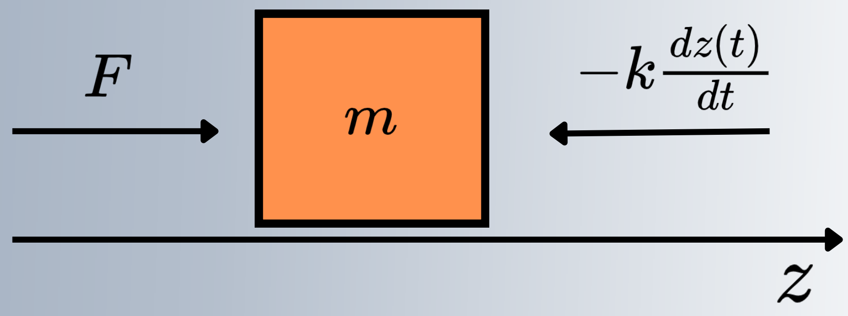
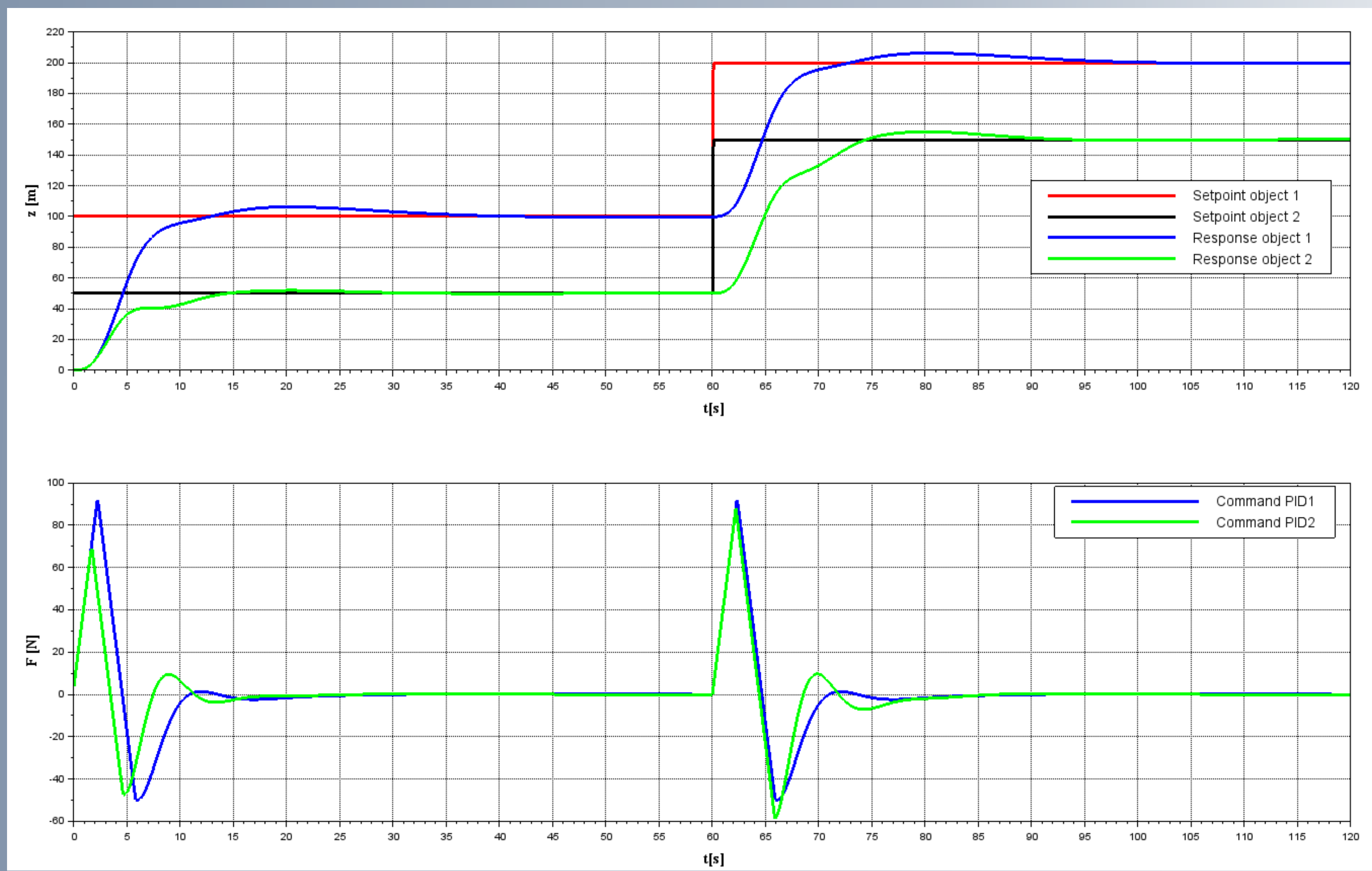
Reusable PID controller in C

```c
struct PID
{
    float Kp;              // Proportional gain constant
    float Ki;              // Integral gain constant
    float Kd;              // Derivative gain constant
    float Kaw;             // Anti-windup gain constant
    float T_C;             // Time constant for derivative filtering
    float T;               // Time step
    float max;             // Max command
    float min;             // Min command
    float max_rate;        // Max rate of change of the command
    float integral;        // Integral term
    float err_prev;        // Previous error
    float deriv_prev;      // Previous derivative
    float command_sat_prev;// Previous saturated command
    float command_prev;    // Previous command
};

struct Object
{
    float m;               // Mass of the object
    float k;               // Damping constant
    float F_max;           // Max force applied to the object
    float F_min;           // Min force applied to the object
    float T;               // Time step
    float v;               // Velocity of the object
    float z;               // Position of the object
};
```

