多旋翼姿态控制 mc_att_control 源码简单分析

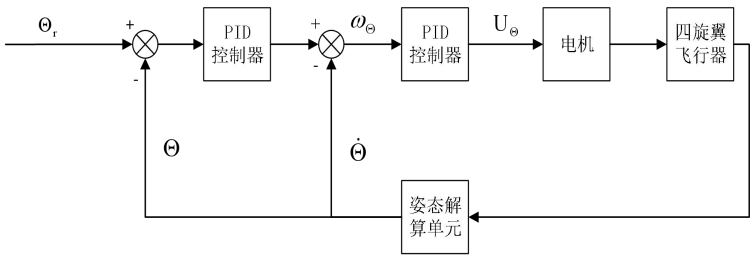
Better | gen_better@163.com

备注：源码对应 px4 v1.5.5 版本，源码链接为

https://github.com/PX4/Firmware/blob/v1.5.5/src/modules/mc_att_control/mc_att_control_main.cpp

一．姿态控制框架

姿态控制要实现的是飞机从一个姿态达到期望的姿态，控制分为内外两环串级 PID。



外环：作用于角度差,产生期望的角速度。

```
765
766     /* calculate angular rates setpoint */
767     _rates_sp = _params.att_p.emult(e_R);
768
```

内环：作用于角速度差，产生控制量。

```
816     /* angular rates error */
817     math::Vector<3> rates_err = _rates_sp - rates;
818
819     _att_control =  _params.rate_p.emult(rates_err * tpa) +
820                     _params.rate_d.emult(_rates_prev - rates) / dt +
821                     _rates_int +
822                     _params.rate_ff.emult(_rates_sp);
823
```

控制量 4 个，合力以及三个翻转的力矩。合力决定每个电机的基础转速，在此基础上叠上姿态，最终产生每个电机的转速。即经过混控器求出每个电机转速大小。

如：+字型举例

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} F_1+F_2+F_3+F_4 \\ F_4-F_2 \\ F_3-F_1 \\ F_2+F_4-F_3-F_1 \end{bmatrix} = \begin{bmatrix} k_t\sum_{i=1}^{4}\omega_i^2 \\ k_t\left(\omega_4^2-\omega_2^2\right) \\ k_t\left(\omega_3^2-\omega_1^2\right) \\ k_d\left(\omega_1^2-\omega_2^2+\omega_3^2-\omega_4^2\right) \end{bmatrix},$$

式中，U_1 为垂直速度控制量，U_2 为翻滚输入控制量，U_3 为俯仰控制量，U_4 为偏航控制量. ω 为旋翼转速，F_i 为旋翼所受拉力.

那么在这个过程中，需要注意的有两点：

1、外环中姿态误差怎么表示

2、内环控制量抗饱和的处理

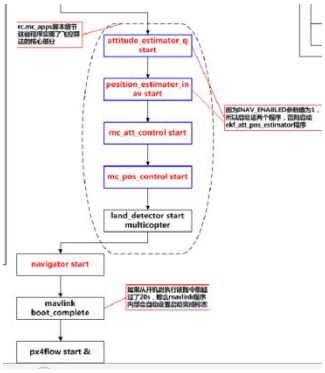
二．文件函数的入口

下面的 px4 入口函数的截图：

```
1076 int mc_att_control_main(int argc, char *argv[])
1077 {
1078     if (argc < 2) {
1079         warnx("usage: mc_att_control {start|stop|status}");
1080         return 1;
1081     }
1082
1083     if (!strcmp(argv[1], "start")) {
1084
1085         if (mc_att_control::g_control != nullptr) {
1086             warnx("already running");
1087             return 1;
1088         }
1089
1090         mc_att_control::g_control = new MulticopterAttitudeControl;
1091
1092         if (mc_att_control::g_control == nullptr) {
1093             warnx("alloc failed");
1094             return 1;
1095         }
1096
1097         if (OK != mc_att_control::g_control->start()) {
1098             delete mc_att_control::g_control;
1099             mc_att_control::g_control = nullptr;
1100             warnx("start failed");
1101             return 1;
1102         }
1103
1104         return 0;
1105     }
```

首先检查参数：start|stop|status

这个参数是哪里传进来，这个参数是启动脚本中参数进来的。



Mc_att_control 的启动在启动代码 rc.mc_app 里面有详细的说明。

```
18
19# LPE
20if param compare SYS_MC_EST_GROUP 1
21then
22    # Try to start LPE. If it fails, start EKF2 as a default
23    # Unfortunately we do not build it on px4fmu-v2 duo to a limited flash.
24    if attitude_estimator_q start
25    then
26        local_position_estimator start
27    else
28        ekf2 start
29    fi
30fi
31
32# EKF
33if param compare SYS_MC_EST_GROUP 2
34then
35    ekf2 start
36fi
37#---------------------------------------
38
39mc_att_control start
40
41mc_pos_control start
42
43#
44# Start Land Detector
45#
46land_detector start multicopter
47
```

再者传参 start 后，源码会进行判断

if (mc_att_control::g_control != nullptr)是否进程已经在运行了，如果没有将新建一个进程。

mc_att_control::g_control = new MulticopterAttitudeControl;

new 类似于 C 语言中的 malloc，对变量进行内存分配的， 即对姿态控制

过程中使用到的变量赋初值。

跳转到 start 函数，并在 start 函数创建姿态控制的进程。

```
if (OK != mc_att_control::g_control->start()) {
    delete mc_att_control::g_control;
    mc_att_control::g_control = nullptr;
    warnx("start failed");
    return 1;
}
```

```
1055int
1056MulticopterAttitudeControl::start()
1057{
1058    ASSERT(_control_task == -1);
1059
1060    /* start the task */
1061    _control_task = px4_task_spawn_cmd("mc_att_control",
1062                       SCHED_DEFAULT,
1063                       SCHED_PRIORITY_MAX - 5,
1064                       1500,
1065                       (px4_main_t)&MulticopterAttitudeControl::task_main_trampoline,
1066                       nullptr);
1067
1068    if (_control_task < 0) {
1069        warn("task start failed");
1070        return -errno;
1071    }
1072
1073    return OK;
1074}
```

其中上面有个封装了 nuttx 自带的生成 task 的任务创建函数（ 他把优先级什么的做了重新的 define， 这么做是便于代码阅读）： px4_task_spawn_cmd()， 注意它的用法。

其函数原型是 px4_task_t px4_task_spawn_cmd(const char *name, int scheduler, int priority, int stack_size, px4_main_t entry,char *const argv[])

第一个参数是 namespace， 第二个参数是选择调度策略，第三个是任务优先级，第四个是任务的栈空间大小，第五个是任务的入口函数， 最后一个一般是 null。

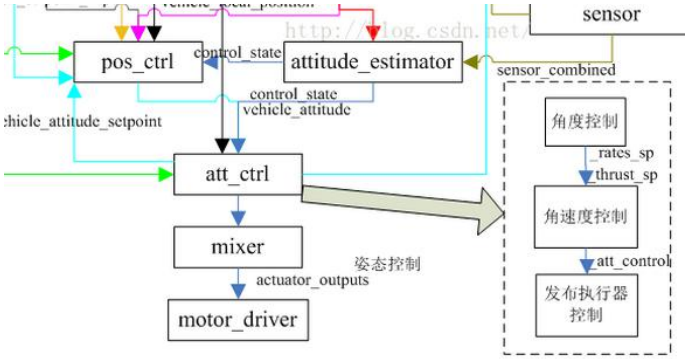
```
844 void
845 MulticopterAttitudeControl::task_main_trampoline(int argc, char *argv[])
846 {
847     mc_att_control::g_control->task_main();
848 }
849
850 void
851 MulticopterAttitudeControl::task_main()
852 {
853
854     /*
855      * do subscriptions
856      */
857     _v_att_sp_sub = orb_subscribe(ORB_ID(vehicle_attitude_setpoint));
858     _v_rates_sp_sub = orb_subscribe(ORB_ID(vehicle_rates_setpoint));
859     _ctrl_state_sub = orb_subscribe(ORB_ID(control_state));
860     _v_control_mode_sub = orb_subscribe(ORB_ID(vehicle_control_mode));
861     _params_sub = orb_subscribe(ORB_ID(parameter_update));
```

最终到 task_main()姿态控制功能代码正在开始执行的地方，下面我们将从此说起。

三．功能代码的实现

1、系统的数据流

如果暂时把姿态控制看做一个黑匣子，那么这个黑匣子是什么数据流入，什么数据流出，中间实现什么功能。



输入数据流主要是：

_v_att_sp_sub = orb_subscribe(ORB_ID(vehicle_attitude_setpoint));

输出的数据流主要是：

```
_controller_status.timestamp = hrt_absolute_time();

if (!_actuators_0_circuit_breaker_enabled) {
    if (_actuators_0_pub != nullptr) {

        orb_publish(_actuators_id, _actuators_0_pub, &_actuators);
        perf_end(_controller_latency_perf);

    } else if (_actuators_id) {
        _actuators_0_pub = orb_advertise(_actuators_id, &_actuators);
    }

}
```

中间过程就是如何从现在姿态达到期望的姿态的控制过程：

```
765
766     /* calculate angular rates setpoint */
767     _rates_sp = _params.att_p.emult(e_R);
768
815
816     /* angular rates error */
817     math::Vector<3> rates_err = _rates_sp - rates;
818
819     _att_control =  _params.rate_p.emult(rates_err * tpa) +
820                     _params.rate_d.emult(_rates_prev - rates) / dt +
821                     _rates_int +
822                     _params.rate_ff.emult(_rates_sp);
823
```

下面讲展开阐述这 4 个过程：数据的订阅，外环的处理，内环的处理，数据的发布

2、数据的订阅

```
/*
 * do subscriptions
 */
_v_att_sp_sub = orb_subscribe(ORB_ID(vehicle_attitude_setpoint));
_v_rates_sp_sub = orb_subscribe(ORB_ID(vehicle_rates_setpoint));
_ctrl_state_sub = orb_subscribe(ORB_ID(control_state));
_v_control_mode_sub = orb_subscribe(ORB_ID(vehicle_control_mode));
_params_sub = orb_subscribe(ORB_ID(parameter_update));
_manual_control_sp_sub = orb_subscribe(ORB_ID(manual_control_setpoint));
_armed_sub = orb_subscribe(ORB_ID(actuator_armed));
_vehicle_status_sub = orb_subscribe(ORB_ID(vehicle_status));
_motor_limits_sub = orb_subscribe(ORB_ID(multirotor_motor_limits));
_battery_status_sub = orb_subscribe(ORB_ID(battery_status));

/* initialize parameters cache */
parameters_update();
```

