算法设计

模糊PID控制器设计文档

自整定

说明

定位器的自整定是给控制模块准备数据的

定位器的自整定过程是一个自动化的过程，旨在为控制模块准备和优化数据，以确保控制系统能够以最佳的性能运行。这个过程通常涉及以下几个关键步骤：

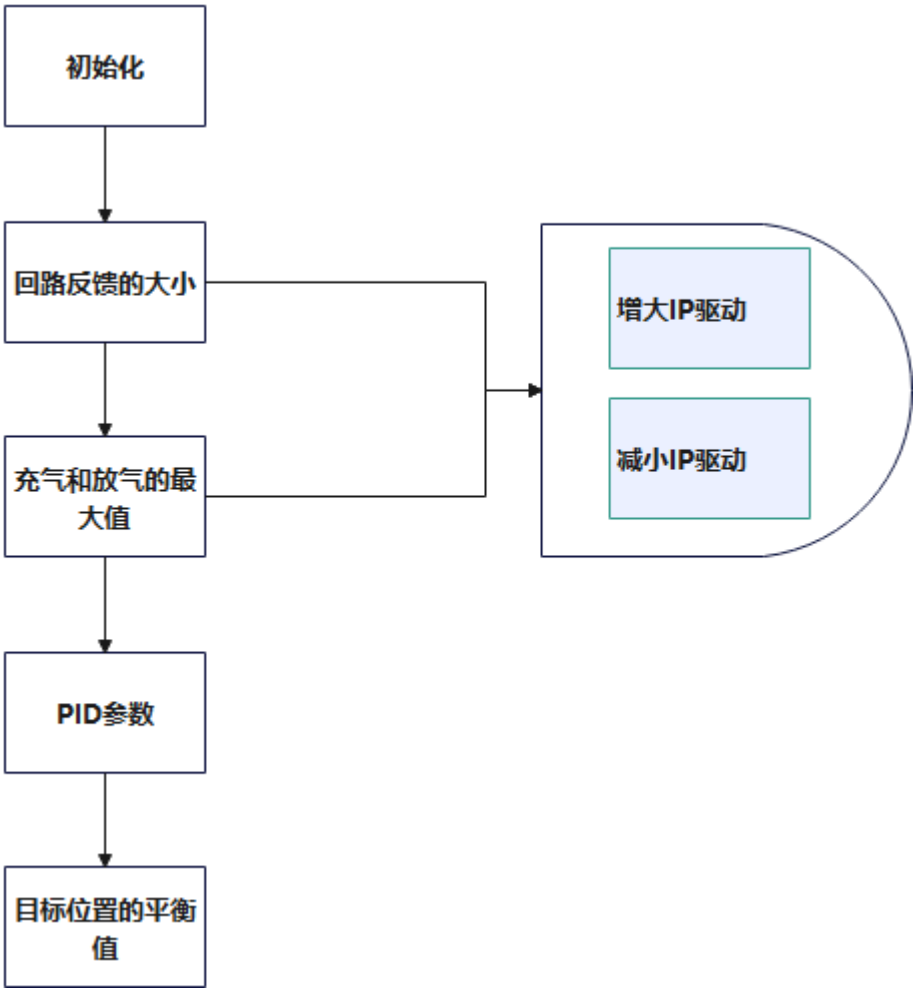
1. **数据收集**：定位器在系统的正常运行过程中或通过特定的测试信号，收集系统的响应数据。这些数据包括但不限于系统的输入、输出、响应时间、稳定性指标等。
2. **系统识别**：通过分析收集到的数据，识别系统的动态特性，如传递函数、时间常数、死区时间等。这一步是自整定过程中非常关键的一环，因为正确的系统识别是制定有效控制策略的基础。
3. **参数优化**：根据系统的动态特性和预定的性能指标（如快速响应、最小超调、稳定性等），优化控制参数。这些参数可能包括PID控制器的比例（P）、积分（I）和

