AP_AHRS_DCM.cpp

AP_AHRS_DCM::update(void)
更新 DCM 矩阵。利用陀螺仪、加速度计和 GPS 的信息更新矩阵，矩阵被用于控制和导航。整个过程分为 6 步，即顺序调用以下 6 个函数：

_ins.update() ：读取 IMU 数据；

matrix_update() ：使用陀螺仪的数据更新 DCM 矩阵；

normalize() ：将 DCM 矩阵标准化；

drift_correction() ：修正漂移；

check_matrix() ：检查矩阵中元素是否有坏值；

euler_angles() ：计算欧拉角。

除 _ins.update() 外，其余函数在此 cpp 中定义。

==========

1.**matrix_update**

使用陀螺仪的数据更新 DCM 矩阵。算法涉及的核心概念是一个非线性微分方程，其将方向余弦的变化率和陀螺信号联系起来。此算法计算方向余弦时不会进行任何违背微分方程非线性的近似化处理。

由于旋转是不可交换的，因此不能简单地通过积分陀螺速率信号来得到角度。考虑一个向量 r 进行旋转的运动学方程：

$$\frac{d\mathbf{r}(t)}{dt} = \omega(t) \times \mathbf{r}(t) \tag{1}$$

此方程为非线性，ω 为我们想要积分的量。若初始条件已知，则对上式积分得到

$$\mathbf{r}(t) = \mathbf{r}(0) + \int_0^t d\theta(\tau) \times \mathbf{r}(\tau) \tag{2}$$

$$d\theta(\tau) = \omega(\tau)d\tau$$

算法的策略是将此方程应用于 DCM 矩阵的行或列，将它们作为旋转的向量对待。注意到此方程中向量 $\mathbf{r}$ 和向量 ω 需是在同一个参考系中测得的；我们希望跟踪机体轴在地系中的变化，而陀螺所测量是是以机体系为参考的，故方程不能直接应用。由于旋转具有对称性，在机体参考系下地轴的旋转可等价为惯性参考系下机体轴的旋转。因此，我们改为跟踪机体系下地系坐标轴的旋转。具体做法是改变陀螺信号（即方程中积分项）的符号为负，我们可以通过交换叉乘的顺序从而使符号再变回来：

$$\mathbf{r}_{earth}(t) = \mathbf{r}_{earth}(0) + \int_0^t \mathbf{r}_{earth}(\tau) \times d\theta(\tau) \tag{3}$$

$\mathbf{r}_{earth}(t)$ 是在机体系下观测的地系任一坐标轴，即 DCM 矩阵的行向量。要按此方程进行计算，可以借助欧拉旋转定理和旋转向量的方法。将方程变回微分方程的形式：

$$\mathbf{r}_{earth}(t+dt) = \mathbf{r}_{earth}(t) + \mathbf{r}_{earth}(t) \times d\theta(t) \tag{4}$$
$$d\theta(t) = \omega(\tau)d\tau$$

由于要针对陀螺的漂移进行修正，ω 具体包括两部分：

$$\omega(t) = \omega_{gyro}(t) + \omega_{correction}(t) \tag{5}$$

$\omega_{gyro}(t)$ 是陀螺三轴上的数据，$\omega_{correction}(t)$ 是修正量（包含比例项和积分项，构成 PI 控制）。

对机体参考系下 3 个地系坐标轴按照方程 (4) 计算，可写成矩阵形式：

$$\mathbf{R}(t+dt) = \mathbf{R}(t) \begin{pmatrix} 1 & -d\theta_z & d\theta_y \\ d\theta_z & 1 & -d\theta_x \\ -d\theta_y & d\theta_x & 1 \end{pmatrix} \tag{6}$$
$$d\theta_x = \omega_x dt, \quad d\theta_y = \omega_y dt, \quad d\theta_z = \omega_z dt$$

即旋转向量的方法，其已由 Matrix.cpp 中的 rotate() 函数实现。通过以小步长（如 0.02s）重复进行以上矩阵乘法，即可实现 DCM 矩阵的实时更新。

具体代码：

```
1    _omega=_ins.get_gyro()+_omega_I;
2    _dcm_matrix.rotate((_omega+_omega_P+_omega_yaw_P)*_G_Dt);
```

