```
/*
-------------------------------------------------------------------------
OBJECT NAME:    gpsc.c

FULL NAME:      Corrected IRS/GPS

ENTRY POINTS:   slatc(), slonc(), svnsc(), svewc()

DESCRIPTION:    Create a corrected set of Reference variables by combining
                the IRS and GPS data.  Low Pass filter the GPS and High
                Pass filter the INS, to remove the Schuler oscillation.

INPUT:          LAT LON GLAT GLON VNS VEW GVNS GVEW ROLL GSTAT GMODE

OUTPUT:         LATC LONC VNSC VEWC

REFERENCE:      Electra Measurements of Wind and Position in HaRP.
                William Cooper.

COPYRIGHT:      University Corporation for Atmospheric Research, 1993-2006
-------------------------------------------------------------------------
*/

#include "nimbus.h"
#include "amlib.h"
#include <cassert>
#include <values.h>

static NR_TYPE
        UPFCTR  = 0.999444,
        FCTRF   = 0.997,
        TAUP    = 600.0,        // Very original value was 150.
        TAU     = 600.0,
        CDM     = 111120.0,
        ROLL_MAX = 40.0;

static const int NCF = 3;

static NR_TYPE  latc[nFeedBackTypes], lonc[nFeedBackTypes],
                vnsc[nFeedBackTypes], vewc[nFeedBackTypes];

static NR_TYPE  deltaT[nFeedBackTypes], factorp[nFeedBackTypes],
                dlat[nFeedBackTypes], dlon[nFeedBackTypes],
                dvy[nFeedBackTypes], dvx[nFeedBackTypes],
                time_duration[nFeedBackTypes], fctrf[nFeedBackTypes];

static double   h[NCF][NCF], zf[nFeedBackTypes][4][6];
static double   am[2][NCF], bm[2][NCF], c[2][NCF], cp[2][NCF];

static NR_TYPE  filter(double, double *);
static double   invert(double a[][NCF]);

static bool returnMissingValue = false;


/* -------------------------------------------------------------------- */
void initLATC(var_base *varp)
{
  float  *tmp;

  /* Get constant values from Defaults file, if available
   */
  if ((tmp = GetDefaultsValue("GPS_ROLL_MAX", varp->name)) == NULL)
```

```
  {
    sprintf(buffer, "GPS_ROLL_MAX set to %f in AMLIB function slatc.\n", ROLL_MAX);
    LogMessage(buffer);
  }
  else
    ROLL_MAX = tmp[0];

  if ((tmp = GetDefaultsValue("GPS_TAUP", varp->name)) == NULL)
  {
    sprintf(buffer, "GPS_TAUP set to %f in AMLIB function slatc.\n", TAUP);
    LogMessage(buffer);
  }
  else
    TAUP = tmp[0];

  if ((tmp = GetDefaultsValue("GPS_TAU", varp->name)) == NULL)
  {
    sprintf(buffer, "GPS_TAU set to %f in AMLIB function slatc.\n", TAU);
    LogMessage(buffer);
  }
  else
    TAU = tmp[0];

  if ((tmp = GetDefaultsValue("GPS_UPFCTR", varp->name)) == NULL)
  {
    sprintf(buffer, "GPS_UPFCTR set to %f in AMLIB function slatc.\n", UPFCTR);
    LogMessage(buffer);
  }
  else
    UPFCTR = tmp[0];

  if ((tmp = GetDefaultsValue("GPS_FCTRF", varp->name)) == NULL)
  {
    sprintf(buffer, "GPS_FCTRF set to %f in AMLIB function slatc.\n", FCTRF);
    LogMessage(buffer);
  }
  else
    FCTRF = tmp[0];

  memset((char *)zf, 0, sizeof(zf));
  memset((char *)am, 0, sizeof(am));
  memset((char *)bm, 0, sizeof(bm));
  memset((char *)h, 0, sizeof(h));
  memset((char *)c, 0, sizeof(c));
  memset((char *)cp, 0, sizeof(cp));

  for (int i = 0; i < nFeedBackTypes; ++i)
    {
    if (i == LOW_RATE_FEEDBACK)
      {
      deltaT[i] = 1.0;
      fctrf[i] = FCTRF;
      }
    else
      {
      float r = (float)(cfg.ProcessingRate() - 1) / cfg.ProcessingRate();
      fctrf[i] = FCTRF + ((1.0 - FCTRF) * r);
      deltaT[i] = 1.0 / (float)cfg.ProcessingRate();
