

mc_pos_control.cpp

1. 位置控制的大体过程是什么？

- (1) copy commander 和 navigator 产生的期望位置-----_pos_sp_triplet 结构体
- (2) 产生位置/速度设定值(期望值)-----_pos_sp<3>向量和_vel_sp<3>向量
- (3) 产生可利用的速度设定值(期望值)-----_vel_sp<3>向量
- (4) 产生可利用的推力设定值(期望值)-----thrust_sp<3>向量
- (5) 根据推力向量计算姿态设定值(期望姿态)-----q_sp 四元数矩阵和 R_sp 旋转矩阵
- (6) 将之前程序得到的各种信息填充_local_pos_sp 结构体，并发布出去-----_local_pos_sp (第 2、3 步得到的)
- (7) 根据具体应用更改之前得到的姿态设定值(期望姿态)，并发布出去-----_att_sp (第 5 步得到的)

```
extern "C" __EXPORT int mc_pos_control_main(int argc, char *argv[]);
int mc_pos_control_main(int argc, char *argv[]){
    .....
    pos_control::g_control = new MulticopterPositionControl;//构造函数
    .....
    if (OK != pos_control::g_control->start())
    {.....}
    .....
}
进入 start()
MulticopterPositionControl::start(){
    ASSERT(_control_task == -1);
    /* start the task */
    _control_task = px4_task_spawn_cmd("mc_pos_control",
                                      SCHED_DEFAULT,
                                      SCHED_PRIORITY_MAX - 5,
                                      1900,
                                      (px4_main_t)&MulticopterPositionControl::task_main_trampoline,//创建线程
                                      nullptr);
    if (_control_task < 0) {
        warn("task start failed");
        return -errno;
    }
    return OK;
}
void MulticopterPositionControl::task_main_trampoline(int argc, char *argv[]){
    pos_control::g_control->task_main();
}
```

