] = {0};
41 double P_Local[7][7] = {0};
42 double P_Temp[7][7] = {0};
43 double P_new[7][7] = {0};
44 double Q[7] = {0,0,0,0,0.0000008,0.0000008,0.0000008}; /* Process covariance */
45 double R[6] = {100.765,100.765,100.765,10000.45,10000.45,10000.45}; /* Noise covariance
    */
46 double Rot_qtn[3][3] = {0};
47 double mat_sum = 0;
48 double Rm_mat[3] = {0};
49 double bx = 0, bz = 0;
50 double h_mes[6] = {0};
51 double y_res[6] = {0};
52 double S_cov[6][6] = {0};
53 double S_Temp[6][6] = {0};
54 double S_cov_inv[6][6] = {0};
55 double K_Gain[7][6] = {0};
56 double K_Temp[7][6] = {0};
57 double Del_x[7] = {0};
58
59 /*--- Last time step data (Prev state info) ---*/
60
61 q0 = x_Extern[0];
62 q1 = x_Extern[1];
63 q2 = x_Extern[2];
64 q3 = x_Extern[3];
65 wxb = x_Extern[4];
66 wyb = x_Extern[5];
67 wzb = x_Extern[6];
68
69 /*--- Raw data units conversion and scaling ---*/
70
71 for (i = 0; i < 3; i++) {
72     a[i] = IMU_Data[i]*0.004F*9.81F*acc_scale[i] + acc_bias[i] ; //in m/s2
73     w[i] = (IMU_Data[i+3] / 14.375F)*0.0174532925; //deg to rad
74     m[i] = IMU_Data[i+6] * 4.35F * 0.001F + mag_bias[i]; // in gauss

```

```

75     }
76
77     ax = a[0]; ay = a[1]; az = a[2];
78     wx = w[0]; wy = w[1]; wz = w[2];
79     mx = m[0]; my = m[1]; mz = m[2];
80
81 #define DO_PREDICT
82 #define DO_UPDATE
83
84 #ifdef DO_PREDICT
85 /*--- Quaternion state propagation ---*/
86
87     q0 = q0+ 0.02F *0.5F * ((-q1 * (w[0] - wxb) - q2 * (w[1] - wyb)) - q3 * (w[2] - wzb));
88     q1 = q1+ 0.02F *0.5F * (( q0 * (w[0] - wxb) - q3 * (w[1] - wyb)) + q2 * (w[2] - wzb));
89     q2 = q2+ 0.02F *0.5F * (( q3 * (w[0] - wxb) + q0 * (w[1] - wyb)) - q1 * (w[2] - wzb));
90     q3 = q3+ 0.02F *0.5F * ((-q2 * (w[0] - wxb) + q1 * (w[1] - wyb)) + q0 * (w[2] - wzb));
91
92 /*--- Normalize quaternion ---*/
93
94     qnorm = (double)sqrt(((q0 * q0 + q1 * q1) + q2 * q2) + q3 * q3);
95     q0 = q0 / qnorm;
96     q1 = q1 / qnorm;
97     q2 = q2 / qnorm;
98     q3 = q3 / qnorm;
99     x_Extern[0] = q0;
100    x_Extern[1] = q1;
101    x_Extern[2] = q2;
102    x_Extern[3] = q3;
103
104 #endif
105
106 #ifdef DO_UPDATE
107 /*--- Populate F (state) Jacobian ---*/
108
109     F[0][0] = 1.0F;
110     F[0][1] = -(dt * 0.5F) * (wx - wxb);
111     F[0][2] = -(dt * 0.5F) * (wy - wyb);
112     F[0][3] = -(dt * 0.5F) * (wz - wzb);
113     F[0][4] = 0.5F*q1*dt;

```