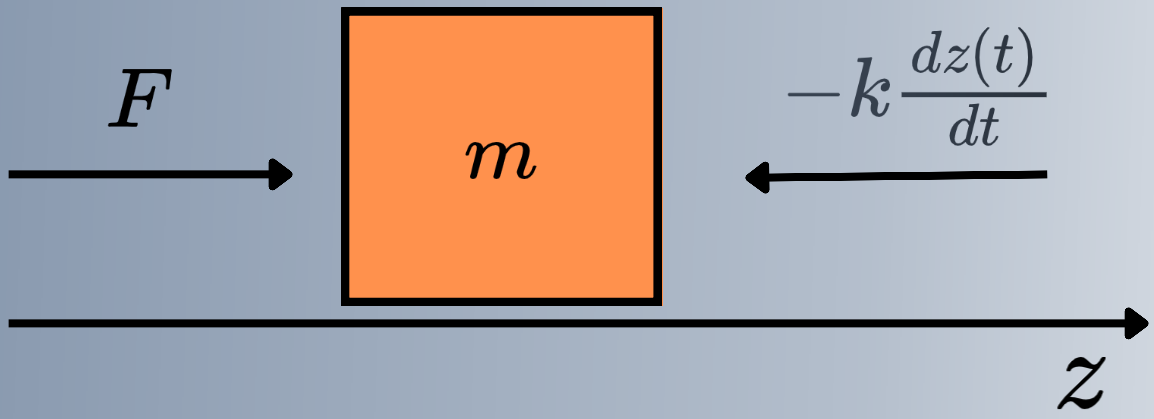




Simone Bertoni

Code
https://github.com/simorxb/PID-C-Struct

Object model



$$m\frac{d^2z(t)}{dt^2} = F - k\frac{dz(t)}{dt}$$

$$m = 10kg$$
$$k = 0.5\frac{Ns}{m}$$

Simone Bertoni

PID and object structure

```
struct PID
{
    float Kp;               // Proportional gain constant
    float Ki;               // Integral gain constant
    float Kd;               // Derivative gain constant
    float Kaw;              // Anti-windup gain constant
    float T_C;              // Time constant for derivative filtering
    float T;                // Time step
    float max;              // Max command
    float min;              // Min command
    float max_rate;         // Max rate of change of the command
    float integral;         // Integral term
    float err_prev;         // Previous error
    float deriv_prev;       // Previous derivative
    float command_sat_prev;// Previous saturated command
    float command_prev;     // Previous command
};

struct Object
{
    float m;                // Mass of the object
    float k;                // Damping constant
    float F_max;            // Max force applied to the object
    float F_min;            // Min force applied to the object
    float T;                // Time step
    float v;                // Velocity of the object
    float z;                // Position of the object
};
```