接下来进入 task_main(), task_main()特别长/情况也特别复杂，所以需要分清楚层次/程序运行的条件

```
void MulticopterPositionControl::task_main(){
    init 部分(只运行一次)
    while (!_task_should_exit) {
        .....//获取传感器数据/_vel(i)赋值/标志位赋值等
        if (_control_mode.flag_control_altitude_enabled ||
            _control_mode.flag_control_position_enabled ||
            _control_mode.flag_control_climb_rate_enabled ||
            _control_mode.flag_control_velocity_enabled) {//基本上运行到这都会满足这个条件
            /*****高度控制、位置控制、爬升速率控制、速度控制的相关程序开始*****/
            control_manual(dt);/ control_offboard(dt);/ control_auto(dt);
```

```

if (!_control_mode.flag_control_manual_enabled && _pos_sp_triplet.current.valid
    && _pos_sp_triplet.current.type == position_setpoint_s::SETPOINT_TYPE_IDLE) {.....}
else if (_control_mode.flag_control_manual_enabled
    && _vehicle_status.condition_landed) {.....}

else {
.....//飞行器起飞/降落等情况的处理
.....//飞行器优化处理为了得到更好的_vel_sp， 比如利用限制水平方向加速度等
//这部分为了得到_vel_sp(i)
if (_control_mode.flag_control_climb_rate_enabled || _control_mode.flag_control_velocity_enabled) {
    if (_control_mode.flag_control_climb_rate_enabled) {.....}
    if (_control_mode.flag_control_velocity_enabled) {.....}
    if (!control_vel_enabled_prev && _control_mode.flag_control_velocity_enabled) {.....}
    math::Vector<3> thrust_sp = vel_err.emult(_params.vel_p) + _vel_err.d.emult(_params.vel_d) +
        thrust_int;
    //推力设定值(三维)=速度差*P+速度差的差*D+积分
    if (_pos_sp_triplet.current.type == position_setpoint_s::SETPOINT_TYPE_TAKEOFF
        && !_takeoff_jumped && !_control_mode.flag_control_manual_enabled) {.....}
    if (!_control_mode.flag_control_velocity_enabled) {.....}
    if (!_control_mode.flag_control_climb_rate_enabled) {.....}
    _vel_z_lp = _vel_z_lp * (1.0f - dt * 8.0f) + dt * 8.0f * _vel(2); //垂直速度低通滤波
    _acc_z_lp = _acc_z_lp * (1.0f - dt * 8.0f) + dt * 8.0f * vel_z_change; //垂直加速度低通滤波
    if (!_control_mode.flag_control_manual_enabled && _pos_sp_triplet.current.valid &&
        _pos_sp_triplet.current.type == position_setpoint_s::SETPOINT_TYPE_LAND) {.....}
        //着陆处理
    if (_control_mode.flag_control_velocity_enabled) {.....} //限制最大斜率(xy 方向推力限幅)
    if (_control_mode.flag_control_altitude_enabled) {.....} //推力补偿，用于高度控制
    if (thrust_abs > thr_max) {.....} //推力限幅
    //经过之前的处理，得到合适的 thrust_sp
    if (_control_mode.flag_control_velocity_enabled) {
        //body_x、body_y、body_z 应该是方向余弦矩阵的三个列向量
        body_z = -thrust_sp / thrust_abs; //body_z 矩阵是推力设定值矩阵的标准化
        y_C(-sinf(_att_sp.yaw_body), cosf(_att_sp.yaw_body), 0.0f);
        //y_C 相当于矩阵(-sin(偏航角),cos(偏航角),0)
        body_x = y_C % body_z; //是求叉积运算
        body_y = body_z % body_x;
        /*****再求出 R<3,3>矩阵*****/
        /* fill rotation matrix */
        for (int i = 0; i < 3; i++) {
            R(i, 0) = body_x(i);
            R(i, 1) = body_y(i);
            R(i, 2) = body_z(i);
        }
        /*****将 R<3,3>矩阵 copy 到_att_sp.R_body[]*****/
        memcpy(&_att_sp.R_body[0], R.data, sizeof(_att_sp.R_body));
        /****由方向余弦旋转矩阵 R 得到四元数，并 copy 到 att_sp.q_d[]*****/
        math::Quaternion q_sp;
        q_sp.from_dcm(R);
        memcpy(&_att_sp.q_d[0], &q_sp.data[0], sizeof(_att_sp.q_d));
        /*****由旋转矩阵 R 得到姿态设置欧拉角，只是 log 调试用，不是给控制用的**/

```

```

        math::Vector<3> euler = R.to_euler();
        _att_sp.roll_body = euler(0);
        _att_sp.pitch_body = euler(1);
        //yaw 已经用于构建原始矩阵
    }

    else if (!_control_mode.flag_control_manual_enabled) {.....}
    //没有位置控制的高度控制(故障安全降落), 固定水平姿态, 不改变 yaw 角
    /*****用于 log, 方便调试*****/
    _local_pos_sp.acc_x = thrust_sp(0) * ONE_G;
    _local_pos_sp.acc_y = thrust_sp(1) * ONE_G;
    _local_pos_sp.acc_z = thrust_sp(2) * ONE_G;
}

}

/*****将之前程序得到的各种信息填充_local_pos_sp 结构体, 并发布出去*****/
/* fill local position, velocity and thrust setpoint */
_local_pos_sp.timestamp = hrt_absolute_time();
_local_pos_sp.x = _pos_sp(0);
_local_pos_sp.y = _pos_sp(1);
_local_pos_sp.z = _pos_sp(2);
//第二部分第一步: 产生位置/速度设定值(期望值)
_local_pos_sp.yaw = _att_sp.yaw_body;
_local_pos_sp.vx = _vel_sp(0);
_local_pos_sp.vy = _vel_sp(1);
_local_pos_sp.vz = _vel_sp(2);
//第二部分第二步的重点(1):产生可利用的速度设定值(期望值)
/* publish local position setpoint */
if (_local_pos_sp_pub != nullptr) {
    orb_publish(ORB_ID(vehicle_local_position_setpoint), _local_pos_sp_pub, &_local_pos_sp);
} else {
    _local_pos_sp_pub = orb_advertise(ORB_ID(vehicle_local_position_setpoint), &_local_pos_sp);
}

/*****高度控制、位置控制、爬升速率控制、速度控制的相关程序结束*****/
else {.....}

if (_control_mode.flag_control_manual_enabled && _control_mode.flag_control_attitude_enabled) {.....}
/*****此判断是并列于“高度控制、位置控制、爬升速率控制、速度控制”的判断*****/
*****所以会出现混控现象, 在执行任务的时候还可以遥控控制*****
*****手动控制和姿态控制都使能, 则运行以下程序产生姿态设定值*****
*****刷新之前求得的_att_sp*****/
if (!(_control_mode.flag_control_offboard_enabled &&
    !(_control_mode.flag_control_position_enabled ||
    _control_mode.flag_control_velocity_enabled))) {
    if (_att_sp_pub != nullptr) {
        orb_publish(_attitude_setpoint_id, _att_sp_pub, &_att_sp);
    } else if (_attitude_setpoint_id) {
        _att_sp_pub = orb_advertise(_attitude_setpoint_id, &_att_sp);
    }
}

//发布姿态设定值, 如果位置/速度失能而机外(offboard)使能, 则不发布姿态设定值, 因为这种情况
//姿态设定值是通过 mavlink 应用发布的, 飞机工作于垂直起降或者做一个过渡, 也不发布, 因为此
//时由垂直起降控制部分发布