```

114   F[0][5] = 0.5F*q2*dt;
115   F[0][6] = 0.5F*q3*dt;
116   F[1][0] = (dt * 0.5F) * (wx - wxb);
117   F[1][1] = 1.0F;
118   F[1][2] = (dt * 0.5F) * (wz - wzb);
119   F[1][3] = -(dt * 0.5F) * (wy - wyb);
120   F[1][4] = -0.5F*q0*dt;
121   F[1][5] = 0.5F*q3*dt;
122   F[1][6] = -0.5F*q2*dt;
123   F[2][0] = (dt * 0.5F) * (wy - wyb);
124   F[2][1] = -(dt * 0.5F) * (wz - wzb);
125   F[2][2] = 1;
126   F[2][3] = (dt * 0.5F) * (wx - wxb);
127   F[2][4] = -0.5F*q3*dt;
128   F[2][5] = -0.5F*q0*dt;
129   F[2][6] = 0.5F*q1*dt;
130   F[3][0] = (dt * 0.5F) * (wz - wzb);
131   F[3][1] = (dt * 0.5F) * (wy - wyb);
132   F[3][2] = -(dt * 0.5F) * (wx - wxb);
133   F[3][3] = 1;
134   F[3][4] = 0.5F*q2*dt;
135   F[3][5] = -0.5F*q1*dt;
136   F[3][6] = -0.5F*q0*dt;
137
138   F[4][4] = 1;
139   F[5][5] = 1;
140   F[6][6] = 1;
141
142   /*---- Predicted covariance estimate (P = FPF'+Q)----*/
143
144   for (i = 0; i < 7; i++){
145       for (j = 0; j < 7; j++){
146           F_Trans[i][j] = F[j][i];
147           P_Local[i][j] = P_Extern[i][j];
148       }
149   }
150
151   multiply_square(F, P_Local, P_Local_Temp);
152   multiply_square(P_Local_Temp, F_Trans, P_Local);

```

```

153
154     for (i = 0; i < 7; i++){
155         P_Local[i][i] = P_Local[i][i] + Q[i];
156     }
157
158 /*---- Normalize accelerometer and magnetometer measurements ----*/
159
160     qnorm = (double)sqrt(ax * ax + ay * ay + az * az);
161     z_norm_am[0] = ax / qnorm;
162     z_norm_am[1] = ay / qnorm;
163     z_norm_am[2] = az / qnorm;
164
165     qnorm = (double)sqrt(mx * mx + my * my + mz * mz);
166     z_norm_am[3] = mx / qnorm;
167     z_norm_am[4] = my / qnorm;
168     z_norm_am[5] = mz / qnorm;
169
170 /*---- Build quaternion rotation matrix ----*/
171
172     Rot_qtn[0][0] = 1-2*q2*q2-2*q3*q3;
173     Rot_qtn[0][1] = 2*(q1*q2-q0*q3);
174     Rot_qtn[0][2] = 2*(q1*q3+q0*q2);
175     Rot_qtn[1][0] = 2*(q1*q2+q0*q3);
176     Rot_qtn[1][1] = 1-2*q1*q1-2*q3*q3;
177     Rot_qtn[1][2] = 2*(q2*q3-q0*q1);
178     Rot_qtn[2][0] = 2*(q1*q3-q0*q2);
179     Rot_qtn[2][1] = 2*(q2*q3+q0*q1);
180     Rot_qtn[2][2] = 1-2*q1*q1-2*q2*q2;
181
182 /*---- Rotate magnetic vector into reference frame ----*/
183
184     for (i = 0; i < 3; i++){
185         mat_sum= 0;
186         for (j = 0; j < 3; j++){
187             mat_sum = mat_sum +Rot_qtn[i][j]*z_norm_am[j+3];
188         }
189         Rm_mat[i] = mat_sum;
190     }
191

```