微分（D）参数。

4. **验证和微调**：将优化后的参数应用于控制模块，观察系统的实际运行性能。根据实际性能与预期目标的差异，进行必要的微调。
5. **实施和监控**：一旦确认参数能够使系统达到最佳性能，就将这些参数固化为控制模块的默认设置，并持续监控系统性能，确保长期稳定运行。

针对GPS2000定位器需要准备的数据有，回路反馈的大小、充气和放气的最大值、PID参数、目标位置的平衡值。

设计



流程

```
void mode_control_adjust_process(uint8_t *state)
{
    uint8_t ts = *state;
    BOOL ble_output = FALSE;
    BIT_CLR(ts, BIT7); // 高位清零
```

```
{
    if (mode_control_adjust == NULL)
    {
        mode_control_adjust = osel_mem_alloc(sizeof(mode_control_adjust_t));
        DBG_ASSERT(mode_control_adjust != NULL __DBG_LINE);
    }
}

mode_control_adjust->adjust_state2 = (mode_control_adjust_state_e)ts;

switch (ts)
{
case CONTROL_ADJUST_IDEL: // 空闲
    mode_control_adjust_idle(state, CONTROL_ADJUST_UPWARD_SLOPE);
    mode_control_adjust->test_index = 1;
    break;
case CONTROL_ADJUST_UPWARD_SLOPE: // 上坡
    mode_control_adjust_upward_slope(state, CONTROL_ADJUST_DOWNWARD_SLOPE);
    ble_output = TRUE;
    break;
case CONTROL_ADJUST_DOWNWARD_SLOPE: // 下坡
    mode_control_adjust_downward_slope(state, CONTROL_ADJUST_SAVE);
    ble_output = TRUE;
    break;
case CONTROL_ADJUST_SAVE: // 存储变量
    mode_control_adjust_save(state, CONTROL_ADJUST_TUNING);
    break;
case CONTROL_ADJUST_TUNING: // 自整定
    mode_control_adjust_tuning_pi(state, CONTROL_ADJUST_STOP);
    ble_output = TRUE;
    break;
case CONTROL_ADJUST_STOP: // 停止
    mode_control_adjust_stop(state);
    mode_control_pid_load();
    mode_control_start();
    break;
case CONTROL_ADJUST_FAIL: // 整定失败
    mode_control_adjust_fail(state);
    break;
case CONTROL_ADJUST_BLEEDING: // 在整定状态中放气
    mode_control_adjust_bleeding(state);
    break;
case CONTROL_ADJUST_AERATE: // 在整定状态中充气
    mode_control_adjust_aerate(state);
    break;
case CONTROL_ADJUST_TEST:
    mode_control_adjust_test(state, mode_control_adjust->test_index);
    break;
case CONTROL_ADJUST_PAUSE:
    mode_control_adjust_pause(state);
    break;
default:
    break;
}

mode_control_adjust->enter_count1++;
if (ble_output == TRUE && FSM_IS_WAIT(*state) && mode_control_adjust-
```

```
>enter_count1 % (MODE_CONTROL_CYCLE_BASE * 2 / mode_control->wait_count_max) == 0)
    {
        float current_trip;
        if (ts == CONTROL_ADJUST_TUNING)
        {
            mode_control_bluetooth_output_control();
        }
        else
        {
            current_trip =
get_pid_travel(actual_adc_convert_percent(mode_control_adjust->current_psb));
            mode_control_bluetooth_output_trip(current_trip);
        }
    }
}
```

控制模块

说明

算法控制模块是一个控制系统的实现，主要用于管理和调整一个或多个过程的状态。它包括一系列的函数，用于启动和停止控制过程、调整控制参数、获取调整结果、以及管理控制算法的运行状态。以下是模块中各个部分的简要说明：