_ins.get_gyro() 为 ω_{gyro}，_omega_P，_omega_yaw_P 和 _omega_I 为 $\omega_{correction}$，其中 _omega_P，_omega_yaw_P 为比例项，_omega_I 为积分项，共同构成 PI 控制。_omega_P 和 _omega_I 由函数 drift_correction() 计算，_omega_yaw_P 由函数 drift_correction_yaw() 计算。

不把比例项 _omega_P 并入 _omega 是因为 spin_rate 由 _omega.length() 计算，而 spin_rate 增大会导致 _P_gain 增大，_P_gain 又被用于计算 _omega_P 和 _omega_yaw_P，从而形成正反馈。

2.**normalize()**

将更新后的 DCM 矩阵进行标准化处理。数值计算产生的误差会逐渐积累，破坏 DCM 矩阵的正交性，因此需要对更新后的 DCM 矩阵进行处理 (orthogonality conditions renormalization)。具体步骤如下：

第一步：计算 DCM 矩阵行向量 X 和 Y 的点积。其值应为 0，因此结果可被用来作为误差的度量

$$\mathbf{X} = \begin{bmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{bmatrix} \tag{7}$$

$$error = \mathbf{X} \cdot \mathbf{Y} = \mathbf{X}^T\mathbf{Y} = [r_{xx}\ r_{xy}\ r_{xz}] \begin{bmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{bmatrix} \tag{8}$$

我们将误差平均分给两个行向量，利用交叉耦合来旋转 X 和 Y：

$$\begin{bmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{bmatrix}_{orthogonal} = \mathbf{X}_{orthogonal} = \mathbf{X} - \frac{error}{2}\mathbf{Y} \tag{9}$$

$$\begin{bmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{bmatrix}_{orthogonal} = \mathbf{Y}_{orthogonal} = \mathbf{Y} - \frac{error}{2}\mathbf{X} \tag{10}$$

这样处理后正交性误差会显著减小。把误差平均分给两个向量与全部分给一个向量相比，能够得到更小的残差。

第二步：调整行向量 **Z**，使其与 **X** 和 **Y** 正交。只需令 **Z** 等于 **X** 和 **Y** 的叉乘即可：

$$\begin{bmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{bmatrix}_{orthogonal} = \mathbf{Z}_{orthogonal} = \mathbf{X}_{orthogonal} \times \mathbf{Y}_{orthogonal} \tag{11}$$

第三步：将 DCM 矩阵每个行向量单位化。以行向量 **X** 为例，

$$\mathbf{X} = \frac{\mathbf{X}}{|\mathbf{x}|} \tag{12}$$

此步骤由 renorm() 函数实现，主要代码为

```
    renorm_val = 1.0f / a.length();
    result = a * renorm_val;
```

normalize() 函数具体代码：

```
    error = _dcm_matrix.a * _dcm_matrix.b;
    t0 = _dcm_matrix.a - (_dcm_matrix.b * (0.5f * error));
    t1 = _dcm_matrix.b - (_dcm_matrix.a * (0.5f * error));
    t2 = t0 % t1;       // vector cross product

    if (!renorm(t0, _dcm_matrix.a) ||
        !renorm(t1, _dcm_matrix.b) ||
        !renorm(t2, _dcm_matrix.c)) {
        // Our solution is blowing up and we will force back
        // to last euler angles
        reset(true);
    }
```

3.**drift_correction()**

计算陀螺仪的漂移量,将误差传递给 PI 控制器得到修正量 _omega_I,_omega_P 和 _omega_yaw_P，用于反馈补偿到陀螺仪输出的角速率向量。具体分为两个通道：

1) Yaw：以磁罗盘或 GPS 数据为参考，计算陀螺仪偏航角速率的漂移。此部分通过调用函数 drift_correction_yaw(void) 实现，具体分析点我。

```
_omega_yaw_P.z = error_z * _P_gain(spin_rate) * _kp_yaw,
_omega_I_sum.z += error_z * _ki_yaw * yaw_deltat,
```

2) Roll&Pitch：以加速度计（或空速管）数据为参考，计算陀螺仪滚转角和俯仰角速率的漂移；

```
_omega_P = error * _P_gain(spin_rate) * _kp;
_omega_I_sum += error * _ki * _ra_deltat;
_omega_I += _omega_I_sum;
```