      }

    factorp[i] = 0.002 * (1000.0 / deltaT[i]) * M_PI / TAUP;

    time_duration[i] = dvy[i] = dvx[i] = dlat[i] = dlon[i] = 0.0;
```

```
    }

}       /* END INITLATC */

/* ---------------------------------------------------------------------- */
void slatc(DERTBL *varp)
{
  int           i, j, k;
  NR_TYPE       alat, alon, vew, vns, roll, glat, glon, gvew, gvns;
  NR_TYPE       omegat, sinwt, coswt, gvnsf, gvewf, vnsf, vewf;
  long          gstat, gmode;

  double        det;

  static bool   firstTime[nFeedBackTypes] = { true, true },
                matrix_updated[nFeedBackTypes] = { false, false };
  static int    countr[nFeedBackTypes] = { 0, 0 },
                goodGPS = 0, gps_is_flat = 0;

  static NR_TYPE old_glat[nFeedBackTypes], old_glon[nFeedBackTypes];


  alat  = GetSample(varp, 0);   // IRS Lat
  alon  = GetSample(varp, 1);   // IRS Lon
  glat  = GetSample(varp, 2);   // GPS Lat
  glon  = GetSample(varp, 3);   // GPS Lon
  vns   = GetSample(varp, 4);   // IRS NS ground speed
  vew   = GetSample(varp, 5);   // IRS EW ground speed
  gvns  = GetSample(varp, 6);   // GPS NS ground speed
  gvew  = GetSample(varp, 7);   // GPS EW ground speed
  roll  = GetSample(varp, 8);   // IRS Roll
  gstat = (long)GetSample(varp, 9);     /* nSats for Tans & Garmin      */
  gmode = (long)GetSample(varp, 10);    /* GMODE or GGMODE              */

  if (isnan(gmode))
    gmode = 0;

  if (isnan(glat) || isnan(glon)|| isnan(gvns) || isnan(gvew))
    {
//    sprintf(buffer, "gpsc: GPS isnan(), nsats=%d, mode=%d, LRT=%d, SampleOffset=%d, g
oodGPS=%d", (int)gstat, (int)gmode, FeedBack == LOW_RATE_FEEDBACK, SampleOffset, goodGP
S);
//    LogStdMsg(buffer);
    gmode = 0;
    goodGPS = 0;
    }

  if (isnan(alat) || isnan(alon) || isnan(vns) || isnan(vew))
    {
//    LogStdMsg("gpsc: IRS isnan()");
    returnMissingValue = true;
    PutSample(varp, floatNAN);
    return;
    }
  else
    returnMissingValue = false;

  if (firstTime[FeedBack])
    {
    old_glat[FeedBack] = glat;
    old_glon[FeedBack] = glon;

    firstTime[FeedBack] = false;
```

```
    }


  /* If no IRS, then bail out.
   */
  if (alat == 0.0 && alon == 0.0)
    {
    latc[FeedBack] = lonc[FeedBack] = vewc[FeedBack] = vnsc[FeedBack] = 0.0;
    PutSample(varp, latc[FeedBack]);
    return;
    }


  time_duration[FeedBack] += deltaT[FeedBack];

  /* Check GPS status, only do this on the Low-rate pass.
   */
  if (FeedBack == LOW_RATE_FEEDBACK)
    {
    double dx, dy, dgps;
    dx  = glat - old_glat[FeedBack];
    dy  = glon - old_glon[FeedBack];
    dgps = dx*dx + dy*dy;

    ++goodGPS;

    /* Major hack to determine Garmin vs. Trimble GPS.  MODE/STAT vars
     * are different.  Fake the Garmin variables to mimic the Trimble.
     *  gstat = number of satellites
     *  gmode = GPS quality indication
     *      0 = fix not available
     *      1 = non-differential GPS fix available
     *      2 = differential GPS (DGPS) fix available
     *      6 = estimated
     */
    if (varp->depend[3][1] == 'G') /* This is the Garmin */
      {
      if (gstat > 2)
        gstat = 0;

      if (gmode == 1 || gmode == 2)
        {
        gmode = 4;
        }
      else              /*  gmode = 0 or gmode = 6 defined as "bad" */
        {
        gmode = 0;
        }
      }

    if (gstat == 0x0b00)        /* 3 satellites is also considered good */
      gstat = 0;


    /* Bad Positions for using GPS?