控制过程函数：

- `_process_start`：启动控制过程。
- `_process_stop`：停止控制过程，并将控制状态设置为停止。

调整控制函数：

- `_adjust_start`和 `_adjust_stop`：分别用于开始和停止调整过程。
- `_adjust_result`：返回调整过程的结果，判断是否正在调整或已完成调整。
- `_adjust_step_count`和 `_adjust_step_current`：用于管理调整步骤的计数和当前步骤。
- `_adjust_data`：填充调整数据结构，包括位置、输出、电流、时间和PID系数等信息。

状态检查函数：

- `_control_idle`：检查控制过程是否处于空闲状态。
- `_adjust_isrun`：检查是否正在进行调整过程。

- `_algorithm_calibrated_status`：检查算法是否已校准。

辅助函数：

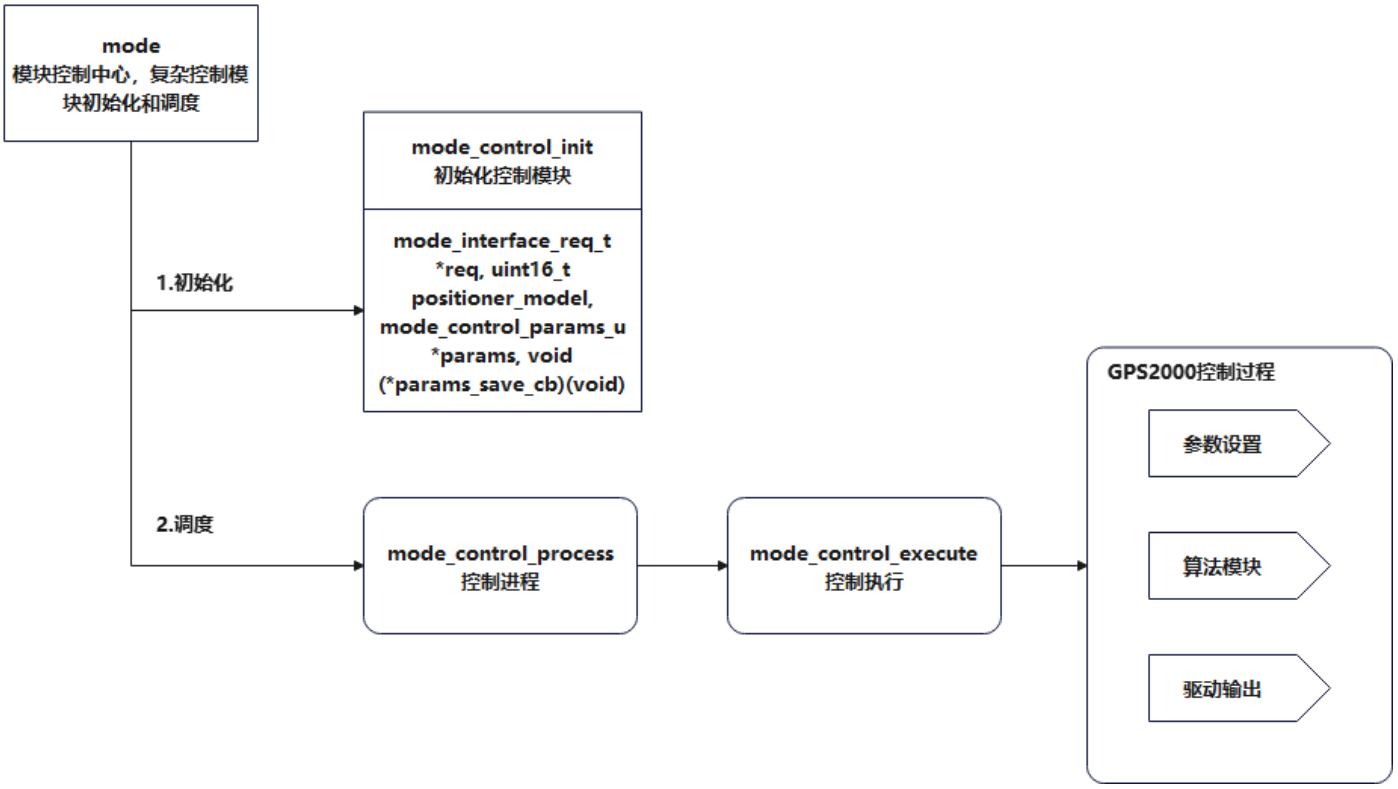
- `update_timer_count`：更新定时器计数器。
- `set_control_arr`：根据定位器模型设置控制参数。
- `get_stable_arr`：根据目标值获取稳定的控制参数。
- `limiting_amplitude`：限制振幅，根据控制过程的状态和方向调整最小和最大输出限制。