紧随上面的代码就是参数数据的获取， parameters 主要就是我们前期定义的感兴趣的数据， 在姿态控制中的这些数据都是私有数据（ private）， 比如 roll、 pitch、 yaw 以及与它们对应的 PID 参数。 注意区分_params_handles 和_params 这两种数据结构（ struct 类型） 。

```
465 int
466 MulticopterAttitudeControl::parameters_update()
467 {
468     float v;
469
470     float roll_tc, pitch_tc;
471
472     param_get(_params_handles.roll_tc, &roll_tc);
473     param_get(_params_handles.pitch_tc, &pitch_tc);
474
475     /* roll gains */
476     param_get(_params_handles.roll_p, &v);
477     _params.att_p(0) = v * (ATTITUDE_TC_DEFAULT / roll_tc);
478     param_get(_params_handles.roll_rate_p, &v);
479     _params.rate_p(0) = v * (ATTITUDE_TC_DEFAULT / roll_tc);
480     param_get(_params_handles.roll_rate_i, &v);
481     _params.rate_i(0) = v;
482     param_get(_params_handles.roll_rate_d, &v);
483     _params.rate_d(0) = v * (ATTITUDE_TC_DEFAULT / roll_tc);
484     param_get(_params_handles.roll_rate_ff, &v);
485     _params.rate_ff(0) = v;
486
487     /* pitch gains */
488     param_get(_params_handles.pitch_p, &v);
489      params.att p(1) = v * (ATTITUDE TC DEFAULT / pitch tc);
```

这一部分代码涉及文件 mc_att_control_params.c 默认参数获取。

其中 param_get()函数比较重要，特别是内部使用的 lock 和 unlock 的使用（主要就是通过 sem 信号量控制对某一数据的互斥访问）。

```
498 int
499 param_get(param_t param, void *val)
500 {
501     int result = -1;
502
503     param_lock();
504
505     const void *v = param_get_value_ptr(param);
506
507     if (val != NULL) {
508         memcpy(val, v, param_size(param));
509         result = 0;
510     }
511
512     param_unlock();
513
514     return result;
515 }
```

到目前为止已经拿到了很多数据，但其中对于数据

_ctrl_state_sub=orb_subscribe(ORB_ID(control_state));

系统采用阻塞等待的方式获取数据，对于其他数据则是用 check 的方式检查更新，为什么地位如此不同。因为在姿态控制里面，最基本的数据是飞机当前的姿态，control_state 主题中包含飞机当前的姿态数据，每次当前姿态数据发生变化，姿态控制都需要重新运行，如果拿不到最新的姿态的数据，姿态控制的运行也没多大意义，所以阻塞等待“control_state”这

个当前姿态信息关键数据。

```
871     /* wakeup source: vehicle attitude */
872     px4_pollfd_struct_t fds[1];
873
874     fds[0].fd = _ctrl_state_sub;
875     fds[0].events = POLLIN;
876
877     while (!_task_should_exit) {
878
879         /* wait for up to 100ms for data */
880         int pret = px4_poll(&fds[0], (sizeof(fds) / sizeof(fds[0])), 100);
881
882         /* timed out - periodic check for _task_should_exit */
883         if (pret == 0) {
884             continue;
885         }
886
887         /* this is undesirable but not much we can do - might want to flag unhappy status */
888         if (pret < 0) {
889             warn("mc att ctrl: poll error %d, %d", pret, errno);
890             /* sleep a bit before next try */
891             usleep(100000);
892             continue;
893         }
894
895         perf_begin(_loop_perf);
896
897         /* run controller on attitude changes */
898         if (fds[0].revents & POLLIN) {
899             static uint64_t last_run = 0;
900             float dt = (hrt_absolute_time() - last_run) / 1000000.0f;
901             last_run = hrt_absolute_time();
```

关于 poll 阻塞等待：

int poll(struct pollfd fds[], nfds_t nfds, int timeout)

功能：监控文件描述符（多个）；

说明：timemout=0,poll()函数立即返回而不阻塞；timeout=INFTIM(-1),poll()会一直阻塞下去，直到检测到 return > 0；

参数：

fds:struct pollfd 结构类型的数组；

nfds:用于标记数组 fds 中的结构体元素的总数量；

timeout:是 poll 函数调用阻塞的时间，单位：毫秒；

返回值：

>0：数组 fds 中准备好读、写或出错状态的那些 socket 描述符的总数量；

==0:poll()函数会阻塞 timeout 所指定的毫秒时间长度之后返回;

-1:poll 函数调用失败；同时会自动设置全局变量 errno；

poll( )函数用于监测多个等待事件，若事件未发生，进程睡眠，放弃 CPU 控制权。若监测的任何一个事件发生，poll 函数将唤醒睡眠的进程，并判断是什么等待事件发生，并执行相应的操作。poll( )函数退出后，struct poll dfs 变量的所有值被清零，需要重新设置。

这里只阻塞等待 control_state，且以 100ms 周期阻塞等待。但是 100ms 并不是这个这个进程执行的频率，因为在 100ms 内随时拿到新的数据，随时就可以进行下面的运算了，所以正在的执行频率决定于 control_state 主题更新的频率，决定于底层传感器数据更新的频率。

```
while (!_task_should_exit) {

    /* wait for up to 100ms for data */
    int pret = px4_poll(&fds[0], (sizeof(fds) / sizeof(fds[0])), 100);

    /* timed out - periodic check for _task_should_exit */
    if (pret == 0) {
        continue;
    }

    /* this is undesirable but not much we can do - might want to flag unhappy status */
    if (pret < 0) {
        warn("mc att ctrl: poll error %d, %d", pret, errno);
        /* sleep a bit before next try */
        usleep(100000);
        continue
    }

    perf_begin(_loop_perf);

    /* run controller on attitude changes */
    if (fds[0].revents & POLLIN) {
        static uint64_t last_run = 0;
        float dt = (hrt_absolute_time() - last_run) / 1000000.0f;
        last_run = hrt_absolute_time();

        /* guard against too small (< 2ms) and too large (> 20ms) dt's */
        if (dt < 0.002f) {
            dt = 0.002f;

        } else if (dt > 0.02f) {
            dt = 0.02f;
```

阻塞等待 control_state 姿态数据，如果有限时间内没拿到数据 continue，如果运行出错 sleep 一段时间后也 continue。由此可见 poll 的阻塞等待，势必要拿到数据。

如果拿不到数据就会陷入上面的 continue 中，当拿到数据时记录当前的时间，计算 dt 用作 pid 的计算。并对 dt 时间做出归一化。

当检测到数据发生改变时，copy 出数据。

```
        /* copy attitude and control state topics */
        orb_copy(ORB_ID(control_state), _ctrl_state_sub, &_ctrl_state);

        /* check for updates in other topics */
        parameter_update_poll();
        vehicle_control_mode_poll();
        arming_status_poll();
        vehicle_manual_poll();
        vehicle_status_poll();
        vehicle_motor_limits_poll();
        battery_status_poll();
```

关键数据 poll 的方式，等于其他的数据用 check 进行检查更新。

如：

```
void
MulticopterAttitudeControl::parameter_update_poll()
{
    bool updated;

    /* Check if parameters have changed */
    orb_check(_params_sub, &updated);

    if (updated) {
        struct parameter_update_s param_update;
        orb_copy(ORB_ID(parameter_update), _params_sub, &param_update);
        parameters_update();
    }
}
```

int orb_check(int handle, bool *updated)

功能：订阅者可以用来检查一个主题在发布者上一次更新数据后，有没有订阅者调用过 ob_copy 来接收、处理过；

说明：如果主题在在被公告前就有人订阅，那么这个 API 将返回“not-updated”直到主题被公告。可以不用 poll，只用这个函数实现数据的获取。

参数：

handle:主题句柄；

updated:如果当最后一次更新的数据被获取了，检测到并设置 updated 为 ture;

返回值：

OK 表示检测成功；错误返回 ERROR;否则则有根据的去设置 errno;

上面这些都是第一阶段，初始化阶段：获取数据，为后面的控制做数据准备。

```
        battery_status_poll();

        /* Check if we are in rattitude mode and the pilot is above the threshold on pitch
         * or roll (yaw can rotate 360 in normal att control).  If both are true don't
         * even bother running the attitude controllers */
        if (_v_control_mode.flag_control_rattitude_enabled) {
            if (fabsf(_manual_control_sp.y) > _params.rattitude_thres ||
                fabsf(_manual_control_sp.x) > _params.rattitude_thres) {
                _v_control_mode.flag_control_attitude_enabled = false;
            }
        }
```

这是 ratttitude 半自稳模式，该模式是一种新的飞行模式，只控制角速度，不控制角度， 俗称半自稳模式（小舵量自稳 大舵量手动）。 根据介绍，这个模式只有在 pitch 和 roll 都设置为 Rattitude 模式时才有意义，如果 yaw 也设置了该模式，那么就会自动被手动模式替代了。 所以代码中只做了 x、 y 阈值的检测。

```
3557
3558    case vehicle_status_s::NAVIGATION_STATE_STAB:
3559        control_mode.flag_control_manual_enabled = true;
3560        control_mode.flag_control_auto_enabled = false;
3561        control_mode.flag_control_rates_enabled = true;
3562        control_mode.flag_control_attitude_enabled = true;
3563        control_mode.flag_control_rattitude_enabled = true;
3564        control_mode.flag_control_altitude_enabled = false;
3565        control_mode.flag_control_climb_rate_enabled = false;
3566        control_mode.flag_control_position_enabled = false;
3567        control_mode.flag_control_velocity_enabled = false;
3568        control_mode.flag_control_acceleration_enabled = false;
3569        control_mode.flag_control_termination_enabled = false;
3570        /* override is not ok in stabilized mode */
3571        control_mode.flag_external_manual_override_ok = false;
3572        break;
3573
3574    case vehicle_status_s::NAVIGATION_STATE_RATTITUDE:
3575        control_mode.flag_control_manual_enabled = true;
3576        control_mode.flag_control_auto_enabled = false;
3577        control_mode.flag_control_rates_enabled = true;
3578        control_mode.flag_control_attitude_enabled = true;
3579        control_mode.flag_control_rattitude_enabled = true;
3580        control_mode.flag_control_altitude_enabled = false;
3581        control_mode.flag_control_climb_rate_enabled = false;
3582        control_mode.flag_control_position_enabled = false;
3583        control_mode.flag_control_velocity_enabled = false;
3584        control_mode.flag_control_acceleration_enabled = false;
3585        control_mode.flag_control_termination_enabled = false;
3586        break;
3587
```

