Controlling and regulating the cooling of a RPi with a PID controller

Generated by Doxygen 1.9.4

Chapter 1

Controlling and regulating the cooling of a RPi with a PID controller

English version below !

1.1 Asservissement et régulation du refroidissement d'une RPi avec un correcteur PID

Le but de ce projet est d'asservir et de réguler le refroidissement d'une Raspberry Pi (RPi). Pour ce faire, j'ai acheté le kit suivant : `Kit RPi4 acheté`.

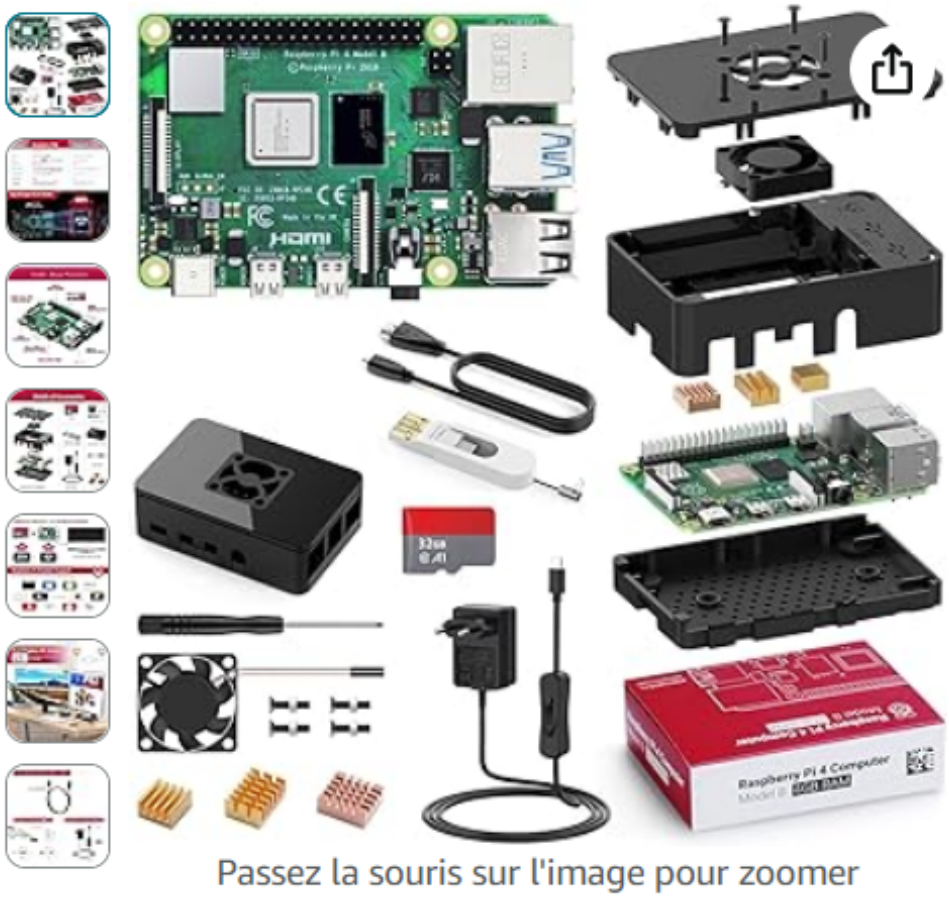


Figure 1.1 Kit Raspberry Pi 4 utilisé, image venant d'Amazon

1.1.1 Sommaire

- Réalisation d'un module de commande du ventilateur
- Analyse du système : moteur DC, servant de ventilateur
- Calcul de la FTBO, de la FTBF avec le choix d'un correcteur PI
- Réalisation d'un programme C++ complet pour l'asservissement
- Réalisation du programme final
- Création du service "ventilateur.service" sur la RPi