即：

$$\begin{aligned} \omega_P &= K_P \cdot error \cdot P_{gain} \\ \omega_{yawP} &= K_{yawP} \cdot error_z \cdot P_{gain} \\ \omega_I &= \int_{t_1}^{t_2} K_I \cdot error \cdot d\tau_{ra} + \int_{t_1}^{t_2} K_{yawI} \cdot error_z \cdot d\tau_{yaw} \\ \omega_{Correction} &= \omega_P + \omega_{yawP} + \omega_I \end{aligned} \tag{13}$$

最后将此修正向量补偿到陀螺仪信号，形成反馈控制回路。

Yaw 通道的计算是通过调用 drift_correction_yaw() 函数实现的，下面分析 Roll&Pitch 通道。在机体参考系下，加速度计所得加速度向量（绝对加速度）是重力加速度与飞行器相对惯性系的加速度（下面称为地面加速度）的矢量和：

$$\mathbf{A}_b(t) = \mathbf{g}_b(t) + \mathbf{a}_b(t) \tag{14}$$

$\mathbf{A}_b(t)$ 为加速度计的输出，$\mathbf{g}_b(t)$ 是在机体参考系下的重力加速度，$\mathbf{a}_b(t)$ 是在机体参考系下的地面加速度。

利用旋转矩阵的估计值将上式转换到惯性系：

$$\hat{\mathbf{R}}(t) \cdot \mathbf{A}_b(t) = \mathbf{g}_e + \mathbf{a}_e(t) \tag{15}$$

$\hat{\mathbf{R}}(t)$ 是由陀螺仪数据得到的旋转矩阵的估计值，$\mathbf{g}_e$ 和 $\mathbf{a}_e(t)$ 是惯性参考系下的重力加速度和地面加速度。

如果旋转矩阵的估计值是正确的，则上式为一恒等式。但是由于陀螺漂移的存在，实际情况不可能准确满足上式。等号两侧的两个向量的叉乘可用来指示陀螺的漂移量。为了提高精度，将上式积分：

$$\int_{t_1}^{t_2} \hat{\mathbf{R}}_\tau \cdot \mathbf{A}_b(\tau) \cdot d\tau = (t_2 - t_1) \cdot \mathbf{g}_e + (\mathbf{V}_e(t_2) - \mathbf{V}_e(t_1)) \tag{16}$$

积分后，我们可以从使用加速度信息改为使用速度信息。等号左侧是旋转矩阵和加速度计输出的乘积之和，等号右侧 $\mathbf{g}_e$ 重力加速度是已知的常值向量，第二项是由 GPS 得到的两个时刻的速度矢量差。将等号两侧两个向量叉乘得到误差旋转向量：

$$\mathbf{error}_{earth}(t_2) = \left(\int_{t_1}^{t_2} \hat{\mathbf{R}}(\tau) \cdot \mathbf{A}_b(\tau) \cdot d\tau \right) \times ((t_2 - t_1) \cdot \mathbf{g}_e + (\mathbf{V}_e(t_2) - \mathbf{V}_e(t_1))) \tag{17}$$

为得到与时间间隔不相关的误差表达式，将上式变形为

$$\mathbf{error}_{earth}(t_2) = \frac{\left[\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \hat{\mathbf{R}}(\tau) \cdot \mathbf{A}_b(\tau) \cdot d\tau \right] \times \left[\mathbf{g}_e + \frac{(\mathbf{V}_e(t_2) - \mathbf{V}_e(t_1))}{t_2 - t_1} \right]}{\left| \mathbf{g}_e + \frac{(\mathbf{V}_e(t_2) - \mathbf{V}_e(t_1))}{t_2 - t_1} \right|} \tag{18}$$

式中将加速度参考向量 $(\mathbf{g}_e + \mathbf{a}_e)$ 进行了标准化。最后，将误差向量转换到机体参考系中，从而可直接作为 PI 控制器的输入量：

$$\mathbf{error}_{body} = \hat{\mathbf{R}}^T(t_2) \cdot \mathbf{error}_{earth}(t_2) \tag{19}$$

主要代码分析：

1. 调用 drift_correction_yaw() 函数计算 yaw 通道的漂移量以及相应 PI 控制器修正项：

```
drift_correction_yaw();
```