从 commander 可以看出，所谓 RATTITUDE 模式其实过程和 STABILIZE 模式过程一样，

```
control_mode.flag_control_rates_enabled = true;

control_mode.flag_control_attitude_enabled = true;

control_mode.flag_control_rattitude_enabled = true;

control_mode.flag_control_manual_enabled = true;
```

简单来说就是“小舵量自稳 大舵量手动”，如果飞手的输入超过了设定的阈值，则将其转换成横滚、俯仰、偏航角速度命令传送给自驾仪；如果输入没有超过阈值，则将其转换成横滚、俯仰转角度 以及偏航角速度 命令。油门直接输出到混控器。

3、姿态控制正常外环处理

姿态控制的外环分两种三种情况：正常的外环 垂直起降的外环 以及 manual 手动下外环。

自动控制下的外环：

```
933  if (_v_control_mode.flag_control_attitude_enabled) {
934
935      if (_ts_opt_recovery == nullptr) {
936          // the  tailsitter recovery instance has not been created, thus, the vehicle
937          // is not a tailsitter, do normal attitude control
938          control_attitude(dt);
939
940      } else {
941          vehicle_attitude_setpoint_poll();
942          _thrust_sp = _v_att_sp.thrust;
943          math::Quaternion q(_ctrl_state.q[0], _ctrl_state.q[1], _ctrl_state.q[2], _ctrl_state.q[3]);
944          math::Quaternion q_sp(&_v_att_sp.q_d[0]);
945          _ts_opt_recovery->setAttGains(_params.att_p, _params.yaw_ff);
946          _ts_opt_recovery->calcOptimalRates(q, q_sp, _v_att_sp.yaw_sp_move_rate, _rates_sp);
947
948          /* limit rates */
949          for (int i = 0; i < 3; i++) {
950              _rates_sp(i) = math::constrain(_rates_sp(i), -_params.mc_rate_max(i), _params.mc_rate_max(i));
951          }
952      }
953
954      /* publish attitude rates setpoint */
955      _v_rates_sp.roll = _rates_sp(0);
956      _v_rates_sp.pitch = _rates_sp(1);
957      _v_rates_sp.yaw = _rates_sp(2);
958      _v_rates_sp.thrust = _thrust_sp;
959      _v_rates_sp.timestamp = hrt_absolute_time();
960
961      if (_v_rates_sp_pub != nullptr) {
962          orb_publish(_rates_sp_id, _v_rates_sp_pub, &_v_rates_sp);
963
```

手动控制下的外环：

```
971      /* attitude controller disabled, poll rates setpoint topic */
972      if (_v_control_mode.flag_control_manual_enabled) {
973          /* manual rates control - ACRO mode */
974          _rates_sp = math::Vector<3>(_manual_control_sp.y, -_manual_control_sp.x,
975                                      _manual_control_sp.r).emult(_params.acro_rate_max);
976          _thrust_sp = math::min(_manual_control_sp.z, MANUAL_THROTTLE_MAX_MULTICOPTER);
977
978          /* publish attitude rates setpoint */
979          _v_rates_sp.roll = _rates_sp(0);
980          _v_rates_sp.pitch = _rates_sp(1);
981          _v_rates_sp.yaw = _rates_sp(2);
982          _v_rates_sp.thrust = _thrust_sp;
983          _v_rates_sp.timestamp = hrt_absolute_time();
984
985          if (_v_rates_sp_pub != nullptr) {
986              orb_publish(_rates_sp_id, _v_rates_sp_pub, &_v_rates_sp);
987
988          } else if (_rates_sp_id) {
989              _v_rates_sp_pub = orb_advertise(_rates_sp_id, &_v_rates_sp);
990          }
991
992      } else {
993          /* attitude controller disabled, poll rates setpoint topic */
994          vehicle_rates_setpoint_poll();
995          _rates_sp(0) = _v_rates_sp.roll;
996          _rates_sp(1) = _v_rates_sp.pitch;
997          _rates_sp(2) = _v_rates_sp.yaw;
998          _thrust_sp = _v_rates_sp.thrust;
999      }
1000 }
```

在正式探讨代码前，我们先来看看几个 msg 背后的数据。

```
# This is similar to the mavlink message CONTROL_SYSTEM_STATE, but for onboard use */
uint8 AIRSPD_MODE_MEAS = 0  # airspeed is measured airspeed from sensor
uint8 AIRSPD_MODE_EST = 1   # airspeed is estimated by body velocity
uint8 AIRSPD_MODE_DISABLED = 2  # airspeed is disabled

float32 x_acc           # X acceleration in body frame
float32 y_acc           # Y acceleration in body frame
float32 z_acc           # Z acceleration in body frame
float32 x_vel           # X velocity in body frame
float32 y_vel           # Y velocity in body frame
float32 z_vel           # Z velocity in body frame
float32 x_pos           # X position in local earth frame
float32 y_pos           # Y position in local earth frame
float32 z_pos           # z position in local earth frame
float32 airspeed        # Airspeed, estimated
bool airspeed_valid     # False: Non-finite values or non-updating sensor
float32[3] vel_variance # Variance in body velocity estimate
float32[3] pos_variance # Variance in local position estimate
float32[4] q            # Attitude Quaternion
float32[4] delta_q_reset    # Amount by which quaternion has changed during last reset
uint8 quat_reset_counter    # Quaternion reset counter
float32 roll_rate       # Roll body angular rate (rad/s, x forward/y right/z down)
float32 pitch_rate      # Pitch body angular rate (rad/s, x forward/y right/z down)
float32 yaw_rate        # Yaw body angular rate (rad/s, x forward/y right/z down)
float32 horz_acc_mag    # low pass filtered magnitude of the horizontal acceleration

```

这是主题 control_state 里面的包含了 飞机的姿态数据：q 和角速度数据

roll_rates\pitch_rates\yaw_rates。

```
# Please keep the following messages identical;
#   vehicle_attitude_setpoint.msg
#   mc_virtual_attitude_setpoint.msg
#   fw_virtual_attitude_setpoint.msg
#
###############################################################################################


float32 roll_body               # body angle in NED frame
float32 pitch_body              # body angle in NED frame
float32 yaw_body                # body angle in NED frame

float32 yaw_sp_move_rate        # rad/s (commanded by user)

# For quaternion-based attitude control
float32[4] q_d                  # Desired quaternion for quaternion control
bool q_d_valid                  # Set to true if quaternion vector is valid

float32 thrust                  # Thrust in Newton the power system should generate

bool roll_reset_integral            # Reset roll integral part (navigation logic change)
bool pitch_reset_integral           # Reset pitch integral part (navigation logic change)
bool yaw_reset_integral             # Reset yaw integral part (navigation logic change)

bool fw_control_yaw                 # control heading with rudder (used for auto takeoff on runway)
bool disable_mc_yaw_control         # control yaw for mc (used for vtol weather-vane mode)

bool apply_flaps

float32 landing_gear

# WAS vehicle_attitude_setpoint mc_virtual_attitude_setpoint fw_virtual_attitude_setpoint
```

主题 vehicle_attitude_setpoint 包含了期望的姿态。

下面开始讨论姿态控制的外环，外环的主要作用是 根据姿态差算出期望的角速度。

```
/* calculate angular rates setpoint */
_rates_sp = _params.att_p.emult(e_R);
```

正常的外环处理：

```
if (_v_control_mode.flag_control_attitude_enabled) {

    if (_ts_opt_recovery == nullptr) {
        // the  tailsitter recovery instance has not been created, thus, the vehicle
        // is not a tailsitter, do normal attitude control
        control_attitude(dt);

    } else {
        vehicle_attitude_setpoint_poll();
        _thrust_sp = _v_att_sp.thrust;
        math::Quaternion q(_ctrl_state.q[0], _ctrl_state.q[1], _ctrl_state.q[2], _ctrl_state.q[3]);
        math::Quaternion q_sp(&_v_att_sp.q_d[0]);
        _ts_opt_recovery->setAttGains(_params.att_p, _params.yaw_ff);
        _ts_opt_recovery->calcOptimalRates(q, q_sp, _v_att_sp.yaw_sp_move_rate, _rates_sp);

        /* limit rates */
        for (int i = 0; i < 3; i++) {
            _rates_sp(i) = math::constrain(_rates_sp(i), -_params.mc_rate_max(i), _params.mc_rate_max(i));
        }
    }

    /* publish attitude rates setpoint */
    _v_rates_sp.roll = _rates_sp(0);
    _v_rates_sp.pitch = _rates_sp(1);
    _v_rates_sp.yaw = _rates_sp(2);
    _v_rates_sp.thrust = _thrust_sp;
    _v_rates_sp.timestamp = hrt_absolute_time();

    if (_v_rates_sp_pub != nullptr) {
        orb_publish(_rates_sp_id, _v_rates_sp_pub, &_v_rates_sp);
```

正常的外环 正常的情况下都会被使能 flag_control_attitude_enabled。在 ARCO 特技模式下不走外环直接控制角速率。

```
case vehicle_status_s::NAVIGATION_STATE_ACRO:
    control_mode.flag_control_manual_enabled = true;
    control_mode.flag_control_auto_enabled = false;
    control_mode.flag_control_rates_enabled = true;
    control_mode.flag_control_attitude_enabled = false;
    control_mode.flag_control_rattitude_enabled = false;
    control_mode.flag_control_altitude_enabled = false;
    control_mode.flag_control_climb_rate_enabled = false;
    control_mode.flag_control_position_enabled = false;
    control_mode.flag_control_velocity_enabled = false;
    control_mode.flag_control_acceleration_enabled = false;
    control_mode.flag_control_termination_enabled = false;
    break;

case vehicle_status_s::NAVIGATION_STATE_TERMINATION:
    /* disable all controllers on termination */
    control_mode.flag_control_manual_enabled = false;
    control_mode.flag_control_auto_enabled = false;
    control_mode.flag_control_rates_enabled = false;
    control_mode.flag_control_attitude_enabled = false;
    control_mode.flag_control_rattitude_enabled = false;
    control_mode.flag_control_position_enabled = false;
    control_mode.flag_control_velocity_enabled = false;
    control_mode.flag_control_acceleration_enabled = false;
    control_mode.flag_control_altitude_enabled = false;
    control_mode.flag_control_climb_rate_enabled = false;
    control_mode.flag_control_termination_enabled = true;
    break;
```