     */
    if (gstat > 0 || gmode < 4)
      {
      sprintf(buffer, "latc: GPS disabled, status reject gstat=%d, gmode=%d.",
                (int)gstat, (int)gmode);
      LogStdMsg(buffer);
      goodGPS = 0;
      }
```

```
    if (fabs(glat) > 90.0 || fabs(glon) > 180.0)
      {
      LogStdMsg("latc: GPS disabled, bad position.");
      goodGPS = 0;
      }

    if (fabs(roll) > ROLL_MAX)
      {
      sprintf(buffer, "latc: GPS disabled, ROLL_MAX of %.1f exceeded.", ROLL_MAX);
      LogStdMsg(buffer);
      goodGPS = 0;
      }

    if (dgps == 0.0)
      {
      if (++gps_is_flat > 2)    /* > 2 seconds  */
        {
        LogStdMsg("latc: GPS is flat-lined.");
        goodGPS = 0;
        }
      }
    else
      gps_is_flat = 0;
    }


  if (goodGPS < 10)     /* < 10 seconds         */
    {
    if (countr[FeedBack] > 1800 / deltaT[FeedBack])
      { /* 30 minutes of operation.     */
      if (FeedBack == LOW_RATE_FEEDBACK && matrix_updated[FeedBack])
        {
        matrix_updated[FeedBack] = false;

        h[0][1] = h[1][0];
        h[0][2] = h[2][0];
        h[1][2] = h[2][1];

        double hi[NCF][NCF];

        memcpy(hi, h, sizeof(h));

        det = invert(hi);

        /* Beware of matrix that cannot be inverted.
         */
        if (det == 0.0)
          {
          LogStdMsg("latc: GPS determinate is zero, reseting countr.");
          countr[FeedBack] = 0;
          goto label546;
          }

        /* Now multiply times am to get matrix of coefficients.
         */
        for (k = 0; k < 2; ++k)
          for (i = 0; i < NCF; ++i)
            {
            c[k][i] = cp[k][i] = 0.0;

            for (j = 0; j < NCF; ++j)
              {
```

```
              c[k][i] += hi[i][j] * am[k][j];
              cp[k][i] += hi[i][j] * bm[k][j];
              }
            }
        }

      omegat = 2.0 * M_PI * time_duration[FeedBack] / 5040.0;
      sinwt = sin(omegat);
      coswt = cos(omegat);

      /* Want to avoid sharp transition to fit because that would
       * produce spike affecting power spectra.  Instead, slowly
       * adjust from last value toward fit value, time constant of
       * about 5 minutes chosen to be small compared to Schuller
       * period but large compared to wavelengths inportant in
       * spectral analysis.
       */
/*      gpsflg = 1.0;
*/
      dvy[FeedBack] = dvy[FeedBack]*fctrf[FeedBack]+(1.0-fctrf[FeedBack])*(c[0][0]+c[0]
[1]*sinwt+c[0][2]*coswt);
      dvx[FeedBack] = dvx[FeedBack]*fctrf[FeedBack]+(1.0-fctrf[FeedBack])*(c[1][0]+c[1]
[1]*sinwt+c[1][2]*coswt);
      dlat[FeedBack]= dlat[FeedBack]*fctrf[FeedBack]+(1.0-fctrf[FeedBack])*(cp[0][0]+cp
[0][1]*sinwt+cp[0][2]*coswt);
      dlon[FeedBack]= dlon[FeedBack]*fctrf[FeedBack]+(1.0-fctrf[FeedBack])*(cp[1][0]+cp
[1][1]*sinwt+cp[1][2]*coswt);
      }
    else
      {
      /* No good fit accumulated yet, so just let last correction
       * factors slowly decay.
       */
/*      gpsflg = 0.001;
*/
      dvy[FeedBack]     *= fctrf[FeedBack];
      dvx[FeedBack]     *= fctrf[FeedBack];
      dlat[FeedBack]    *= fctrf[FeedBack];
      dlon[FeedBack]    *= fctrf[FeedBack];
      }
    }
  else
    {
    /* Good GPS comes here.