更新限制幅度函数：

- `limiting_amplitude_update`：根据目标值和控制输出百分比更新限制幅度的相关参数。

整个模块主要关注于控制过程的启动、停止、调整，以及状态的管理和检查。它通过一系列的函数实现对控制过程的精细管理，包括调整PID控制参数、管理调整步骤、以及根据控制目标和实际输出调整控制输出的限制幅度。此外，模块还包括对定时器的管理和根据不同的定位器模型调整控制参数的功能。

设计



流程

初始化

```
/**
 * @brief 初始化模式控制
 *
 * 初始化模式控制的相关参数和回调函数，并配置相应的接口请求结构体。
 *
 * @param req 模式控制接口请求结构体指针
 * @param positioner_model 定位器型号
 * @param params 模式控制参数结构体指针
 * @param params_save_cb 保存参数回调函数
 */
void mode_control_init(mode_interface_req_t *req, uint16_t positioner_model,
mode_control_params_u *params, void (*params_save_cb)(void))
{
    // 断言 req 不为空，否则在 __DBG_LINE 指定的位置触发断言错误
    DBG_ASSERT(req != NULL __DBG_LINE);
    // 断言 params 不为空，否则在 __DBG_LINE 指定的位置触发断言错误
    DBG_ASSERT(params != NULL __DBG_LINE);
    // 断言 params_save_cb 回调函数不为空，否则在 __DBG_LINE 指定的位置触发断言错误
    DBG_ASSERT(params_save_cb != NULL __DBG_LINE);

    // 设置 req 结构体中的各个函数指针
    req->mode_process_start = _process_start;
    req->mode_process_stop = _process_stop;
    req->mode_adjust_start = _adjust_start;
    req->mode_adjust_stop = _adjust_stop;
    req->mode_get_adjust_data = _adjust_data;
    req->mode_adjust_result = _adjust_result;
    req->mode_adjust_step_count = _adjust_step_count;
    req->mode_adjust_step_current = _adjust_step_current;
    req->mode_control_idle = _control_idle;
    req->mode_is_adjusting = _adjust_isrun;

    // 启用 VIP_H_EN
    VIP_H_EN_ENABLE();

    // 如果 mode_control 为空，则为其分配内存
    if (mode_control == NULL)
    {
        mode_control = (mode_control_t *)osel_mem_alloc(sizeof(mode_control_t));
    }

    // 检查存储结构体的大小是否大于数组长度，如果是，则触发断言错误
    if (sizeof(mode_control_storage_t) > ARRAY_LEN(mode_control->storage_data-
>data))
    {
        DBG_ASSERT(FALSE __DBG_LINE);
    }

    // 将 mode_control 结构体清零
    osel_memset((uint8_t *)mode_control, 0, sizeof(mode_control_t));
```

```
    mode_control->default_min_value = mode_control_get_default_min_value();
    mode_control->control_data.pressure_drain_flag = TRUE;
    mode_control_output(mode_control->default_min_value);
    update_timer_count(MODE_CONTROL_CYCLE);
    mode_control->positioner_model = positioner_model;
    mode_control->storage_data = params;
    mode_control->params_save_cb = params_save_cb;

    // 初始化滤波器相关结构体
    mode_control->filter.target_lpf_window = lpf_window_init(10);
    mode_control->filter.actual_lpf_window = lpf_window_init(10);
    mode_control->filter.ctrl_output_lpf_window = lpf_window_init(10);
    mode_control->filter.ctrl_output_lpf.alpha = 0.1;
    lpf_init(&mode_control->filter.ctrl_output_lpf);

    // 检查 positioner_model 是否超出最大范围
    if (mode_control->positioner_model >= POSITIONER_MODEL_MAX)
    {
        // 如果超出范围，则设置控制状态为控制失败
        mode_control_state_set(CONTROL_PROCESS_CONTROL_FAIL);
    }
    else
    {
        // 如果算法已经校准，则加载 PID 参数并设置其他参数
        if (_algorithm_calibrated_status() == TRUE)
        {
            mode_control_pid_load();
            mode_control_set_parms();
        }
        else
        {
            // 如果算法未校准，则将相关参数标记为未校准状态
            calib_param[CALIBPARA_IPSB].is_calibration = FALSE;
        }

        // 启动模式控制
        mode_control_start();
    }

    // mode_control_state_set(CONTROL_PROCESS_ADJUST);
}
```