跟踪 **void**

MulticopterAttitudeControl::control_attitude(float dt)

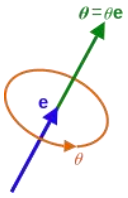
这个函数是这篇文档需要重点说明的一个问题。下面将用 1、2、3 这种符号进行区分说明。

0、在进入代码前，先铺垫一点基础。

姿态控制的外环是作用于姿态差产生期望的角速度，即姿态有误差以多大的角速度去弥补。问题就变到了，姿态偏差如何表示，这里用的是轴角法。再者姿态控制过程中是三次旋转对齐姿态，还是解耦合控制，让 roll 和 pitch 进行联动，yaw 单独控制。这里解耦合控制，因为 roll 和 pitch 响应较快，而 yaw 响应较慢，对应实际就是 roll pitch 决定飞机的“姿态”稳定要求反映一定要快，而 yaw 决定航向的我们反而不希望航向变化。首先还需要的基础的是姿态的表示方法。

1、轴角法

旋转的轴角表示用两个值参数化了旋转: 一个轴或直线，和描述绕这个轴的旋转量的一个角。它也叫做旋转的指数坐标。有时也叫做旋转向量表示，因为这两个参数(轴和角)可用在这个轴上的其模是旋转角的一个向量来表示。轴角表示在处理刚体动力学的时候是方便的。它对特征化旋转还有在刚体运动的不同表示之间的转换是有用的。



例子 [1]

假如你站在地面上，选取重力的方向为负 z 方向。如果你左转，你将绕 z 轴旋转 弧度 $\frac{\pi}{2}$ (或 90 度)。在轴角表示中，这将是

$$\langle \text{axis}, \text{angle} \rangle = \left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, \theta \right) = \left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \frac{\pi}{2} \right)$$

这可以表示为指示 z 方向的模为 $\frac{\pi}{2}$ 的旋转向量。

$$\begin{bmatrix} 0 \\ 0 \\ \frac{\pi}{2} \end{bmatrix}$$

详细可参考维基百科 https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation

2、解耦合控制

PX4 的姿态控制部分使用的是 roll-pitch 和 yaw 分开控制的（注释是为了解耦控制行为），即 tilt 和 torsion 两个环节。

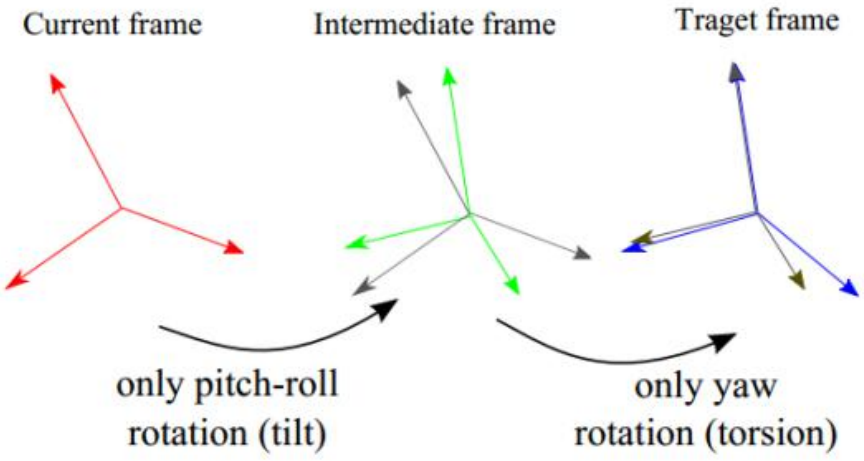


Fig. 4: Tilt-torsion decomposition. For any current (Red) and target (Blue) rotations, add an intermediate (Green) rotation. Then execute a two-stage control action.

这一部分参考论文

High Performance Full Attitude Control of a Quadrotor on SO(3)

Yun Yu, Shuo Yang, Mingxi Wang1, Cheng Li, Zexiang Li2

2015 IEEE International Conference on Robotics and Automation (ICRA) Washington State Convention Center Seattle, Washington, May 26-30, 2015

那么如何实现这种解耦合控制？

先对其姿态 z 轴，对其 z 轴的过程一定是 roll 和 pitch 进行联动，对其 z 轴后再旋转 yaw。

先实现 roll-pitch，然后 yaw；即先 tilt 后 torsion。

$$R_e = R_{torsion} R_{tilt}$$

假设目标姿态矩阵的Z轴$Z_t = [0,0,0]^T$，当前姿态矩阵的Z轴$Z_c = [x,y,z]^T$，则$Z_t = R_e Z_c$。反向推导得出$Z_c = {R_e}^T Z_t$。旋转轴为$r = Z_c \% {R_e}^T Z_t$，旋转角度为 θ，则$R_{tilt} = e^{\hat{r}\theta}$。该旋转的目的主要就是为了使当前姿态的Z轴和目标姿态的Z轴对齐。然后再进行$R_{torsion}$旋转对齐XY轴。分开控制的目的是为了解耦控制行为，即分别执行较快响应的动作和较慢效应的动作。

3、那么在对齐 z 轴的过程中 R_tilt 怎计算，罗德里格旋转公式。

罗德里格旋转公式解决的就是问题： 给定两个向量 V0 现在姿态和 V1 期望姿态，如何计算出其对应的旋转矩阵 R？即 R*V0 =V1。

$$R = e^{\hat{w}\theta} = \mathrm{I} + \sin\theta * \hat{w} + (1 - \cos\theta) * \hat{w}^2 \qquad - \text{Rodrigues' rotation formula}$$

展开后等于：

$$\begin{bmatrix} \cos\theta+\omega_x^2(1-\cos\theta) & \omega_x\omega_y(1-\cos\theta)-\omega_z\sin\theta & \omega_y\sin\theta+\omega_x\omega_z(1-\cos\theta) \\ \omega_z\sin\theta+\omega_x\omega_y(1-\cos\theta) & \cos\theta+\omega_y^2(1-\cos\theta) & -\omega_x\sin\theta+\omega_y\omega_z(1-\cos\theta) \\ -\omega_y\sin\theta+\omega_x\omega_z(1-\cos\theta) & \omega_x\sin\theta+\omega_y\omega_z(1-\cos\theta) & \cos\theta+\omega_z^2(1-\cos\theta) \end{bmatrix}$$

其中I是3x3的单位矩阵，

$\tilde{\omega}$ 是叉乘中的反对称矩阵r:

$$\tilde{\omega}=\begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}.$$

公式的证明可参考：http://www.cnblogs.com/xpvincent/archive/2013/02/15/2912836.html

其中 w 可以理解成旋转轴， θ 可以理解成旋转角。形式类似四元数。

、旋转轴

由1中可知，旋转角所在的平面为有P和Q所构成的平面，那么旋转轴必垂直该平面。

假定旋转前向量为**a**(a1, a2, a3)， 旋转后向量为**b**（b1, b2, b3)。由叉乘定义得：

$$\begin{aligned}\mathbf{a}\times\mathbf{b} &= a_1b_1\mathbf{0}+a_1b_2\mathbf{k}+a_1b_3(-\mathbf{j})+a_2b_1(-\mathbf{k})+a_2b_2\mathbf{0}+a_2b_3\mathbf{i}+a_3b_1\mathbf{j}+a_3b_2(-\mathbf{i})+a_3b_3\mathbf{0} \\ &= (a_2b_3-a_3b_2)\mathbf{i}+(a_3b_1-a_1b_3)\mathbf{j}+(a_1b_2-a_2b_1)\mathbf{k}.\end{aligned}$$

所以旋转轴**c**(c1, c2, c3)为：

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}=\begin{pmatrix} a_2b_3-a_3b_2 \\ a_3b_1-a_1b_3 \\ a_1b_2-a_2b_1 \end{pmatrix}$$

罗德里格旋转公式 wiki 链接 https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula。

4、最后补点向量的运算

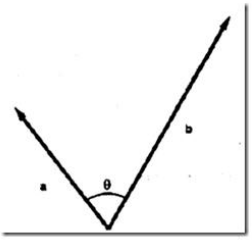
向量点乘

运算法则：

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix}\cdot\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}=\mathbf{a}_1\mathbf{b}_1+\mathbf{a}_2\mathbf{b}_2+\cdots+\mathbf{a}_{n-1}\mathbf{b}_{n-1}+\mathbf{a}_n\mathbf{b}_n$$

几何解释：

点乘结果描述了两个向量的"相似"程序，点乘结果越大，两向量越相近。



点乘等于向量大小与向量加角的cos值的积

$$\mathbf{a}\cdot\mathbf{b}=\|\mathbf{a}\|\,\|\mathbf{b}\|\cos\theta$$

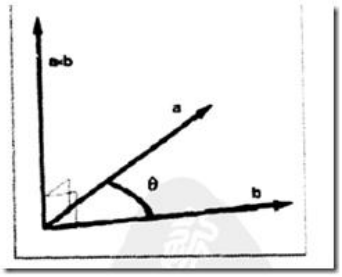
向量叉乘

运算法则：

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{bmatrix}$$

几何解释：

叉乘得到的向量垂直与原来两个向量



图中，向量a和b在一个平面中。向量a * b 指向该平面的正上方，垂直于a和b

a×b 的长度等于向量的大小与向量夹角 sin 值的积，如下：

$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \ \|\mathbf{b}\| \sin\theta$$

源码中我们用向量的点乘判断两个向量是否垂直，用向量的叉乘判断两个向量是否平行，以及可以由叉乘来就算旋转轴和向量间偏离的误差大小。

开始结合源码进行讨论：所有源码都会标注颜色，以示区分！

```
void
MulticopterAttitudeControl::control_attitude(float dt)
{
vehicle_attitude_setpoint_poll();
_thrust_sp = _v_att_sp.thrust;
```

拿期望的目标姿态 vehicle_attitude_setpoint，和期望的推力。这个期望的推力主要由两种来源，手控下直接来源于摇杆，油门摇杆的舵量对应的就是 4 个电机的转速，如何理解你就想象只打油门摇杆时 是不是转速越来越大 飞机渐减向上飞，油门摇杆映射的是四个电机的转速，这种基础转速是一样的。

```
void
MulticopterAttitudeControl::vehicle_attitude_setpoint_poll()
{
    /* check if there is a new setpoint */
    bool updated;
    orb_check(_v_att_sp_sub, &updated);

    if (updated) {
        orb_copy(ORB_ID(vehicle_attitude_setpoint), _v_att_sp_sub, &_v_att_sp);
    }
}
```