2. 计算式 (16) 等号左侧的向量 $\int_{t_1}^{t_2} \hat{\mathbf{R}}_\tau \cdot \mathbf{A}_b(\tau) \cdot d\tau$：

```
// rotate accelerometer values into the earth frame
_accel_ef = _dcm_matrix * _ins.get_accel();
// integrate the accel vector in the earth frame between
    GPS readings
_ra_sum += _accel_ef * deltat;
// keep a sum of the deltat values, so we know how much
    time we have integrated over
_ra_deltat += deltat;
```

3. 计算式 (16) 等号右侧的向量中的速度向量。如果没有 GPS 或 GPS 信号差，则使用空速信息来计算速度向量：

```
if (!have_gps() ||
    _gps->status() < GPS::GPS_OK_FIX_3D ||
    _gps->num_sats < _gps_minsats) {
    ...
    velocity = _dcm_matrix.colx() * airspeed;
    velocity += _wind;
```

否则，使用 GPS 信息更新速度向量：

```
        velocity = Vector3f(_gps->velocity_north(), _gps->
            velocity_east(), _gps->velocity_down());
        ...
        // keep last airspeed estimate for dead-reckoning
            purposes
        Vector3f airspeed = velocity - _wind;
        airspeed.z = 0;
        _last_airspeed = airspeed.length();
```

4. 更新飞行器的位置。如果有 GPS，直接使用其数据更新位置：

```
        _last_lat = _gps->latitude;
        _last_lng = _gps->longitude;
        _position_offset_north = 0;
        _position_offset_east = 0;
```

否则，使用上一时刻的位置和第 3 步得到的速度向量进行航迹推算：

```
        _position_offset_north += velocity.x * _ra_deltat;
        _position_offset_east  += velocity.y * _ra_deltat;
```

5. 计算两个向量的叉乘，得到误差向量，并转换到机体系：

```
    Vector3f GA_e;
    GA_e = Vector3f(0, 0, -1.0f);
    ...
    float v_scale = gps_gain.get()/(_ra_deltat*GRAVITY_MSS);
    Vector3f vdelta = (velocity - _last_velocity) * v_scale;
    GA_e += vdelta;
    GA_e.normalize();
    ...
    _ra_sum /= (_ra_deltat * GRAVITY_MSS);

    // get the delayed ra_sum to match the GPS lag
    Vector3f GA_b;
    if (using_gps_corrections) {
        GA_b = ra_delayed(_ra_sum);
    } else {
        GA_b = _ra_sum;
    }
    GA_b.normalize();
    Vector3f error = GA_b % GA_e;

    if (use_compass()) {
        if (have_gps() && gps_gain == 1.0f) {
```

```
23              error.z *= sinf(fabsf(roll));
24          } else {
25              error.z = 0;
26          }
27      // convert the error term to body frame
28      error = _dcm_matrix.mul_transpose(error);
```

其中，GA_b 和 GA_e 分别是式 (16) 等号左侧和右侧的向量除以 (_ra_deltat * GRAVITY_MSS) 然后标准化得到的向量。GA_b 为单位加速度向量估计值，GA_e 为单位加速度参考向量。其叉乘即误差向量。注意此通道只需得到滚转和俯仰的漂移量，但叉乘得到的误差向量实际上也包含了偏航的漂移量（error.z），而其已在调用 drift_correction_yaw() 函数计算 yaw 通道漂移时算过一次并计入了 _omega_I_sum.z 中，此处如果保留 error.z，则下面计算 _omega_I 时会再将其再次计入，从而发生了重复，会产生过大的控制量。因此，21-26 行对误差的 z 轴分量进行处理，只有滚转角较大时（因为此时偏航角误差转换到机体系后在 z 轴上的分量较小?）使其发生作用。

6. 计算 _omega_P 和 _omega_I。使用 constrain_float() 函数对积分项进行限幅是为了防止积分饱和：

```
1       // base the P gain on the spin rate
2       float spin_rate = _omega.length();
3       _omega_P = error * _P_gain(spin_rate) * _kp;
4       // accumulate some integrator error
5       if (spin_rate < ToRad(SPIN_RATE_LIMIT)) {
6           _omega_I_sum += error * _ki * _ra_deltat;
7           _omega_I_sum_time += _ra_deltat;
8       }
9       if (_omega_I_sum_time >= 5) {
10          // limit the rate of change of omega_I to the hardware
11          // reported maximum gyro drift rate. This ensures that
12          // short term errors don't cause a buildup of omega_I
13          // beyond the physical limits of the device
14          float change_limit = _gyro_drift_limit *
                _omega_I_sum_time;
15          _omega_I_sum.x = constrain_float(_omega_I_sum.x, -
                change_limit, change_limit);
16          _omega_I_sum.y = constrain_float(_omega_I_sum.y, -
                change_limit, change_limit);
17          _omega_I_sum.z = constrain_float(_omega_I_sum.z, -
                change_limit, change_limit);
18          _omega_I += _omega_I_sum;
19          _omega_I_sum.zero();
```

```
20          _omega_I_sum_time = 0;
21      }
```