控制进程

```
/**
 * @brief 模式控制处理函数
 *
 * 根据mode_control的状态执行不同的处理流程。
 *
 * @param 无
 *
```

```
 * @return 无
 */
void mode_control_process(void)
{
    // 检查mode_control是否为空
    if (mode_control == NULL)
    {
        return;
    }
    // 获取mode_control的控制数据指针
    mode_control_params_t *p = &mode_control->control_data;
    // 进入计数加1
    p->enter_count2++;

    // 根据process_state的值进行不同的处理
    switch (mode_control->process_state)
    {
    case CONTROL_PROCESS_CONTROL: // 控制
    {
        // 如果算法校准状态为TRUE
        if (_algorithm_calibrated_status() == TRUE)
        {
            // 更新mode_control
            mode_control_update();
            // 执行mode_control
            mode_control_execute();
        }
        break;
    }
    case CONTROL_PROCESS_ADJUST: // 整定
        // 更新mode_control
        mode_control_update();
        // 如果获取整定参数为NULL
        if (mode_control_adjust_get() == NULL)
        {
            // 设置整定状态为IDLE
            mode_control->adjust_state = CONTROL_ADJUST_IDEL;
        }
        // 执行整定处理
        mode_control_adjust_process((uint8_t *)&mode_control->adjust_state);
        break;
    case CONTROL_PROCESS_MANUAL: // 手动
    {
        // 更新mode_control
        mode_control_update();
        // 如果手动输出值小于等于DAC_MAX
        if (mode_control->manual_output <= DAC_MAX)
        {
            // 输出手动控制值
            mode_control_output(mode_control->manual_output);
        }

        break;
    }
    case CONTROL_PROCESS_ADJUST_STOP:
    {
        // 停止整定
```

```
        mode_control_adjust_stop(0);
        break;
    }
    case CONTROL_PROCESS_STROKE_TEST:
    {
        // 更新mode_control
        mode_control_update();
        // 如果获取整定参数为NULL
        if (mode_control_adjust_get() == NULL)
        {
            // 设置整定状态为IDLE
            mode_control->adjust_state = CONTROL_ADJUST_IDEL;
        }
        break;
    }
    case CONTROL_PROCESS_STOP:
        // 更新mode_control
        mode_control_update();
        break;
    case CONTROL_PROCESS_CONTROL_FAIL:
        // 控制失败处理
        break;
    default:
        // 默认处理
        break;
    }
}
```