更细函数里拿出了期望的姿态，放在 v_att_sp 变量中，（vehicle attitude setpoint），下面用四元数表示的期望姿态用旋转矩阵表示，R_sp 代表期望的姿态，R 代表现在的姿态。注意这个计算过程中只涉及 roll pitch yaw 不涉及油门，其实在整个姿态控制过程中推力 thrust 是不做什么处理的，推力的处理都在位置控制里，因为在位置控制里不同模式下需要不同推力的补偿，在那里已经计算好推力了，这里姿态控制相当于对力的分解，产生 xyz 三轴的加速度进而产生速度产生位置。

```
    /* construct attitude setpoint rotation matrix */
    math::Quaternion q_sp(_v_att_sp.q_d[0], _v_att_sp.q_d[1], _v_att_sp.q_d[2],
_v_att_sp.q_d[3]);
    math::Matrix<3, 3> R_sp = q_sp.to_dcm();
    /* get current rotation matrix from control state quaternions */
    math::Quaternion q_att(_ctrl_state.q[0], _ctrl_state.q[1], _ctrl_state.q[2], _ctrl_state.q[3]);
    math::Matrix<3, 3> R = q_att.to_dcm();
```

R_sp 期望的姿态，R 现在的姿态。

```
    math::Vector<3> R_z(R(0, 2), R(1, 2), R(2, 2));
    math::Vector<3> R_sp_z(R_sp(0, 2), R_sp(1, 2), R_sp(2, 2));
    /* axis and sin(angle) of desired rotation */
    math::Vector<3> e_R = R.transposed() * (R_z % R_sp_z);
```

开始进行解耦合的处理：

比较 z 轴，这两个 z 轴还是针对地理坐标系，实际姿态控制过程是以机体坐标系来旋转的，所以将 z 轴之间的偏差转换到机体上。这里如何表示两个向量之间的误差，之前已经说过了。

```
    float e_R_z_sin = e_R.length();
    float e_R_z_cos = R_z * R_sp_z;
```

这就是简单的向量运算，为什么不直接用叉乘算 sin 呢，因为叉乘算出来是向量，.length（）

才是大小 sin。Cos 则可以直接由点乘算出来。

```
/* calculate weight for yaw control */
float yaw_w = R_sp(2, 2) * R_sp(2, 2);
```

转动是有顺序的，就是如果严格按先转 z 轴重合，再转其他轴重合是没有误差的。但是这里先转动对齐 z 轴的过程中也会影响到偏航，为什么是 R_sp(2,2)呢，这个值代表呢两个 z 轴之间夹角的余弦，当两个 z 轴的夹角越大时对偏航角度的影响也越大。

e_R(2) = atan2f((R_rp_x % R_sp_x) * R_sp_z, R_rp_x * R_sp_x) * yaw_w;

可以理解成，如两个 z 轴重合 yaw_w=1，e_R(2) 就是算出来的，相当于转动过程对偏航没有影响，这种情况确实没有影响。如果如两个 z 轴 90 度 yaw_w=0 其实 e_R(2) 偏航=0，就是这种情况下根本不用偏航了。

yaw_w 就是对偏航转动的权重分量。如果 z 轴重合，yaw_w 就是 1，权重最大，也就是以只有偏航转动，或者以其为主。随着 z 轴夹角增大，yaw 会两次方减小，降低偏航权重，使得转动以俯仰滚转为主。这或许就是解耦的思想。

```
math::Matrix<3, 3> R_rp;
```

对齐 z 轴需要旋转的量 R_tilt 就是 R_rp，这里 rp 代表 roll pitch 联动。

$$R_e = R_{torsion} R_{tilt}$$

这个旋转矩阵怎么求，用的罗德里格旋转公式，没有用欧拉角进行旋转的组合。

```
if (e_R_z_sin > 0.0f) {//0-180 度姿态控制，实际上 90 度内用它
        /* get axis-angle representation */
        float e_R_z_angle = atan2f(e_R_z_sin, e_R_z_cos);
        math::Vector<3> e_R_z_axis = e_R / e_R_z_sin;

        e_R = e_R_z_axis * e_R_z_angle;//这里误差用轴角法进行标示

        /* cross product matrix for e_R_axis */
        math::Matrix<3, 3> e_R_cp;
        e_R_cp.zero();
        e_R_cp(0, 1) = -e_R_z_axis(2);
        e_R_cp(0, 2) = e_R_z_axis(1);
```

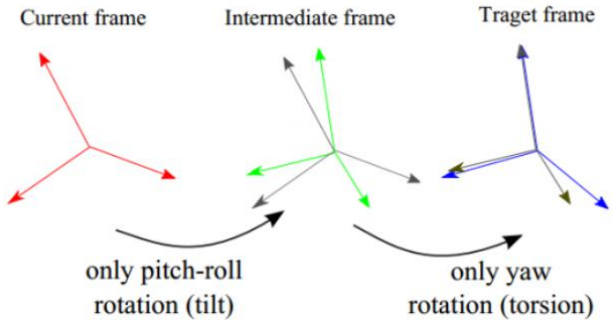
```
            e_R_cp(1, 0) = e_R_z_axis(2);
            e_R_cp(1, 2) = -e_R_z_axis(0);
            e_R_cp(2, 0) = -e_R_z_axis(1);
            e_R_cp(2, 1) = e_R_z_axis(0);

            /* rotation matrix for roll/pitch only rotation */
            R_rp = R * (_I + e_R_cp * e_R_z_sin + e_R_cp * e_R_cp * (1.0f - e_R_z_cos));

        } else {
            /* zero roll/pitch rotation */
            R_rp = R;
        }
```

上面代码分为两部分，其一对齐 z 轴的姿态误差用轴角法进行标示。e_R = e_R_z_axis * e_R_z_angle;，这里的轴角的计算可以考虑向量运算的结果。

其二对齐 z 轴产生的旋转矩阵怎么算 R_rp，为什么要算他，下面就有他表示姿态 z 轴已经对齐，再比较他的 x 轴和期望姿态的 x 轴得到就是相差的偏航了。



R_rp 怎么求，用的罗德里格旋转，w 代表旋转轴 θ代表旋转角。这种形式可以联想一下四元数的指数形式，是不是很像。

$$R = e^{\hat{w}\theta} = \mathrm{I} + \sin\theta * \hat{w} + (1 - \cos\theta) * \hat{w}^2 \qquad -\ \text{Rodrigues' rotation formula}$$

展开后等于：

$$\begin{bmatrix} \cos\theta+\omega_x^2(1-\cos\theta) & \omega_x\omega_y(1-\cos\theta)-\omega_z\sin\theta & \omega_y\sin\theta+\omega_x\omega_z(1-\cos\theta) \\ \omega_z\sin\theta+\omega_x\omega_y(1-\cos\theta) & \cos\theta+\omega_y^2(1-\cos\theta) & -\omega_x\sin\theta+\omega_y\omega_z(1-\cos\theta) \\ -\omega_y\sin\theta+\omega_x\omega_z(1-\cos\theta) & \omega_x\sin\theta+\omega_y\omega_z(1-\cos\theta) & \cos\theta+\omega_z^2(1-\cos\theta) \end{bmatrix}$$

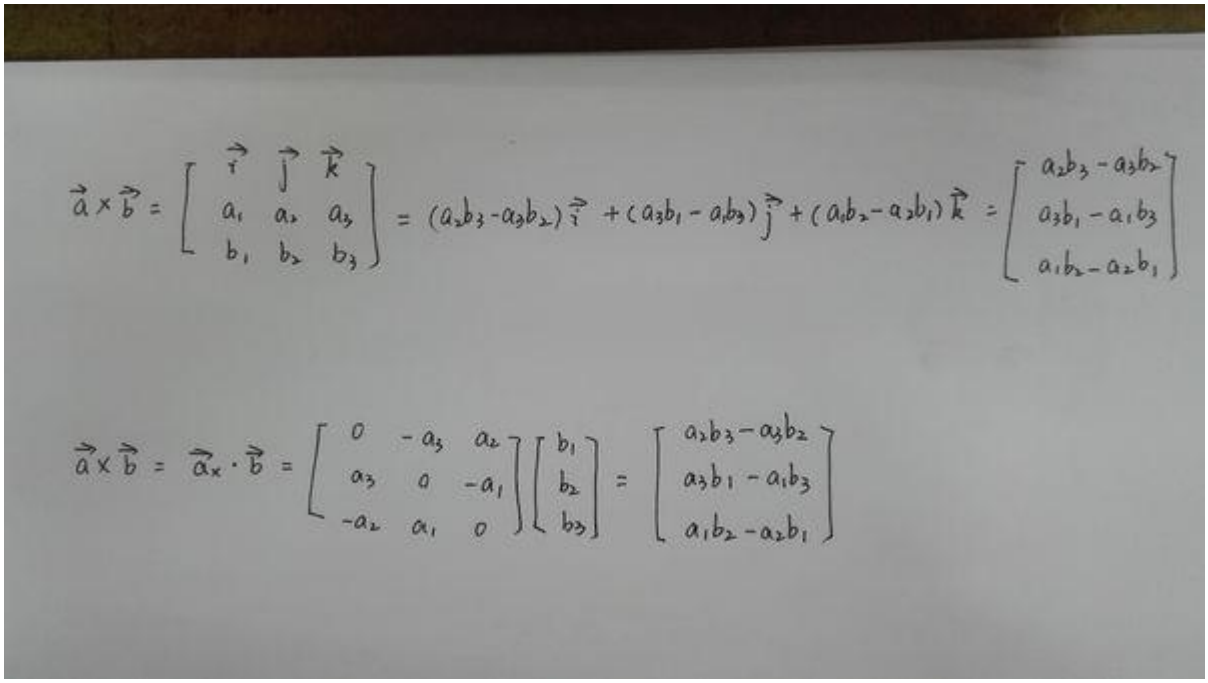
其中I是3x3的单位矩阵，

$\tilde{\omega}$ 是叉乘中的反对称矩阵r:

$$\tilde{\omega}=\begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}.$$

这里 w 写成了反对称矩阵的形式，反对称矩阵可以把两个向量的叉乘运算表示成点乘运算，从而可以简化运算。已知有两个向量 x、 y， 其叉乘公式为：

$x \times y = \hat{x} * y$。




```
math::Matrix<3, 3> e_R_cp;
e_R_cp.zero();
e_R_cp(0, 1) = -e_R_z_axis(2);
e_R_cp(0, 2) = e_R_z_axis(1);
e_R_cp(1, 0) = e_R_z_axis(2);
e_R_cp(1, 2) = -e_R_z_axis(0);
e_R_cp(2, 0) = -e_R_z_axis(1);
```

