## 0.1 PI

### 0.1.1 Software design of a PI controller

In order to implement the aforementioned PI-controller in a fashion that complies with the API provided by FreeRTOS, it has been chosen to design a single interface. This is to avoid writing multiple sub-programs with the same functionality, and thus, it is required to be generic if multiple PI-controllers are to be implemented.

To meet the requirements, the following design criteria must be fulfilled.

- Must comply with FreeRTOS's API
- Must be re-entrant
- Must be able to be initialized from main
- Must be generic

In order to write a routine that will act as the PI-controller, it must be wrapped into a never ending loop. In this loop the desired calculations will be done. This is done to meet the first criteria and is achieved in listing 1.

**Listing 1:** *Function body*

```c
void tsk_PI(void* pvArgs);

void tsk_PI(void* pvArgs)
{
    PI controller = *((PI*)pvArgs);            //use pvArgs to set up the PI
    controller.conditional_anti = 1;
    int16_t error = 0;

    for (;;)
    {
       error = calculate_error_PI(controller.ref, controller.measured_y_n, &controller );
       calculate_PI( error, &controller );

       vTaskDelay(pdMS_TO_TICKS(10));
```

tsk_PI() will be the argument to the function by FreeRTOS, that creates a task. It is noticed that if the functions, calculate_error_PI() and calculate_P(), are assumed to be re-entrant and the third and fourth criteria are achieved by passing the struct PI from the function task_PI(), all of the above criteria are achieved.

The struct will contain all the necessary gains and data-samples, which a PI-controller needs in order to operate. The struct members can be seen in listing 2.

**Listing 2:** *struct definition*

```c
typedef struct PI_dif_eq
{
    int16_t x_n[3];
    float y_n[3];
    float kp;
    float ki;
    float sample_rate;
    int16_t saturation;
    int16_t* ref;
    int16_t* PWM_out;
    int16_t* measured_y_n;
    int16_t* clamp_ref;
    uint8_t conditional_anti;
}PI;
```

Since the design requirements are met, the next step is to implement the functionality of a PI-controller with the following requirements:

- Must be implemented as a difference equation
- Must include anti-windup

The reasoning for it to be implemented as a difference equation, is due to it being an exact relationship between current and past samples.

### 0.1.2 Deriving the PI difference equation

In order to obtain the difference equation for the PI-controller, the standard PI equation in the s-domain is observed:

$$u(s) = \left(k_p + \frac{k_i}{s}\right) \cdot e(s) \tag{0.1}$$

To get the PI equation into the time-discrete z-domain, it is chosen to use Tustin's Method. This is done by substituting $s$ with the following

$$s = \frac{2}{T} \cdot \left(\frac{z-1}{z+1}\right)$$

Equation (0.1) now becomes

$$u(z) = \left(k_p + k_i \cdot \frac{T}{2}\frac{z+1}{z-1}\right) \cdot e(z)$$

Now the expression on the right side is refactored

$$u(z) = \left(\frac{k_p \cdot 2(z-1) + k_i \cdot T \cdot (z+1)}{2 \cdot (z-1)}\right) \cdot e(z)$$

In order to have the discrete equation in terms of previous samples, both sides of the fraction are divided by $z^{-1}$

$$u(z) = \left(\frac{k_p \cdot (1-z^{-1}) + k_i \cdot \frac{T}{2} \cdot (1+z^{-1})}{(1-z^{-1})}\right) \cdot e(z)$$

It is wished to get an expression in terms of u(z) and e(z) with no non-constant-fractions, therefore both sides are multiplied by $(1-z^{-1})$

$$u(z) \cdot (1-z^{-1}) = \left(k_p \cdot (z-1) + k_i \cdot \frac{T}{2} \cdot (z+1)\right) \cdot e(z)$$

The inverse z-transformation is now performed on both sides

$$u[n] - u[n-1] = k_p \cdot (e[n] - e[n-1]) + k_i \cdot \frac{T}{2}(e[n] + e[n-1])$$

The final difference equation becomes

$$u[n] = u[n-1] + k_p \cdot (e[n] - e[n-1]) + k_i \cdot \frac{T}{2}(e[n] + e[n-1])$$

### 0.1.3 Implementation of the difference equation

With the software design chosen and the difference equation derived, the next step is to implement it. This is done in the following code snippet,seen in listing 3, which includes anti-windup whilst having the PI-controller expressed as a difference equation.

**Listing 3:** *PI controller, with x_n = e[n] and y_n = u[n]*

```c
void calculate_PI(int16_t error, PI* K)
{
    int16_t tmp_output = 0;
    K->x_n[2] = K->x_n[1];
    K->x_n[1] = K->x_n[0];
    K->x_n[0] = error;

    K->y_n[2] = K->y_n[1];
    K->y_n[1] = K->y_n[0];
    K->y_n[0] = K->y_n[1] + ( K->kp * (K->x_n[0] - K->x_n[1]) +
    ( K->conditional_anti * K->ki * K->sample_rate / 2) * (K->x_n[0] + K->x_n[1]) );
    K->conditional_anti = 1;

    tmp_output = K->y_n[0];

    if(tmp_output >= K->saturation)
    {
         tmp_output = K->saturation;
         K->conditional_anti = 0;
    }
    else
    {
        if (tmp_output <= -K->saturation)
        {
            tmp_output = -K->saturation;
            K->conditional_anti = 0;
        }
    }

    *K->PWM_out = tmp_output;
}
```