     */
    assert(!isnan(glat));
    assert(!isnan(glon));
    assert(!isnan(gvew));
    assert(!isnan(gvns));
    gvnsf       = filter((double)gvns, zf[FeedBack][0]);
    gvewf       = filter((double)gvew, zf[FeedBack][1]);
    vnsf        = filter((double)vns, zf[FeedBack][2]);
    vewf        = filter((double)vew, zf[FeedBack][3]);

    dvy[FeedBack]       = gvnsf - vnsf;
    dvx[FeedBack]       = gvewf - vewf;
    dlat[FeedBack]      = glat - alat;
    dlon[FeedBack]      = glon - alon;

    if (FeedBack == LOW_RATE_FEEDBACK) /* Only do this in the Low-rate pass */
      {
      omegat = 2.0 * M_PI * time_duration[FeedBack] / 5040.0;
      sinwt = sin(omegat);
```

```
      coswt = cos(omegat);

      am[0][0] = UPFCTR * am[0][0] + dvy[FeedBack];
      am[0][1] = UPFCTR * am[0][1] + dvy[FeedBack] * sinwt;
      am[0][2] = UPFCTR * am[0][2] + dvy[FeedBack] * coswt;
      am[1][0] = UPFCTR * am[1][0] + dvx[FeedBack];
      am[1][1] = UPFCTR * am[1][1] + dvx[FeedBack] * sinwt;
      am[1][2] = UPFCTR * am[1][2] + dvx[FeedBack] * coswt;
      bm[0][0] = UPFCTR * bm[0][0] + dlat[FeedBack];
      bm[0][1] = UPFCTR * bm[0][1] + dlat[FeedBack] * sinwt;
      bm[0][2] = UPFCTR * bm[0][2] + dlat[FeedBack] * coswt;
      bm[1][0] = UPFCTR * bm[1][0] + dlon[FeedBack];
      bm[1][1] = UPFCTR * bm[1][1] + dlon[FeedBack] * sinwt;
      bm[1][2] = UPFCTR * bm[1][2] + dlon[FeedBack] * coswt;

      /* Note: only one information matrix needed: independant of dx,dy
       */
      h[0][0] = UPFCTR * h[0][0] + 1.0;
      h[1][0] = UPFCTR * h[1][0] + sinwt;
      h[2][0] = UPFCTR * h[2][0] + coswt;
      h[1][1] = UPFCTR * h[1][1] + sinwt * sinwt;
      h[2][1] = UPFCTR * h[2][1] + sinwt * coswt;
      h[2][2] = UPFCTR * h[2][2] + coswt * coswt;

      matrix_updated[FeedBack] = true;
      ++countr[FeedBack];
      }
    }

label546:
  vnsc[FeedBack] = vns + dvy[FeedBack];
  vewc[FeedBack] = vew + dvx[FeedBack];

  if (fabs(lonc[FeedBack] - alon) > 5 || fabs(latc[FeedBack] - alat) > 5)
    {
    lonc[FeedBack] = alon;
    latc[FeedBack] = alat;
    }

  lonc[FeedBack] += vewc[FeedBack] / (CDM * cos(alat * DEG_RAD));
  latc[FeedBack] += vnsc[FeedBack] / CDM;
  lonc[FeedBack] += factorp[FeedBack] * (alon+dlon[FeedBack]-lonc[FeedBack]);
  latc[FeedBack] += factorp[FeedBack] * (alat+dlat[FeedBack]-latc[FeedBack]);

  PutSample(varp, latc[FeedBack]);

}       /* END SLATC */

/* ---------------------------------------------------------------------- */
void slonc(DERTBL *varp)
{
  if (returnMissingValue)
    PutSample(varp, floatNAN);
  else
    PutSample(varp, lonc[FeedBack]);
}

/* ---------------------------------------------------------------------- */
void svewc(DERTBL *varp)
{
  if (returnMissingValue)
    PutSample(varp, floatNAN);
  else
```

```
    PutSample(varp, vewc[FeedBack]);
}

/* ---------------------------------------------------------------- */
void svnsc(DERTBL *varp)
{
  if (returnMissingValue)
    PutSample(varp, floatNAN);
  else
    PutSample(varp, vnsc[FeedBack]);
}

/* ---------------------------------------------------------------- */
/* Version 2 (A Cooper, 5/28/92, coded by CJW 7/30/93)
 * This function returns the low-pass-filtered value of the time series
 * 'x', if the function is called sequentially for each element in x.
 * The filter is a three-pole Butterworth lowpass filter with
 * cutoff frequency (i.e., attentuation by 0.707) at T/tau where T is
 * the sampling frequency and tau is the data constant below.  (For
 * a sequence sampled once per second and tau=600, the cutoff frequency
 * is at 1/600 = (10 min)**{-1}.)  The algorithm used is described
 * in Bosic, p. 49.
 * The array zf is used to save values for the next call, to make it
 * easy to filter more than one variable.