```

192  bx = sqrt(Rm_mat[0]*Rm_mat[0] + Rm_mat[1]*Rm_mat[1]);
193  bz = Rm_mat[2];
194
195  h_mes[0] = -2*(q1*q3 - q0*q2);
196  h_mes[1] = -2*(q2*q3 + q0*q1);
197  h_mes[2] = -(1-2*q1*q1-2*q2*q2);
198  h_mes[3] = bx*(1-2*q2*q2-2*q3*q3) + 2*bz*(q1*q3 - q0*q2);
199  h_mes[4] = 2*bx*(q1*q2 - q0*q3) + 2*bz*(q2*q3 + q0*q1);
200  h_mes[5] = 2*bx*(q1*q3 + q0*q2) + bz*(1-2*q1*q1-2*q2*q2);
201
202  /*--- Measurement residual ---*/
203
204  for (i = 0; i < 6; i++){
205      y_res[i] = z_norm_am[i] - h_mes[i];
206  }
207
208  /*--- Populate H (measurement) Jacobian ---*/
209
210  H[0][0] = 2*q2;
211  H[0][1] = -2*q3;
212  H[0][2] = 2*q0;
213  H[0][3] = -2*q1;
214  H[0][4] = 0;
215  H[0][5] = 0;
216  H[0][6] = 0;
217  H[1][0] = -2*q1;
218  H[1][1] = -2*q0;
219  H[1][2] = -2*q3;
220  H[1][3] = -2*q2;
221  H[1][4] = 0;
222  H[1][5] = 0;
223  H[1][6] = 0;
224  H[2][0] = 0;
225  H[2][1] = 4*q1;
226  H[2][2] = 4*q2;
227  H[2][3] = 0;
228  H[2][4] = 0;
229  H[2][5] = 0;
230  H[2][6] = 0;

```

```

231 H[3][0] = -2*bz*q2;
232 H[3][1] = 2*bz*q3;
233 H[3][2] = -4*q2*bx - 2*bz*q0;
234 H[3][3] = -4*bx*q3 + 2*bz*q1;
235 H[3][4] = 0;
236 H[3][5] = 0;
237 H[3][6] = 0;
238 H[4][0] = -2*bx*q3 + 2*bz*q1;
239 H[4][1] = 2*bx*q2 + 2*bz*q0;
240 H[4][2] = 2*bx*q1 + 2*bz*q3;
241 H[4][3] = -2*bx*q0 + 2*bz*q2;
242 H[4][4] = 0;
243 H[4][5] = 0;
244 H[4][6] = 0;
245 H[5][0] = 2*bx*q2;
246 H[5][1] = 2*bx*q3 - 4*q1*bz;
247 H[5][2] = 2*bx*q0 - 4*bz*q2;
248 H[5][3] = 2*bx*q1;
249 H[5][4] = 0;
250 H[5][5] = 0;
251 H[5][6] = 0;
252
253 for (i = 0; i < 7; i++){
254     for (j = 0; j < 6; j++){
255         H_Trans[i][j] = H[j][i];
256     }
257 }
258
259 /*--- Residual covariance ---*/
260
261 multiply_rect1(H,P_Local,P_Local_Temp);
262 multiply_rect2(P_Local_Temp,H_Trans,S_cov);
263
264 for (i = 0; i < 6; i++){
265     S_cov[i][i] = S_cov[i][i]+R[i];
266 }
267 for (i = 0; i < 6; i++){
268     for (j = 0; j < 6;j++){
269         x_inv[i*6+j] = S_cov[i][j];

```

```

270     }
271 }
272
273 invNxN() ;
274
275 for (i = 0; i < 6; i++){
276     for (j = 0; j < 6; j++){
277         S_cov_inv[i][j] = y[i*6+j];
278     }
279 }
280
281 /*--- Kalman gain matrix computation ---*/
282
283 multiply_rect3(P_Local, H_Trans, K_Temp);
284 multiply_rect4(K_Temp, S_cov_inv, K_Gain);
285
286 /*--- Update state estimate ---*/
287
288 multiply_rect5(K_Gain, y_res, Del_x);
289
290 for (i=0; i<7; i++){
291     x_Extern[i] = x_Extern[i] + Del_x[i];
292 }
293
294 /*--- Normalize quaternion (Updated state) ---*/
295
296 qnorm = sqrt(x_Extern[0]*x_Extern[0] + x_Extern[1]*x_Extern[1] + x_Extern[2]*x_Extern[2]
297             + x_Extern[3]*x_Extern[3]);
298 q0 = x_Extern[0]/qnorm;
299 q1 = x_Extern[1]/qnorm;
300 q2 = x_Extern[2]/qnorm;
301 q3 = x_Extern[3]/qnorm;
302
303 /*--- Normalised state information ---*/
304
305 x_Extern[0] = q0;
306 x_Extern[1] = q1;
307 x_Extern[2] = q2;
308 x_Extern[3] = q3;

```