PID_Step function

```c
float PID_Step(struct PID *pid, float measurement, float setpoint)
{
    /* This function implements a PID controller.
     *
     * Inputs:
     *   measurement: current measurement of the process variable
     *   setpoint: desired value of the process variable
     *   pid: a pointer to a PID struct containing the controller parameters
     *
     * Returns:
     *   command_sat: the control output of the PID controller (saturated based on max. min, max_rate)
     */

    float err;
    float command;
    float command_sat;
    float deriv_filt;

    /* Error calculation */
    err = setpoint - measurement;

    /* Integral term calculation - including anti-windup */
    pid->integral += pid->Ki*err*pid->T + pid->Kaw*(pid->command_sat_prev - pid->command_prev)*pid->T;

    /* Derivative term calculation using filtered derivative method */
    deriv_filt = (err - pid->err_prev + pid->T_C*pid->deriv_prev)/(pid->T + pid->T_C);
    pid->err_prev = err;
    pid->deriv_prev = deriv_filt;

    /* Summing the 3 terms */
    command = pid->Kp*err + pid->integral + pid->Kd*deriv_filt;

    /* Remember command at previous step */
    pid->command_prev = command;

    /* Saturate command */
    if (command > pid->max)
    {
        command_sat = pid->max;
    }
    else if (command < pid->min)
    {
        command_sat = pid->min;
    }
    else
    {
        command_sat = command;
    }

    /* Apply rate limiter */
    if (command_sat > pid->command_sat_prev + pid->max_rate*pid->T)
    {
        command_sat = pid->command_sat_prev + pid->max_rate*pid->T;
    }
    else if (command_sat < pid->command_sat_prev - pid->max_rate*pid->T)
    {
        command_sat = pid->command_sat_prev - pid->max_rate*pid->T;
    }
    else
    {
        /* No action */
    }

    /* Remember saturated command at previous step */
    pid->command_sat_prev = command_sat;

    return command_sat;
}
```

Simone Bertoni

Object_Step function

```c
float Object_Step(struct Object *obj, float F){

    /* This function updates the position of an object in 1D based on the applied force F and
     * the object's mass, viscous damping coefficient k, max/min forces, and time step T.
     *
     * Inputs:
     *   F: the force applied to the object
     *   obj: a pointer to an object struct containing its properties (mass, damping, etc.)
     *
     * Returns:
     *   z: the position of the object in meters
     */

    /* Declare variables for the derivative dv/dt and the saturated force command */
    float dv_dt;
    float F_sat;

    /* Apply saturation to the input force */
    if (F > obj->F_max)
    {
        F_sat = obj->F_max;
    }
    else if (F < obj->F_min)
    {
        F_sat = obj->F_min;
    }
    else
    {
        F_sat = F;
    }

    /* Calculate the derivative dv/dt using the input force and the object's velocity and properties */
    dv_dt = (F_sat - obj->k*obj->v)/obj->m;

    /* Update the velocity and position of the object by integrating the derivative using the time step T */
    obj->v += dv_dt*obj->T;
    obj->z += obj->v*obj->T;

    /* Return the updated position of the object */
    return obj->z;
}
```