```

```

        reset_int_z_manual = _control_mode.flag_armed && _control_mode.flag_control_manual_enabled
        && !_control_mode.flag_control_climb_rate_enabled;
        //手动控制后复位高度控制的积分(悬停油门)，以便更好的转变为手动模式
    }
}

```

2. 控制环是什么样子的？

当需要位置控制时，用 **P-PID** 控制环；当不需要位置控制时，不要外环，只用速度环(内环) **PID**。

程序中需要注意 `_run_pos_control/` `_run_alt_control` 标志位，而 `_run_pos_control/` `_run_alt_control` 标志位的改变又涉及 `_pos_hold_engaged/` `_alt_hold_engaged` 标志位

//飞行器的位置或者速度(local_position_estimator 或者 position_estimator_inav 中的)

```

_pos(0) = _local_pos.x;
_pos(1) = _local_pos.y;
if (_params.alt_mode == 1 && _local_pos.dist_bottom_valid) {
    _pos(2) = -_local_pos.dist_bottom;
} else {
    _pos(2) = _local_pos.z;
}
_vel(0) = _local_pos.vx;
_vel(1) = _local_pos.vy;
if (_params.alt_mode == 1 && _local_pos.dist_bottom_valid) {
    _vel(2) = -_local_pos.dist_bottom_rate;
} else {
    _vel(2) = _local_pos.vz;
}

```

.....

```

//速度的微分
_vel_err_d(0) = _vel_x_deriv.update(-_vel(0));
_vel_err_d(1) = _vel_y_deriv.update(-_vel(1));
_vel_err_d(2) = _vel_z_deriv.update(-_vel(2));
if (_control_mode.flag_control_altitude_enabled ||
    _control_mode.flag_control_position_enabled ||
    _control_mode.flag_control_climb_rate_enabled ||
    _control_mode.flag_control_velocity_enabled ||
    _control_mode.flag_control_acceleration_enabled) {

```

.....