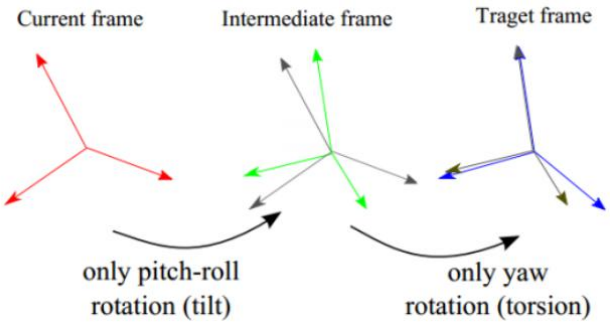
```
e_R_cp(2, 1) = e_R_z_axis(0);
```

利用旋转轴构建反对称矩阵。

```
R_rp = R * (_I + e_R_cp * e_R_z_sin + e_R_cp * e_R_cp * (1.0f - e_R_z_cos));
```

利用罗德里格旋转求得对齐 z 轴后的旋转矩阵（姿态）R_rp 。

现在已经对齐 z 轴了，现在的姿态和期望姿态之间还差一个偏航。

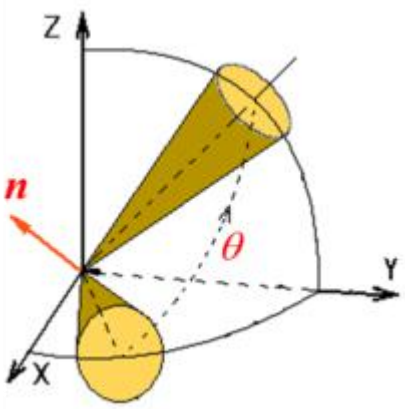



```
/* R_rp and R_sp has the same Z axis, calculate yaw error */
math::Vector<3> R_sp_x(R_sp(0, 0), R_sp(1, 0), R_sp(2, 0));
math::Vector<3> R_rp_x(R_rp(0, 0), R_rp(1, 0), R_rp(2, 0));
e_R(2) = atan2f((R_rp_x % R_sp_x) * R_sp_z, R_rp_x * R_sp_x) * yaw_w;
```

相差的偏航怎么求？比较对齐 z 轴后的姿态 和 期望姿态的 x 轴，两个之间相差就是一个偏航。

求出偏航角。

但这里我有个问题轴角法其实是绕着一个旋转轴旋转一个角，类似四元数的旋转过程。



这里直接求出偏航角赋值 e_R(2)？但是偏航旋转的旋转轴并不是之前的旋转轴了？难道做了近似处理，还是我没理解到。

```
728
729         e_R = e_R_z_axis * e_R_z_angle;
730
```

我们来看看 e_R(2)偏航角的求取，直接 sin cos arctan 求取。

e_R(2) = atan2f((R_rp_x % R_sp_x) * R_sp_z, R_rp_x * R_sp_x) * yaw_w;

(R_rp_x % R_sp_x) * R_sp_z 这是求 sin。

R_rp_x % R_sp_x 求出来是个向量，大小是 sin 但是还有方向呢，方向就是 R_sp_z。

* R_sp_z 是把前面的向量转换为数值。可以算算这个点乘，单位向量，同方向 cos=1，最后剩下的就是仅仅是我们想要的 sin 数值了。

R_rp_x * R_sp_x 求 cos。

atan2f （sin，cos）求偏航角。

yaw_w 在先旋转 roll pitch 对齐 z 轴的过程中，对偏航的影响。float yaw_w = R_sp(2, 2) * R_sp(2, 2)。yaw 权重随着 z 轴夹角增大而二次方减小，所以如果 z 轴夹角越大，就会更加偏向于先通过转动将 z 轴夹角减小。在 z 轴夹角减小后，倾向于偏航转动使 x 轴重合。

```
/* calculate angular rates setpoint */
      _rates_sp = _params.att_p.emult(e_R);
```

最终作用外环计算产生期望的角速度，即角度有速度肯定要有角速度去弥补。而且误差越大期望弥补的角速度越大。

这是小角度的姿态控制过程，小角度指在 90 度以内。如果大角度 90 度以外呢？

```
    if (e_R_z_cos < 0.0f) {//大于 90 度，飞机立起来了，大角度姿态控制
            /* for large thrust vector rotations use another rotation method:
             * calculate angle and axis for R -> R_sp rotation directly */
            math::Quaternion q_error;
            q_error.from_dcm(R.transposed() * R_sp);
            math::Vector<3> e_R_d = q_error(0) >= 0.0f ? q_error.imag()  * 2.0f : -q_error.imag()
* 2.0f;
```

```
            /* use fusion of Z axis based rotation and direct rotation */
            float direct_w = e_R_z_cos * e_R_z_cos * yaw_w;//
            e_R = e_R * (1.0f - direct_w) + e_R_d * direct_w;
        }
```

刚刚姿态偏差的计算分为两步，用轴角法表示，先算 roll 和 pitch 的偏差，再用罗德里格旋转算偏航。

现在大角度下直接求现在姿态和期望姿态之间的差值，并用四元数进行表示。

q_error.from_dcm(R.transposed() * R_sp);

math::Vector<3> e_R_d = q_error(0) >= 0.0f ? q_error.imag() * 2.0f : -q_error.imag() * 2.0f;

四元数可以表示成三角函数形式

(1) 矢量式

$$Q = q_0 + q$$

其中，q_0 称四元数 Q 的标量部分，q 称四元数 Q 的矢量部分，出 q 是三维空间中的一个向量。

(2) 复数式

$$Q = q_0 + q_1 i + q_2 j + q_3 k$$

可视为一个超复数，Q 的共轭复数记为

$$Q^* = q_0 - q_1 i - q_2 j - q_3 k$$

Q^* 称为 Q 的共轭四元数。

(3) 三角式

$$Q = \cos\frac{\theta}{2} + u\sin\frac{\theta}{2}$$

式中，θ 为实数，u 为单位向量。

(4) 指数式

$$Q = e^{u\frac{\theta}{2}}$$

θ 和 u 同上。

q_error(0)就是 cos θ /2，

q_error(0)姿态偏差在 90-180 度

q_error.imag() * 2.0f 求姿态角度差

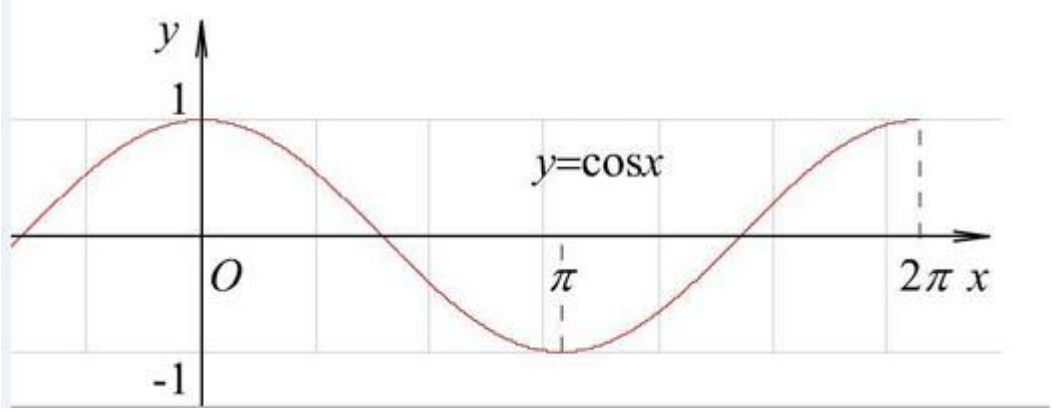
e_R_d 最终算出来就是现在姿态和期望姿态间的 直接差值，不加修饰。

float direct_w = e_R_z_cos * e_R_z_cos * yaw_w;//

偏航权重这个比较经验一点，你看 R_sp(2,2)其实就是就是 XY 两个轴需要旋转

的角度的 cos，XY 需要转的越多，那么这个权重就越小，意思就是当 XY 转角较大的时候，yaw 的控制就适当减弱一下。

这种平方的权重，当误差很大的时候，如果>1 平方就会更大，补偿更快。比如飞机遇到风的时候要求快速的补偿误差。当误差很小的手，如<1，平方会衰减的更快，就是当误差越小的时候我们需要弥补的越少，甚至可以不用弥补了免得浪费资源。就如同偏航一样，我们不希望飞机机头转的太快。



而在这里只是单纯的放小,0>e_R_z_cos>-1,代表两个 z 轴之间的偏差,在 90-180 度之间。随着 z 轴之间的角度变大，e_R_z_cos * e_R_z_cos 为正也变大，e_R = e_R * (1.0f - direct_w) + e_R_d * direct_w;最终计算的姿态误差中 e_R_d 大角度计算的这个偏差所占比重 direct_w = e_R_z_cos * e_R_z_cos * yaw_w 也变大。

就是两个 z 轴之间角度越大，大角度姿态控制下，最终姿态偏差越相信 e_R_d（）math::Vector<3> e_R_d = q_error(0) >= 0.0f ? q_error.imag() * 2.0f : -q_error.imag() * 2.0f;

最后为什么是这样的？

e_R = e_R * (1.0f - direct_w) + e_R_d * direct_w;

这是一个一节低通滤波的过程，希望姿态控制的平滑一些。

上面比重人为放大，下面进行滤波希望数据平滑一些。

大角度的情况下，直接就算角度，直接控制迅速点，不讲究稳定先修正把角度拉过去，不需要体轴那么算了，所以求的直接是角度差，先求 q_error，这个就是

两个旋转矩阵之间差的四元数，这个四元数的意义本身就是旋转，然后四元数的虚部，imag 部分，代表的 0.5*旋转角度，这个你可以看看四元数定义，然后就不管什么体轴地轴，先往 R_sp 的方向转那么多角度再说。最后算出大角度下的姿态误差，产生期望角速度。

```
_rates_sp = _params.att_p.emult(e_R);

	/* limit rates */
	for (int i = 0; i < 3; i++) {
		if ((_v_control_mode.flag_control_velocity_enabled ||
_v_control_mode.flag_control_auto_enabled) &&
			!_v_control_mode.flag_control_manual_enabled) {
			_rates_sp(i) = math::constrain(_rates_sp(i), -_params.auto_rate_max(i),
_params.auto_rate_max(i));

		} else {
			_rates_sp(i) = math::constrain(_rates_sp(i), -_params.mc_rate_max(i),
_params.mc_rate_max(i));
		}
	}
```

