0.1 Softwaredesign of a PI controller

In order to design and implement the aforementioned PI-controller in a fashion that complies with API from FreeRTOS. It is chosen that a single interface is to be designed so that only one subprogram has to be written, thus it requires to be generic, if multiple multiple PI-controllers are to be implemented. To meet the given requirements the following design criteria must be met:

- 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 which the desired calculations will be done, this is done to meet the first criteria. This is achieved in listing 0.1.

```c
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(100));
    }
}
```

This will be the function which the task created by FreeRTOS will have as an argument. It is noticed that all of the above criteria are met, if the functions calculate_error_PI() and calculate_P() are assumed to be re-entrant. For it to be generic the struct PI, is passed from the function, task_PI(), this also ensures that it can be initialized from main.

The struct will contain all the necessary gains and data-samples, which a PI-controller will need in order to operate. The struct members can be seen in 0.1.

```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 funtionallity of a PI-controller, which has 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 the easiest and most efficient way to compute the controller-signal.

0.2 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 the Tustin's Method.
This is done by substiting s with the following:

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

Equation ref?? now becomes

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