`_run_pos_control = true;` //用于判断是否需要外环，当为 true 则需要外环，当为 false 则不需要外环

`_run_alt_control = true;` //在后面会根据此标志位判断

```

if (_control_mode.flag_control_manual_enabled) {
    /* manual control */
    control_manual(dt);
    _mode_auto = false;
} else if (_control_mode.flag_control_offboard_enabled) {
    /* offboard control */
    control_offboard(dt);
    _mode_auto = false;
} else {
    /* AUTO */
    control_auto(dt);
}

```

```

.....
if (_run_pos_control) {
    _vel_sp(0) = (_pos_sp(0) - _pos(0)) * _params.pos_p(0);
    _vel_sp(1) = (_pos_sp(1) - _pos(1)) * _params.pos_p(1);
}
.....
if (_run_alt_control) {
    _vel_sp(2) = (_pos_sp(2) - _pos(2)) * _params.pos_p(2);
}
.....
void MulticopterPositionControl::control_manual(float dt)
{
    //req_vel_sp 来自于遥控
    math::Vector<3> req_vel_sp; // X,Y in local frame and Z in global (D), in [-1,1] normalized range
    req_vel_sp.zero();
    if (_control_mode.flag_control_altitude_enabled) {
        /* set vertical velocity setpoint with throttle stick */
        req_vel_sp(2) = -scale_control(_manual.z - 0.5f, 0.5f, _params.alt_ctl_dz, _params.alt_ctl_dy); // D
    }
    if (_control_mode.flag_control_position_enabled) {
        /* set horizontal velocity setpoint with roll/pitch stick */
        req_vel_sp(0) = _manual.x;
        req_vel_sp(1) = _manual.y;
    }
    if (_control_mode.flag_control_altitude_enabled) {
        /* reset alt setpoint to current altitude if needed */
        reset_alt_sp();
    }
    if (_control_mode.flag_control_position_enabled) {
        /* reset position setpoint to current position if needed */
        reset_pos_sp();
    }
    /* limit velocity setpoint */
    float req_vel_sp_norm = req_vel_sp.length();
    if (req_vel_sp_norm > 1.0f) {
        req_vel_sp /= req_vel_sp_norm;
    }
    /* _req_vel_sp scaled to 0..1, scale it to max speed and rotate around yaw */
    math::Matrix<3, 3> R_yaw_sp;
    R_yaw_sp.from_euler(0.0f, 0.0f, _att_sp.yaw_body);
    math::Vector<3> req_vel_sp_scaled = R_yaw_sp * req_vel_sp.emult(
        _params.vel_cruise); // in NED and scaled to actual velocity
    /*
     * assisted velocity mode: user controls velocity, but if velocity is small enough, position
     * hold is activated for the corresponding axis
     */
    /* horizontal axes */
    if (_control_mode.flag_control_position_enabled) {
        /* check for pos. hold */

```

```

if (fabsf(req_vel_sp(0)) < _params.hold_xy_dz && fabsf(req_vel_sp(1)) < _params.hold_xy_dz) {
//遥控没有拨动
    if (!_pos_hold_engaged) {
        if (_params.hold_max_xy < FLT_EPSILON || (fabsf(_vel(0)) < _params.hold_max_xy
            && fabsf(_vel(1)) < _params.hold_max_xy)) {
            //飞机速度小于一个阈值
            _pos_hold_engaged = true;//激活位置控制，需要外环

        } else {
            _pos_hold_engaged = false;//不激活位置控制，不需要外环
        }
    }
} else {
    _pos_hold_engaged = false;//不激活位置控制，不需要外环
}
/* set requested velocity setpoint */
if (!_pos_hold_engaged) { //不激活位置控制，不需要外环
    _pos_sp(0) = _pos(0); //期望位置跟随实际位置
    _pos_sp(1) = _pos(1); //期望位置跟随实际位置
    _run_pos_control = false; /* request velocity setpoint to be used, instead of position setpoint */
    //判断是否需要外环
    _vel_sp(0) = req_vel_sp_scaled(0); //更新期望速度
    _vel_sp(1) = req_vel_sp_scaled(1); //更新期望速度
}
}
/* vertical axis */
if (_control_mode.flag_control_altitude_enabled) {
    /* check for pos. hold */
    if (fabsf(req_vel_sp(2)) < FLT_EPSILON) {
        if (!_alt_hold_engaged) {
            if (_params.hold_max_z < FLT_EPSILON || fabsf(_vel(2)) < _params.hold_max_z) {
                _alt_hold_engaged = true; //激活位置控制，需要外环
            } else {
                _alt_hold_engaged = false; //不激活位置控制，不需要外环
            }
        }
    } else {
        _alt_hold_engaged = false; //不激活位置控制，不需要外环
    }
    /* set requested velocity setpoint */
    if (!_alt_hold_engaged) {
        _run_alt_control = false; /* request velocity setpoint to be used, instead of altitude setpoint */
        _vel_sp(2) = req_vel_sp_scaled(2); //更新期望速度
        _pos_sp(2) = _pos(2); //期望高度跟随实际高度
    }
}
}
}