这种姿态控制外环的计算更多用于自动模式，手动下的外环是不需要这样计算的。对计算出来的期望角速度进行限幅。

```
	/* feed forward yaw setpoint rate */
	_rates_sp(2) += _v_att_sp.yaw_sp_move_rate * yaw_w * _params.yaw_ff;
```

前馈的作用有两个，一，让控制更加跟手，在你打航向的时候，输入误差会增大很多，这样输出到下一环的控制量是变大了，所以控制会更快 二，增加抗风性，风吹飞机一般都是小角度，飞机自稳的时候一般杆量也是 0，这些情况下，都能增加下一环的控制输入，让控制更快。四旋翼航向控制优先级是最弱的，所以增加一些前馈，对航向控制有很大帮助，而 roll/pitch 不加是因为基本上串级 PID 都够了。

4、姿态控制手动外环处理

```
970   } else {
971       /* attitude controller disabled, poll rates setpoint topic */
972       if (_v_control_mode.flag_control_manual_enabled) {
973           /* manual rates control - ACRO mode */
974           _rates_sp = math::Vector<3>(_manual_control_sp.y, -_manual_control_sp.x,
975                       _manual_control_sp.r).emult(_params.acro_rate_max);
976           _thrust_sp = math::min(_manual_control_sp.z, MANUAL_THROTTLE_MAX_MULTICOPTER);
977
978           /* publish attitude rates setpoint */
979           _v_rates_sp.roll = _rates_sp(0);
980           _v_rates_sp.pitch = _rates_sp(1);
981           _v_rates_sp.yaw = _rates_sp(2);
982           _v_rates_sp.thrust = _thrust_sp;
983           _v_rates_sp.timestamp = hrt_absolute_time();
984
985           if (_v_rates_sp_pub != nullptr) {
986               orb_publish(_rates_sp_id, _v_rates_sp_pub, &_v_rates_sp);
987
988           } else if (_rates_sp_id) {
989               _v_rates_sp_pub = orb_advertise(_rates_sp_id, &_v_rates_sp);
990           }
991
992       } else {
993           /* attitude controller disabled, poll rates setpoint topic */
994           vehicle_rates_setpoint_poll();
995           _rates_sp(0) = _v_rates_sp.roll;
996           _rates_sp(1) = _v_rates_sp.pitch;
997           _rates_sp(2) = _v_rates_sp.yaw;
998           _thrust_sp = _v_rates_sp.thrust;
999       }
1000  }
1001
```

只有在一些自动模式下，才会禁用 flag_control_manual_enabled

```
    /* fallthrough */
case vehicle_status_s::NAVIGATION_STATE_AUTO_FOLLOW_TARGET:
case vehicle_status_s::NAVIGATION_STATE_AUTO_RTGS:
case vehicle_status_s::NAVIGATION_STATE_AUTO_LAND:
case vehicle_status_s::NAVIGATION_STATE_AUTO_LANDENGFAIL:
case vehicle_status_s::NAVIGATION_STATE_AUTO_MISSION:
case vehicle_status_s::NAVIGATION_STATE_AUTO_LOITER:
case vehicle_status_s::NAVIGATION_STATE_AUTO_TAKEOFF:
    control_mode.flag_control_manual_enabled = false;
    control_mode.flag_control_auto_enabled = true;
    control_mode.flag_control_rates_enabled = true;
    control_mode.flag_control_attitude_enabled = true;
    control_mode.flag_control_rattitude_enabled = false;
    control_mode.flag_control_altitude_enabled = true;
    control_mode.flag_control_climb_rate_enabled = true;
    control_mode.flag_control_position_enabled = !status.in_transition_mode;
    control_mode.flag_control_velocity_enabled = !status.in_transition_mode;
    control_mode.flag_control_acceleration_enabled = false;
    control_mode.flag_control_termination_enabled = false;
    break;
```

```
case vehicle_status_s::NAVIGATION_STATE_AUTO_LANDGPSFAIL:
        control_mode.flag_control_manual_enabled = false;
        control_mode.flag_control_auto_enabled = false;
        control_mode.flag_control_rates_enabled = true;
        control_mode.flag_control_attitude_enabled = true;
        control_mode.flag_control_rattitude_enabled = false;
        control_mode.flag_control_altitude_enabled = false;
        control_mode.flag_control_climb_rate_enabled = true;
        control_mode.flag_control_position_enabled = false;
        control_mode.flag_control_velocity_enabled = false;
        control_mode.flag_control_acceleration_enabled = false;
        control_mode.flag_control_termination_enabled = false;
        break;

case vehicle_status_s::NAVIGATION_STATE_DESCEND:
        /* TODO: check if this makes sense */
        control_mode.flag_control_manual_enabled = false;
        control_mode.flag_control_auto_enabled = true;
        control_mode.flag_control_rates_enabled = true;
        control_mode.flag_control_attitude_enabled = true;
        control_mode.flag_control_rattitude_enabled = false;
        control_mode.flag_control_position_enabled = false;
        control_mode.flag_control_velocity_enabled = false;
        control_mode.flag_control_acceleration_enabled = false;
        control_mode.flag_control_altitude_enabled = false;
        control_mode.flag_control_climb_rate_enabled = true;
        control_mode.flag_control_termination_enabled = false;
        break;
```

回归源码中：

Manual 代表遥控的舵量，数据来源于 px4io.cpp 里取原始的摇杆值【1000,2000】，在 sensor.cpp 里面进行了归一化【-1，+1】

```
_rates_sp = math::Vector<3>(_manual_control_sp.y, -_manual_control_sp.x,
                                        _manual_control_sp.r).emult(_params.acro_rate_max);
```

摇杆直接控制控制内环的角速度，把杆量转换到实际最大的角速度.emult(_params.acro_rate_max)

```
_thrust_sp=math::min(_manual_control_sp.z, MANUAL_THROTTLE_MAX_MULTICOPTER);
```

拿油门量，并对他进行限幅 MANUAL_THROTTLE_MAX_MULTICOPTER 0.9。

即正常的外环是由姿态差计算出来的期望角速度

```
766     /* calculate angular rates setpoint */
767     _rates_sp = _params.att_p.emult(e_R);
```

手动模式下的外环是由摇杆量直接转换过来的

```
973     /* manual rates control - ACRO mode */
974     _rates_sp = math::Vector<3>(_manual_control_sp.y, -_manual_control_sp.x,
975                     _manual_control_sp.r).emult(_params.acro_rate_max);
976     _thrust_sp = math::min(_manual_control_sp.z, MANUAL_THROTTLE_MAX_MULTICOPTER);
977
```

外环处理结束后，填充数据进行发布：

```
                    /* publish attitude rates setpoint */
                    _v_rates_sp.roll = _rates_sp(0);
                    _v_rates_sp.pitch = _rates_sp(1);
                    _v_rates_sp.yaw = _rates_sp(2);
                    _v_rates_sp.thrust = _thrust_sp;
                    _v_rates_sp.timestamp = hrt_absolute_time();

                    if (_v_rates_sp_pub != nullptr) {
                        orb_publish(_rates_sp_id, _v_rates_sp_pub, &_v_rates_sp);

                    } else if (_rates_sp_id) {
                        _v_rates_sp_pub = orb_advertise(_rates_sp_id, &_v_rates_sp);
                    }
```

5、姿态控制内环处理

```
1001
1002    if (_v_control_mode.flag_control_rates_enabled) {
1003        control_attitude_rates(dt);
1004
1005        /* publish actuator controls */
1006        _actuators.control[0] = (PX4_ISFINITE(_att_control(0))) ? _att_control(0) : 0.0f;
1007        _actuators.control[1] = (PX4_ISFINITE(_att_control(1))) ? _att_control(1) : 0.0f;
1008        _actuators.control[2] = (PX4_ISFINITE(_att_control(2))) ? _att_control(2) : 0.0f;
1009        _actuators.control[3] = (PX4_ISFINITE(_thrust_sp)) ? _thrust_sp : 0.0f;
1010        _actuators.control[7] = _v_att_sp.landing_gear;
1011        _actuators.timestamp = hrt_absolute_time();
1012        _actuators.timestamp_sample = _ctrl_state.timestamp;
1013
1014        /* scale effort by battery status */
1015        if (_params.bat_scale_en && _battery_status.scale > 0.0f) {
1016            for (int i = 0; i < 4; i++) {
1017                _actuators.control[i] *= _battery_status.scale;
1018            }
1019        }
1020
1021        _controller_status.roll_rate_integ = _rates_int(0);
1022        _controller_status.pitch_rate_integ = _rates_int(1);
1023        _controller_status.yaw_rate_integ = _rates_int(2);
1024        _controller_status.timestamp = hrt_absolute_time();
1025
1026        if (!_actuators_0_circuit_breaker_enabled) {
1027            if (_actuators_0_pub != nullptr) {
1028
1029                orb_publish(_actuators_id, _actuators_0_pub, &_actuators);
1030                perf_end(_controller_latency_perf);
```

内环作用于角速度差产生控制量，这里控制量我理解成力矩，类似于角加速度用于产生姿态角，即角速度有误差，需要靠角角速度来弥补。

```
/* angular rates error */
math::Vector<3> rates_err = _rates_sp - rates;

_att_control =  _params.rate_p.emult(rates_err * tpa) +
                _params.rate_d.emult(_rates_prev - rates) / dt +
                _rates_int +
                _params.rate_ff.emult(_rates_sp);
```