4.**check_matrix()**

检查 DCM 矩阵是否有坏值。

如果 _dcm_matrix.is_nan(), 则调用 reset() 复位;

如果 $|\_dcm\_matrix.c.x| \geq 1$，则调用 normalize() 进行标准化处理;

如果处理后 _dcm_matrix.is_nan() 或 |_dcm_matrix.c.x|>10，则调用 reset() 复位。

5.**euler_angles()**

计算欧拉角。_trim 用于补偿传感器与机体坐标系之间的误差（即飞行器静置于地面时由加速度传感器得到的俯仰角和滚转角）。其由 AP_InertialSensor.cpp 中的函数 _calculate_trim() 计算。

具体代码：

```
1     _body_dcm_matrix = _dcm_matrix;
2     _body_dcm_matrix.rotateXYinv(_trim);
3     _body_dcm_matrix.to_euler(&roll, &pitch, &yaw);
4
5     roll_sensor     = degrees(roll)  * 100;
6     pitch_sensor    = degrees(pitch) * 100;
7     yaw_sensor      = degrees(yaw)   * 100;
8
9     if (yaw_sensor < 0)
10        yaw_sensor += 36000;
```

此 cpp 中其它重要函数：

drift_correction_yaw(void) 函数：（返回 drift_correction() 函数）

使用 GPS 或磁罗盘数据作为参考向量，来消除陀螺仪在偏航方向的漂移。磁罗盘所得机头指向 (heading) 或 GPS 所得航向向量 (COG,course over ground)(仅限无侧滑时，此时航向与机头指向同向) 与 IMU 滚转轴 (X) 在水平面上的投影向量的夹角可用来衡量陀螺在偏航轴上的漂移量。修正量等于 DCM 矩阵的列向量 X 与航向向量的叉乘在 Z 轴上的分量。首先得到航向向量：

$$COGX = cos(cog), \quad COGY = sin(cog) \tag{20}$$

cog 为航向角。然后可得偏航角修正量：

$$YawCorrectionGround = r_{xx}COGY - r_{yx}COGX \tag{21}$$

式 (21) 得到的修正量是在惯性参考系下的，为修正陀螺漂移，需将其转化到机体参考系。做法是：将其乘以 DCM 矩阵的行向量 Z：

$$YawCorrectionPlane = YawCorrectionGround \begin{bmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{bmatrix} \tag{22}$$

最终的修正量只有机体坐标系 z 轴上的分量 (r_{zz})。

主要代码分析：

1. 如果使用了罗盘，则使用其数据计算陀螺仪在偏航方向的漂移：

```
1   if (use_compass()){
2           if (!_flags.have_initial_yaw && _compass->read()) {
3               float heading = _compass->calculate_heading(
                    _dcm_matrix);
4               _dcm_matrix.from_euler(roll, pitch, heading);
5               _omega_yaw_P.zero();
6               _flags.have_initial_yaw = true;
7           }
8           new_value = true;
9           yaw_error = yaw_error_compass();
10  }
```

注意代码第 3 行通过 calculate_heading() 函数计算出机头方向与正北的夹角 heading，第 4 行使用 heading 代替 yaw 更新了 DCM 矩阵。即 heading 被作为 yaw 的初始值，因为陀螺仪只能积分得到 yaw 的变化量，而无法给出初始量。最后调用函数 yaw_error_compass() 计算偏航误差，其主要代码为：

```
1   const Vector3f &mag = _compass->get_field();
2   Vector2f rb = _dcm_matrix.mulXY(mag);
3   rb.normalize();
4   ...
5   if( _last_declination != _compass->get_declination() ) {
6       _last_declination = _compass->get_declination();
7       _mag_earth.x = cosf(_last_declination);
8       _mag_earth.y = sinf(_last_declination);
9   }
10  // calculate the error term in earth frame
11  // calculate the Z component of the cross product of rb and
        _mag_earth
12  return rb % _mag_earth;
```