 */
static NR_TYPE filter(double x, double zf[])
{
  static double a[nFeedBackTypes], a2[nFeedBackTypes],
                a3[nFeedBackTypes], a4[nFeedBackTypes];
  static bool   firstTime[nFeedBackTypes] = { true, true };

  if (firstTime[FeedBack])
    {
    double      b1, b2, c, e, f;

    if (FeedBack == HIGH_RATE_FEEDBACK)
      TAU *= (float)cfg.ProcessingRate();

    a[FeedBack] = 2.0 * M_PI / TAU;
    b1          = sqrt(3.0 / 2.0);
    b2          = sqrt(1.0 / 3.0);
    c           = exp(-0.5 * a[FeedBack]);
    e           = a[FeedBack] * b1;
    f           = c * (cos(e) + b2 * sin(e));
    a2[FeedBack]= a[FeedBack] * f;
    a3[FeedBack]= 2.0 * exp(-a[FeedBack] / 2.0) * cos(e);
    a4[FeedBack]= exp(-a[FeedBack]);

    firstTime[FeedBack] = false;
    }

  zf[2] = -a[FeedBack] * x + a2[FeedBack] * zf[5] + a3[FeedBack] * zf[3] - a4[FeedBack]
 * zf[4];
  zf[1] = a[FeedBack] * x + a4[FeedBack] * zf[1];
  zf[4] = zf[3];
  zf[3] = zf[2];
  zf[5] = x;

  return(zf[1] + zf[2]);

}       /* END FILTER */

/* ---------------------------------------------------------------- */
```

```
static int ludcmp(double a[NCF][NCF], int N, int indx[], double *det)
{
  int           i, imax = 0, j, k;
  double        big, dum, sum, temp;
  double        vv[NCF];

  *det = 1.0;

  for (i = 0; i < N; ++i)
    {
    big = 0.0;

    for (j = 0; j < N; ++j)
      if ((temp = fabs(a[i][j])) > big)
        big = temp;

    if (big == 0.0)
      {
      LogStdMsg("gpsc.c: Singular matrix in LUdcmp.");
      return(ERR);
    }

    vv[i] = 1.0 / big;
    }

  for (j = 0; j < N; ++j)
    {
    for (i = 0; i < j; ++i)
      {
      sum = a[i][j];

      for (k = 0; k < i; ++k)
        sum -= a[i][k] * a[k][j];

      a[i][j] = sum;
      }

    big = 0.0;

    for (i = j; i < N; ++i)
      {
      sum = a[i][j];

      for (k = 0; k < j; ++k)
        sum -= a[i][k] * a[k][j];

      a[i][j] = sum;

      if ((dum = vv[i] * fabs(sum)) >= big)
        {
        big = dum;
        imax = i;
        }
      }

    if (j != imax)
      {
      for (k = 0; k < N; ++k)
        {
        dum = a[imax][k];
        a[imax][k] = a[j][k];
        a[j][k] = dum;
        }
```

```
      *det = -(*det);
      vv[imax] = vv[j];
      }

    indx[j] = imax;

    if (a[j][j] == 0.0)
      a[j][j] = MINFLOAT;

    if (j != N-1)
      {
      dum = 1.0 / a[j][j];

      for (i = j+1; i < N; ++i)
        a[i][j] *= dum;
      }
    }

  return(OK);

}       /* END LUDCMP */

/* ---------------------------------------------------------------------- */
static void lubksub(double a[NCF][NCF], int N, int indx[], double b[])
{
  int           i, ii = -1, ip, j;
  double        sum;

  for (i = 0; i < N; ++i)
    {
    ip = indx[i];
    sum = b[ip];
    b[ip] = b[i];

    if (ii != -1)
      for (j = ii; j < i; ++j)
        sum -= a[i][j] * b[j];
    else
      if (sum != 0.0)
        ii = i;

    b[i] = sum;
    }

  for (i = N - 1; i >= 0; --i)
    {
    sum = b[i];

    for (j = i+1; j < N; ++j)
      sum -= a[i][j] * b[j];

    b[i] = sum / a[i][i];
    }
}       /* END LUBKSUB */

/* ---------------------------------------------------------------------- */
static double invert(double array[NCF][NCF])
{
  int           i, j, indx[NCF];
  double        y[NCF][NCF], col[NCF], det;

  if (ludcmp(array, NCF, indx, &det) == ERR)
```

```
    return(0.0);

  for (i = 0; i < NCF; ++i)
    det *= array[i][i];

  for (j = 0; j < NCF; ++j)
    {
    for (i = 0; i < NCF; ++i)
      col[i] = 0.0;

    col[j] = 1.0;
    lubksub(array, NCF, indx, col);

    for (i = 0; i < NCF; ++i)
      y[i][j] = col[i];
    }

  memcpy(array, y, sizeof(y));
  return(det);
}

/* END GPSC.C */
```