内环计算结束后最终产生控制量，填充进行发布。

```
/* publish actuator controls */
_actuators.control[0] = (PX4_ISFINITE(_att_control(0))) ? _att_control(0) : 0.0f;
_actuators.control[1] = (PX4_ISFINITE(_att_control(1))) ? _att_control(1) : 0.0f;
_actuators.control[2] = (PX4_ISFINITE(_att_control(2))) ? _att_control(2) : 0.0f;
_actuators.control[3] = (PX4_ISFINITE(_thrust_sp)) ? _thrust_sp : 0.0f;
_actuators.control[7] = _v_att_sp.landing_gear;
_actuators.timestamp = hrt_absolute_time();
_actuators.timestamp_sample = _ctrl_state.timestamp;

/* scale effort by battery status */
if (_params.bat_scale_en && _battery_status.scale > 0.0f) {
    for (int i = 0; i < 4; i++) {
        _actuators.control[i] *= _battery_status.scale;
    }
}

_controller_status.roll_rate_integ = _rates_int(0);
_controller_status.pitch_rate_integ = _rates_int(1);
_controller_status.yaw_rate_integ = _rates_int(2);
_controller_status.timestamp = hrt_absolute_time();

if (!_actuators_0_circuit_breaker_enabled) {
    if (_actuators_0_pub != nullptr) {

        orb_publish(_actuators_id, _actuators_0_pub, &_actuators);
        perf_end(_controller_latency_perf);

    } else if (_actuators_id) {
        _actuators_0_pub = orb_advertise(_actuators_id, &_actuators);
```

下面进入内环函数 `control_attitude_rates(dt);`看一下：

```
799 void
800 MulticopterAttitudeControl::control_attitude_rates(float dt)
801 {
802     /* reset integral if disarmed */
803     if (!_armed.armed || !_vehicle_status.is_rotary_wing) {
804         _rates_int.zero();//解锁时重置积分项 因为积分是累加的 既然重新开始 就要归零重来，只有内环有
805     }
806
807     /* current body angular rates */
808     math::Vector<3> rates;
809     rates(0) = _ctrl_state.roll_rate; //机体坐标系
810     rates(1) = _ctrl_state.pitch_rate;
811     rates(2) = _ctrl_state.yaw_rate;
812
813     /* throttle pid attenuation factor */
814     float tpa =  fmaxf(0.0f, fminf(1.0f, 1.0f - _params.tpa_slope * (fabsf(_v_rates_sp.thrust) - _params.tpa_bre
815
816     /* angular rates error */
817     math::Vector<3> rates_err = _rates_sp - rates;
818
819     _att_control =  _params.rate_p.emult(rates_err * tpa) +
820                     _params.rate_d.emult(_rates_prev - rates) / dt +
821                     _rates_int +
822                     _params.rate_ff.emult(_rates_sp);
823
824     _rates_sp_prev = _rates_sp;
825     _rates_prev = rates;
```

tpa 的功能类似于一个简单的经验型的非线性 PID，就是根据油门大小调节 P 项的输出，让控制更加符合心理预期，你可以自己用 matlab 或者 excel 做个图，画出来就明白了。

外环在计算过程中只用 P，其实 PI 可以用效果应该会好一点。但是有点鸡肋，其一外环的

角度本就是积分得到的，没有绝对的准确值，再者外环 I 的引入会有滞后性，飞机是一个高速的物体，单纯的定位可以引入 I 增加抗干扰性，操作时外环最好不要引入 I 保证外环计算的速度，免得飞起控制起来死气沉沉。

这里内环的计算 _att_control = _params.rate_p.emult(rates_err * tpa) +

_params.rate_d.emult(_rates_prev - rates) / dt +

_rates_int +

_params.rate_ff.emult(_rates_sp);

前馈的作用使控制平滑一些。

PID 计算过程中最麻烦的是积分项的处理，防止进入饱和区，进去饱和区越深反向退出的时候越慢，会导致飞机外在反映的迟钝。

```
826
827     /* update integral only if not saturated on low limit and if motor commands are not saturated */
828     if (_thrust_sp > MIN_TAKEOFF_THRUST && !_motor_limits.lower_limit && !_motor_limits.upper_limit) {
829         for (int i = AXIS_INDEX_ROLL; i < AXIS_COUNT; i++) {
830             if (fabsf(_att_control(i)) < _thrust_sp) {
831                 float rate_i = _rates_int(i) + _params.rate_i(i) * rates_err(i) * dt;
832
833                 if (PX4_ISFINITE(rate_i) && rate_i > -RATES_I_LIMIT && rate_i < RATES_I_LIMIT &&
834                     _att_control(i) > -RATES_I_LIMIT && _att_control(i) < RATES_I_LIMIT &&
835                     /* if the axis is the yaw axis, do not update the integral if the limit is hit */
836                     !((i == AXIS_INDEX_YAW) && _motor_limits.yaw)) {
837                     _rates_int(i) = rate_i;
838                 }
839             }
840         }
841     }
842 }
843
```

这一句：抗积分饱和，当前输出没有达到饱和时，才把本次的误差积分项累加到积分环节中。

至此内环的处理已经结束，内环作用于角速度产生最终控制量。最后填充数据进行发布：

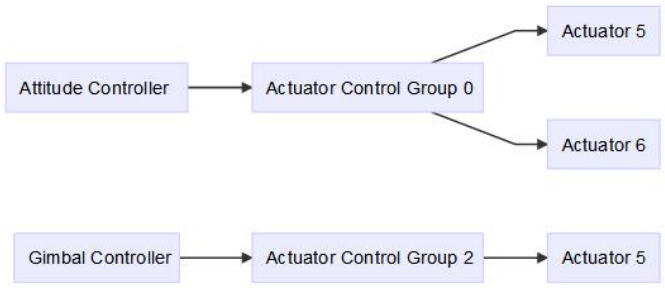
```
1002    if (_v_control_mode.flag_control_rates_enabled) {
1003        control_attitude_rates(dt);
1004
1005        /* publish actuator controls */
1006        _actuators.control[0] = (PX4_ISFINITE(_att_control(0))) ? _att_control(0) : 0.0f;
1007        _actuators.control[1] = (PX4_ISFINITE(_att_control(1))) ? _att_control(1) : 0.0f;
1008        _actuators.control[2] = (PX4_ISFINITE(_att_control(2))) ? _att_control(2) : 0.0f;
1009        _actuators.control[3] = (PX4_ISFINITE(_thrust_sp)) ? _thrust_sp : 0.0f;
1010        _actuators.control[7] = _v_att_sp.landing_gear;
1011        _actuators.timestamp = hrt_absolute_time();
1012        _actuators.timestamp_sample = _ctrl_state.timestamp;
1013
1014        /* scale effort by battery status */
1015        if (_params.bat_scale_en && _battery_status.scale > 0.0f) {
1016            for (int i = 0; i < 4; i++) {
1017                _actuators.control[i] *= _battery_status.scale;
1018            }
1019        }
1020
1021        _controller_status.roll_rate_integ = _rates_int(0);
1022        _controller_status.pitch_rate_integ = _rates_int(1);
1023        _controller_status.yaw_rate_integ = _rates_int(2);
1024        _controller_status.timestamp = hrt_absolute_time();
1025
1026        if (!_actuators_0_circuit_breaker_enabled) {
1027            if (_actuators_0_pub != nullptr) {
1028
1029                orb_publish(_actuators_id, _actuators_0_pub, &_actuators);
1030                perf_end(_controller_latency_perf);
1031
1032            } else if (_actuators_id) {
1033                _actuators_0_pub = orb_advertise(_actuators_id, &_actuators);
```

这里为什么用的 0 号控制组，这是和混控器内容相关 https://dev.px4.io/zh/concept/mixing.html

混控指的是把输入指令（例如：遥控器打右转）分配到电机以及舵机的执行器（如电调或舵机 PWM）指令。对于固定翼的副翼控制而言，每个副翼由一个舵机控制，那么混控的意义就是控制其中一个副翼抬起而另一个副翼落下。同样的，对多旋翼而言，俯仰操作需要改变所有电机的转速。

将混控逻辑从实际姿态控制器中分离出来可以大大提高复用性。

一个特定的控制器（如姿态控制器）发送特定的归一化（-1..+1）的命令到给混合（mixing），然后混合后输出独立的PWM到执行器（电调，舵机等）.在经过输出驱动如（串口，UAVCAN，PWM）等将归一化的值再转回特性的值（如输出1300的PWM等）。



控制组

PX4 有输入组和输出组的概念，顾名思义：控制输入组（如： `attitude` ），就是用于核心的飞行姿态控制，（如： `gimbal` ）就是用于挂载控制. 一个输出组就是一个物理总线，如前8个PWM组成的总线用于舵机控制，组内带8个归一化（-1..+1）值,一个混合就是用于输入和输出连接方式（如:对于四轴来说,输入组有俯仰，翻滚，偏航等，对于于向前打俯仰操作，就需要改变输出组中的4个电调的PWM输出值，前俩个降低转速，后两个增加转速，飞机就向前）。

对于简单的固定翼来说，输入0（roll），就直接连接到输出的0（副翼）。对于多旋翼来说就不同了，输入0（roll）需要连接到所有的4个电机。

Control Group #0 (Flight Control)

- 0: roll (-1..1)
- 1: pitch (-1..1)
- 2: yaw (-1..1)
- 3: throttle (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: flaps (-1..1)
- 5: spoilers (-1..1)
- 6: airbrakes (-1..1)
- 7: landing gear (-1..1)

四、小结

姿态控制源码实现可以看做 4 步骤：

1、订阅数据：int pret = px4_poll(&fds[0], (sizeof(fds) / sizeof(fds[0])), 100);

2、外环计算：

```
/* calculate angular rates setpoint */
_rates_sp = _params.att_p.emult(e_R);
```

```
/* manual rates control - ACRO mode */
_rates_sp = math::Vector<3>(_manual_control_sp.y, -_manual_control_sp.x,
                _manual_control_sp.r).emult(_params.acro_rate_max);
_thrust_sp = math::min(_manual_control_sp.z, MANUAL_THROTTLE_MAX_MULTICOPTER);
```

注意：e_R 两种不同的求法，注意外环 rates_sp 数据的来源。

3、内环计算：

```
_att_control =  _params.rate_p.emult(rates_err * tpa) +
                _params.rate_d.emult(_rates_prev - rates) / dt +
                _rates_int +
                _params.rate_ff.emult(_rates_sp);
```

注意积分环节的处理。

4、发布数据：

```
if (!_actuators_0_circuit_breaker_enabled) {
    if (_actuators_0_pub != nullptr) {

        orb_publish(_actuators_id, _actuators_0_pub, &_actuators);
        perf_end(_controller_latency_perf);

    } else if (_actuators_id) {
        _actuators_0_pub = orb_advertise(_actuators_id, &_actuators);
    }

}
```

注意控制量的理解和混控器的理解。