```

308
309 /*--- Update estimate covariance (P_new = (I - K*H)*P;) ---*/
310
311 multiply_rect6(K_Gain,H,P_Temp);
312
313 for(i=0;i<7;i++){
314     for(j=0;j<7;j++){
315         P_Temp[i][j] = -P_Temp[i][j];
316     }
317 }
318
319 for(i=0;i<7;i++){
320     P_Temp[i][i] = 1 + P_Temp[i][i];
321 }
322
323 multiply_square(P_Temp, P_Local, P_new);
324
325 /*--- Storing in External variable ---*/
326
327 for(i=0;i<7;i++){
328     for(j=0;j<7;j++){
329         P_Extern[i][j] = P_new[i][j];
330     }
331 }
332
333 #endif
334
335 /*--- Generating Euler angles ---*/
336
337 Euler_angles_Extern[0] = 57.2958F*atan2(2.0*(q0*q1+q2*q3),1.0-2.0*(q1*q1+q2*q2));
338 Euler_angles_Extern[1] = 57.2958F*asin(2.0*(q0*q2-q1*q3));
339 Euler_angles_Extern[2] = 57.2958F*atan2(2.0*(q1*q2+q0*q3),1.0-2.0*(q2*q2+q3*q3));
340
341 /*--- Serial port variables ---*/
342
343 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
%0.2f\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);

```

```

344  USART_PutString((uint8_t *)buffM);
345
346  sprintf(buffM, "\r\n");
347  USART_PutString((uint8_t *)buffM);
348
349 }
350
351 /*--- All functions ---*/
352
353 void multiply_square(double A[7][7], double B[7][7], double C[7][7])
354 {
355     int i, j, k;
356     for (i = 0; i < 7; i++)
357     {
358         for (j = 0; j < 7; j++)
359         {
360             C[i][j] = 0;
361             for (k = 0; k < 7; k++)
362                 C[i][j] += A[i][k]*B[k][j];
363         }
364     }
365 }
366
367 void multiply_rect1(double A[6][7], double B[7][7], double C[6][7])
368 {
369     int i, j, k;
370     for(i=0;i<6;i++)
371     {
372         for(j=0;j<7;j++)
373         {
374             C[i][j]=0;
375             for(k=0;k<7;k++)
376                 C[i][j]+=A[i][k]*B[k][j];
377         }
378     }
379 }
380 void multiply_rect2(double A[6][7], double B[7][6], double C[6][6])
381 {
382     int i, j, k;

```

```

383     for(i=0;i<6;i++)
384     {
385         for(j=0;j<6;j++)
386         {
387             C[i][j]=0;
388             for(k=0;k<7;k++)
389                 C[i][j]+=A[i][k]*B[k][j];
390         }
391     }
392 }
393 void multiply_rect3(double A[7][7], double B[7][6], double C[7][6])
394 {
395     int i, j, k;
396     for(i=0;i<7;i++)
397     {
398         for(j=0;j<6;j++)
399         {
400             C[i][j]=0;
401             for(k=0;k<7;k++)
402                 C[i][j]+=A[i][k]*B[k][j];
403         }
404     }
405 }
406 void multiply_rect4(double A[7][6], double B[6][6], double C[7][6])
407 {
408     int i, j, k;
409     for(i=0;i<7;i++)
410     {
411         for(j=0;j<6;j++)
412         {
413             C[i][j]=0;
414             for(k=0;k<6;k++)
415                 C[i][j]+=A[i][k]*B[k][j];
416         }
417     }
418 }
419 void multiply_rect5(double A[7][6], double B[6], double C[7])
420 {
421     int i, j, k;

```