代码中 _last_declination 为地磁偏角，_mag_earth 为地磁场在水平面内的投影向量。磁罗盘三轴与机体轴一致，因此，正确的 DCM 乘以罗盘所得向量

mag 可以得到惯性参考系下的地磁场向量，其与 _mag_earth 平行。如果由陀螺仪得到的 yaw 是正确的，则第 2 行中的 rb 与 _mag_earth 平行，rb 叉乘 _mag_earth 等于 0；否则，_dcm_matrix 不能将向量 mag 转换为惯性参考系下的地磁场向量，此时 rb 与 _mag_earth 的夹角即陀螺仪所得 yaw 与真实值的误差。

2. 未使用罗盘，如果已假设飞行器沿 x 轴向前飞行（即无侧滑），且备有 GPS，则使用 GPS 数据计算陀螺仪在偏航方向的漂移：

```
1       else if (_flags.fly_forward && have_gps()) {
2         if (_gps->last_fix_time != _gps_last_update &&
3             _gps->ground_speed_cm >= GPS_SPEED_MIN) {
4             yaw_deltat = (_gps->last_fix_time -
                  _gps_last_update) * 1.0e-3f;
5             _gps_last_update = _gps->last_fix_time;
6             new_value = true;
7             float gps_course_rad = ToRad(_gps->ground_course_cd
                   * 0.01f);
8             float yaw_error_rad = wrap_PI(gps_course_rad - yaw)
                  ;
9             yaw_error = sinf(yaw_error_rad);
```

在无侧滑的条件下，航向角 course 即偏航角 yaw 的真实值，其与由陀螺得到的 yaw 的差值即误差角度。

当满足以下三种情况之一时，使用 GPS 得到的航向角重置偏航角 yaw：
1) 飞行器速度达到了 GPS_SPEED_MIN 且从未得到偏航角信息；
2) 上一次从 GPS 得到偏航角信息已经过 20 秒，则此时陀螺仪已存在较大漂移；
3) 飞行器速度大于 3*GPS_SPEED_MIN(9m/s)，且偏航角误差大于 60 度。

```
1           if (!_flags.have_initial_yaw ||
2                yaw_deltat > 20 ||
3                (_gps->ground_speed_cm >= 3*GPS_SPEED_MIN &&
                      fabsf(yaw_error_rad) >= 1.047f)) {
4                // reset DCM matrix based on current yaw
5                _dcm_matrix.from_euler(roll, pitch,
                      gps_course_rad);
6                _omega_yaw_P.zero();
7                _flags.have_initial_yaw = true;
8                yaw_error = 0;
```

3. 计算偏航角速度的比例修正项 _omega_yaw_P 和 _omega_I_sum.z(用于计算角速度的积分修正项 omega_I 的 z 轴分量)，公式在第 5 和 11 行:

```
1     //将误差向量由惯性系转换到机体系(公式23)
2     float error_z = _dcm_matrix.c.z * yaw_error;
```

```
3
4       float spin_rate = _omega.length();
5       _omega_yaw_P.z = error_z * _P_gain(spin_rate) * _kp_yaw;
6       if (_flags.fast_ground_gains) {
7           _omega_yaw_P.z *= 8;
8       }
9
10      if (yaw_deltat < 2.0f && spin_rate < ToRad(SPIN_RATE_LIMIT)
            ) {
11          _omega_I_sum.z += error_z * _ki_yaw * yaw_deltat;
12      }
13      _error_yaw_sum += fabsf(yaw_error);
14      _error_yaw_count++;
```

ra_delayed() 函数:

由于 GPS 存在延迟 (AHRS_GPS_DELAY)，在 drift_correction() 函数中求俯仰滚转通道的漂移时对两个加速度向量进行叉乘，这两个向量分别用到 GPS 的数据和加速度计的数据。为使数据同步，需给直接由加速度计得到的加速度施加相应的延迟。此函数返回经过延迟的加速度向量。

参考资料:

1.Direction Cosine Matrix IMU: Theory, William Premerlani and Paul Bizard
2.Roll-Pitch Gyro Drift Compensation, William Premerlani.
http://gentlenav.googlecode.com/files/RollPitchDriftCompensation.pdf

2015.7.29

方酉