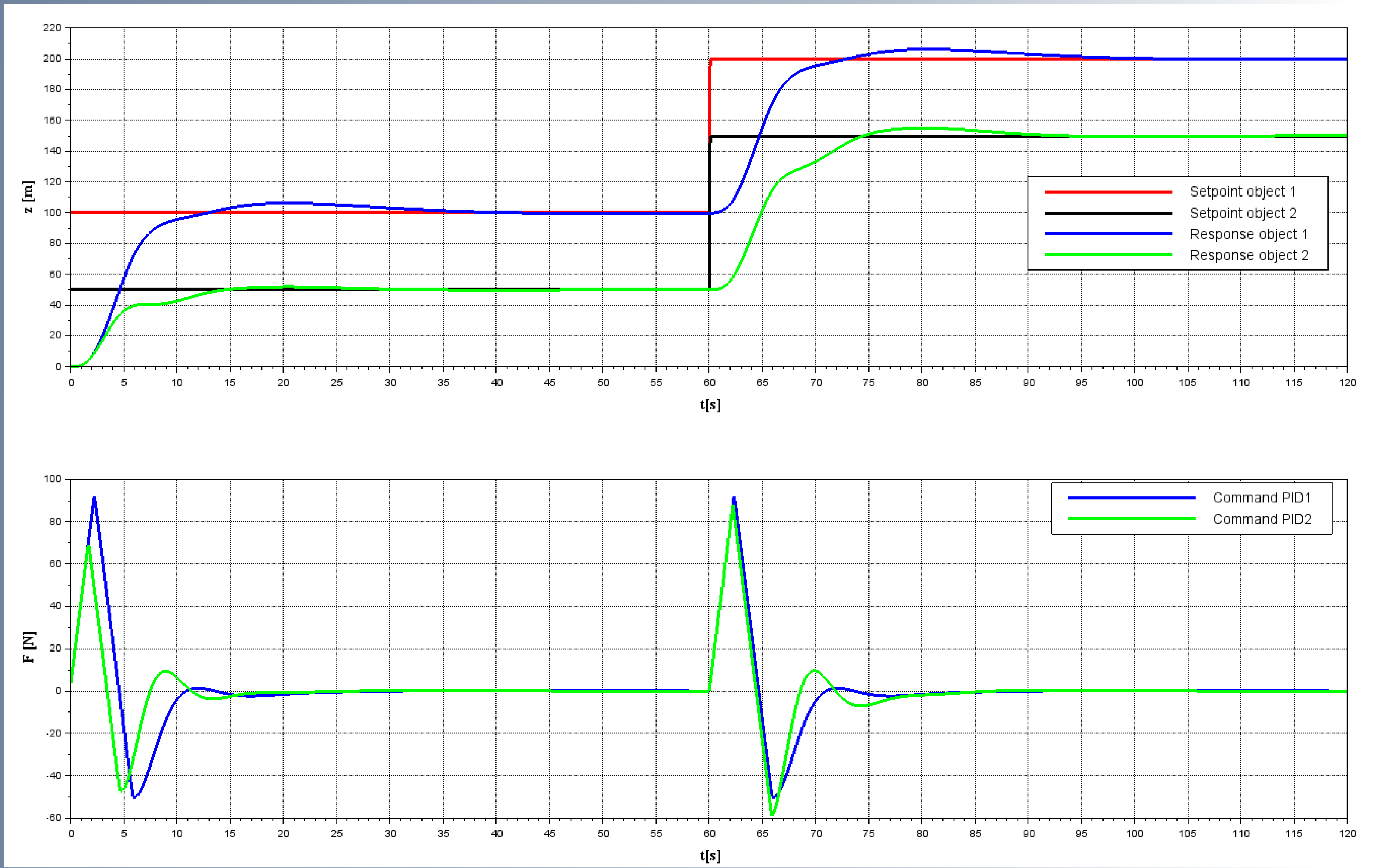
Simone Bertoni

Main function

```c
int main()
{
    // Current simulation time
    float t = 0;

    // Iteration counter
    int i = 0;

    // Setpoint and output of the first control loop
    float command1 = 0;
    float stp1 = 100;
    float z1 = 0;

    // Setpoint and output of the second control loop
    float command2 = 0;
    float stp2 = 50;
    float z2 = 0;

    // PID controller parameters for the first control loop
    struct PID pid1 = {1, 0.1, 5, 0.1, 1, TIME_STEP, 100, -100, 40, 0, 0, 0, 0, 0};

    // Object parameters for the first control loop
    struct Object obj1 = {10, 0.5, 100, -100, TIME_STEP, 0, 0};

    // PID controller parameters for the second control loop
    struct PID pid2 = {1.8, 0.3, 7, 0.3, 1, TIME_STEP, 100, -100, 40, 0, 0, 0, 0, 0};

    // Object parameters for the second control loop
    struct Object obj2 = {10, 0.5, 100, -100, TIME_STEP, 0, 0};

    // Open a file for logging simulation data
    FILE *file = fopen("data_PID_C.txt", "w");

    /* Implement iteration using a while loop */
    while(i < LENGTH)
    {
        /* Change setpoint at t = 60 seconds */
        if (t < 60)
        {
            stp1 = 100;
            stp2 = 50;
        }
        else
        {
            stp1 = 200;
            stp2 = 150;
        }

        // Execute the first control loop
        command1 = PID_Step(&pid1, z1, stp1);
        z1 = Object_Step(&obj1, command1);

        // Execute the second control loop
        command2 = PID_Step(&pid2, z2, stp2);
        z2 = Object_Step(&obj2, command2);

        // Log the current time and control loop values to the file
        fprintf(file, "%f %f %f %f %f %f %f\n", t, command1, z1, stp1, command2, z2, stp2);

        // Increment the time and iteration counter
        t = t + TIME_STEP;
        i = i + 1;
    }

    // Close the file and exit the program
    fclose(file);
    exit(0);
}
```

Simone Bertoni

Simulation result



Simone Bertoni