```

由于 `_run_pos_control = true; _run_alt_control = true` 一直都在程序前面并且循环，所以默认是需要外环的，外环计算出来的期望速度会替代遥控的产生的期望速度；只有当拨动了遥控或者速度比较大时 `_run_pos_control =`

false;_run_alt_control = false 才只有速度控制环

control_offboard(dt)和 control_auto(dt)是一样的分析

control_offboard(dt);里

_run_pos_control/_run_alt_control 可为 false 或者不改

control_auto(dt);里

_run_pos_control/_run_alt_control 没有改动

同时可以解释一个现象：当飞行器是 position(定点)模式下，已经定好点了，此时拨动遥控，期望位置一直跟随实际位置，当不拨动遥控时，飞行器会重新定到新的位置而不返回之前的定点位置。

顺着程序再来介绍内环 PID

.....

比例

```
math::Vector<3> vel_err = _vel_sp - _vel;
```

```
if (!control_vel_enabled_prev && _control_mode.flag_control_velocity_enabled) {
```

```
    // choose velocity xyz setpoint such that the resulting thrust setpoint has the direction
```

```
    // given by the last attitude setpoint
```

```
    //矫正 xy 速度设定值
```

```
_vel_sp(0) = _vel(0) + (-PX4_R(_att_sp.R_body, 0, 2) * _att_sp.thrust - thrust_int(0) - _vel_err_d(0) * _params.vel_d(0)) / _params.vel_p(0);
```

```
_vel_sp(1) = _vel(1) + (-PX4_R(_att_sp.R_body, 1, 2) * _att_sp.thrust - thrust_int(1) - _vel_err_d(1) * _params.vel_d(1)) / _params.vel_p(1);
```

```
_vel_sp(2) = _vel(2) + (-PX4_R(_att_sp.R_body, 2, 2) * _att_sp.thrust - thrust_int(2) - _vel_err_d(2) * _params.vel_d(2)) / _params.vel_p(2);
```

```
_vel_sp_prev(0) = _vel_sp(0);
```

```
_vel_sp_prev(1) = _vel_sp(1);
```

```
_vel_sp_prev(2) = _vel_sp(2);
```

```
control_vel_enabled_prev = true;
```

```
// compute updated velocity error
```

```
//用矫正后的速度设定值-实际速度，跟新速度误差
```

```
vel_err = _vel_sp - _vel;
```

```
}
```

.....

PID 公式

```
math::Vector<3> thrust_sp = vel_err.emult(_params.vel_p) + _vel_err_d.emult(_params.vel_d) + thrust_int;
```

```
//推力设定值(三维)=速度差*P+速度差的差*D+积分
```

.....

积分

```
if (_control_mode.flag_control_velocity_enabled && !saturation_xy) {
```

```
    thrust_int(0) += vel_err(0) * _params.vel_i(0) * dt;
```

```
    thrust_int(1) += vel_err(1) * _params.vel_i(1) * dt;
```

```
}
```

```
if (_control_mode.flag_control_climb_rate_enabled && !saturation_z) {
```

```
    thrust_int(2) += vel_err(2) * _params.vel_i(2) * dt;
```

```
    /* protection against flipping on ground when landing */
```

```
    if (thrust_int(2) > 0.0f) {
```

```
        thrust_int(2) = 0.0f;
```

```
    }
```

```
}
```