控制执行

```
/**
 * @brief 执行模式控制
 *
 * 根据当前的模式控制参数执行模式控制的相关操作。
 *
 */
void mode_control_execute(void)
{
    mode_control_params_t *p = &mode_control->control_data;

    // 增加进入次数
    p->enter_count1++;

    // 如果进入次数满足条件且当前处理状态为CONTROL_PROCESS_CONTROL (每100ms)
    if (p->enter_count1 % (MODE_CONTROL_CYCLE_BASE * 2 / mode_control-
>wait_count_max) == 0 && mode_control->process_state == CONTROL_PROCESS_CONTROL) //
100ms
    {
        // 通过蓝牙输出控制内容
        mode_control_bluetooth_output_control(); // 通过蓝牙输出 控制内容
    }
```

```
    // 如果控制目标大于上限或小于下限，并且当前处理状态为CONTROL_PROCESS_CONTROL
    if ((p->ctrl_target >= udevice.cutoff_limit_hi || p->ctrl_target <=
udevice.cutoff_limit_lo) &&
        mode_control->process_state == CONTROL_PROCESS_CONTROL)
    {
        // 输出限制
        output_limit();
    }
    else
    {
        // 如果当前处理状态为CONTROL_PROCESS_CONTROL
        if (mode_control->process_state == CONTROL_PROCESS_CONTROL)
        {
            // 设置分程状态PID参数
            set_split_range_state_params(); // 分程状态PID参数
        }

        // 检查稳定状态
        check_stable_state(); // check stable state

        // 算法实现
        switch (mode_control->positioner_model)
        {
        case POSITIONER_MODEL_GPS2000:
            // 控制GPS2000
            control_gps2000();
            break;
        case POSITIONER_MODEL_GPS3000:
            // 控制GPS3000
            control_gps3000();
            break;
        default:
            break;
        }
    }
}
```

GPS2000控制方法

```
/**
 * @brief 控制 GPS2000
 *
 * 该函数用于控制 GPS2000 设备的操作逻辑，根据控制参数执行特定的控制算法，
 * 并输出控制结果。
 */
static void control_gps2000(void)
{
    // 获取控制参数结构体指针
    mode_control_params_t *p = &mode_control->control_data;
    limiting_amplitude_t max_min;
```

```
    // 设置最大最小幅度
    limiting_amplitude(&max_min);

    // 如果刹车标志为真且实际误差小于停止点减去死区
    if (p->brake_flag == TRUE && (ABS(p->real_error) < (MODE_CONTROL_STOP_POINT -
get_dead_zone())))
    {
        // 如果刹车计数小于等待时间（1000个单位时间）
        if (p->brake_count++ < mode_control_get_wait_ticks(1000))
        {
            // 设置补偿标志为真
            p->compensate_flag = TRUE;
            // 设置最大最小幅度为刹车百分比
            max_min.min = p->brake_percent;
            max_min.max = p->brake_percent;
        }
        else
        {
            // 清除步进、刹车、补偿标志
            p->step_flag = FALSE;
            p->brake_flag = FALSE;
            p->compensate_flag = FALSE;
            // 清除刹车计数
            p->brake_count = 0;
        }
    }

    // 设置模糊PID的范围为最大最小幅度
    _pid.pid_u.fuzzy.set_range(&_pid.pid_u.fuzzy, max_min.min, max_min.max);
    // 执行模糊PID计算，得到控制输出
    p->ctrl_output = _pid.pid_u.fuzzy.execute(&_pid.pid_u.fuzzy, p->ctrl_target, p-
>ctrl_feedback);
    // 转换控制输出为百分比数组，并输出
    mode_control_output(mode_control_percent_convert_arr(p->ctrl_output));

    // 如果稳定标志为真且实际误差小于等于死区，且当前处于控制过程
    if (p->stable_flag == TRUE && ABS(p->real_error) <= get_dead_zone() &&
mode_control->process_state == CONTROL_PROCESS_CONTROL)
    {
        // 更新最大最小幅度
        limiting_amplitude_update(p->ctrl_target, p->ctrl_output);
    }
}
```

控制问题

1。 充气和放气的最大值

确定充气和放气的最大值对于PID控制算法的性能至关重要，因为这些值直接影响到控制系统的响应速度和稳定性。如何确定GPS2000这2个值

2。PID参数在30%行程下超调问题