```

422     for (i=0; i<7; i++)
423     {
424         C[i]=0;
425         for (j=0; j<6; j++)
426         {
427             C[i]+=A[i][j]*B[j];
428         }
429     }
430 }
431 void multiply_rect6(double A[7][6], double B[6][7], double C[7][7])
432 {
433     int i, j, k;
434     for (i=0; i<7; i++)
435     {
436         for (j=0; j<7; j++)
437         {
438             C[i][j]=0;
439             for (k=0; k<6; k++)
440                 C[i][j]+=A[i][k]*B[k][j];
441         }
442     }
443 }
444
445
446 void invNxN()
447 {
448     double A[36];
449     int32_t i0;
450     int8_t ipiv[6];
451     int32_t j;
452     int32_t c;
453     int32_t pipk;
454     int32_t ix;
455     double smax;
456     int32_t k;
457     double s;
458     int32_t jy;
459     int32_t ijA;
460     int8_t p[6];

```

```

461  for (i0 = 0; i0 < 36; i0++) {
462      y[i0] = 0.0;
463      A[i0] = x_inv[i0];
464  }
465
466  for (i0 = 0; i0 < 6; i0++) {
467      ipiv[i0] = (int8_t)(1 + i0);
468  }
469
470  for (j = 0; j < 5; j++) {
471      c = j * 7;
472      pipk = 0;
473      ix = c;
474      smax = fabs(A[c]);
475      for (k = 2; k <= (6 - j); k++) {
476          ix++;
477          s = fabs(A[ix]);
478          if (s > smax) {
479              pipk = k - 1;
480              smax = s;
481          }
482      }
483
484      if (A[c + pipk] != 0.0) {
485          if (pipk != 0) {
486              ipiv[j] = (int8_t)((j + pipk) + 1);
487              ix = j;
488              pipk += j;
489              for (k = 0; k < 6; k++) {
490                  smax = A[ix];
491                  A[ix] = A[pipk];
492                  A[pipk] = smax;
493                  ix += 6;
494                  pipk += 6;
495              }
496          }
497
498          i0 = (c - j) + 6;
499          for (jy = c + 1; (jy + 1) <= i0; jy++) {

```

```

500     A[jy] /= A[c];
501     }
502 }
503
504     pipk = c;
505     jy = c + 6;
506     for (k = 1; k <= (5 - j); k++) {
507         smax = A[jy];
508         if (A[jy] != 0.0) {
509             ix = c + 1;
510             i0 = (pipk - j) + 12;
511             for (ijA = 7 + pipk; (ijA + 1) <= i0; ijA++) {
512                 A[ijA] += A[ix] * (-smax);
513                 ix++;
514             }
515         }
516
517         jy += 6;
518         pipk += 6;
519     }
520 }
521
522     for (i0 = 0; i0 < 6; i0++) {
523         p[i0] = (int8_t)(1 + i0);
524     }
525
526     for (k = 0; k < 5; k++) {
527         if (ipiv[k] > (1 + k)) {
528             pipk = p[ipiv[k] - 1];
529             p[ipiv[k] - 1] = p[k];
530             p[k] = (int8_t)pipk;
531         }
532     }
533
534     for (k = 0; k < 6; k++) {
535         y[k + (6 * (p[k] - 1))] = 1.0;
536         for (j = k; (j + 1) < 7; j++) {
537             if (y[j + (6 * (p[k] - 1))] != 0.0) {
538                 for (jy = j + 1; (jy + 1) < 7; jy++) {

```

```

539     y[jy + (6 * (p[k] - 1))] -= y[j + (6 * (p[k] - 1))] * A[jy + (6 * j)];
540 }
541 }
542 }
543 }
544
545 for (j = 0; j < 6; j++) {
546     c = 6 * j;
547     for (k = 5; k > -1; k += -1) {
548         pipk = 6 * k;
549         if (y[k + c] != 0.0) {
550             y[k + c] /= A[k + pipk];
551             for (jy = 0; (jy + 1) <= k; jy++) {
552                 y[jy + c] -= y[k + c] * A[jy + pipk];
553             }
554         }
555     }
556 }
557 }

```