1.1.2 Réalisation d'un module de commande du ventilateur

Le moteur DC du ventilateur consomme à sa pleine puissance plus de courant que peut en fournir le GPIO (General Purpose Input Output) de la Raspberry Pi 4. Il est donc nécessaire de réaliser un petit montage électronique pour l'adaptation de puissance :

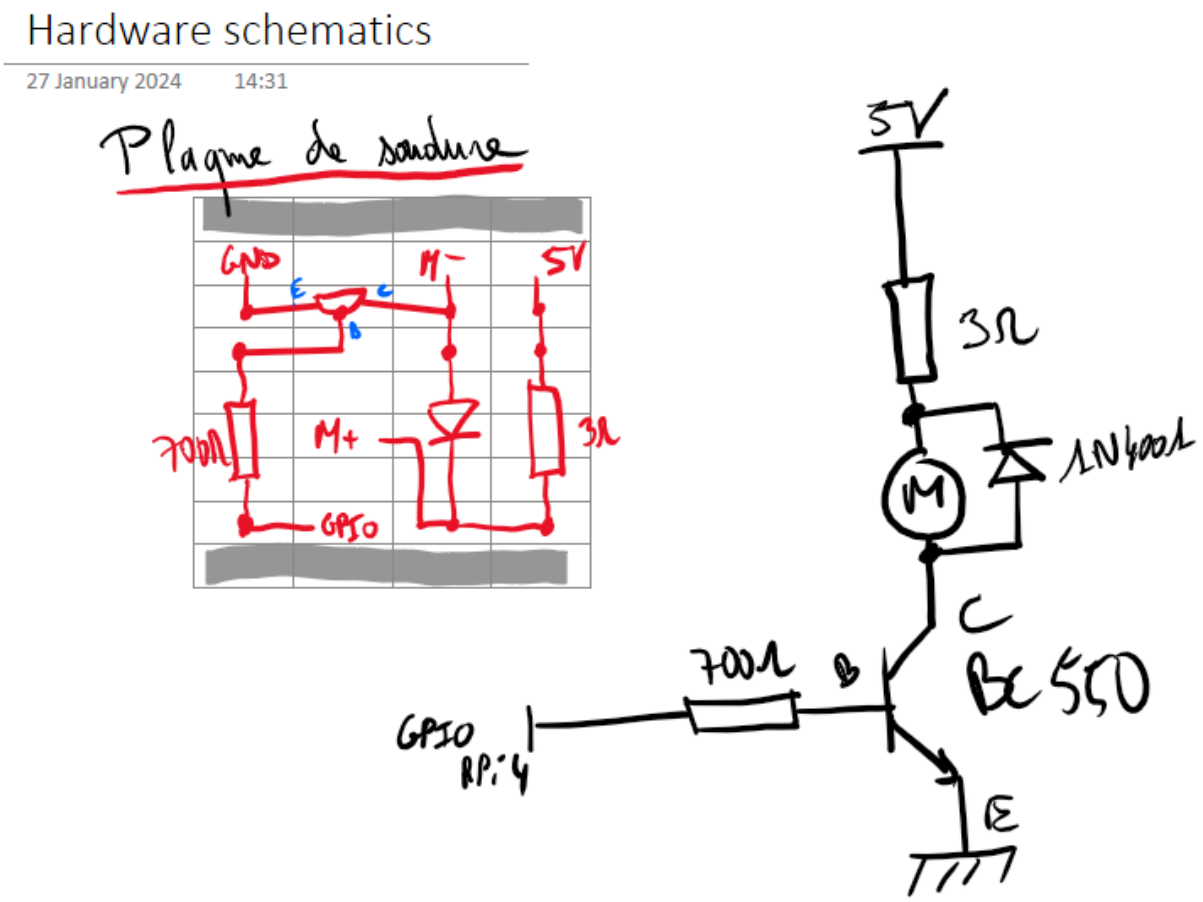


Figure 1.2 Schéma électronique du module de commande du ventilateur

La diode est placée ici comme diode de roue libre pour éviter une tension de contre-électromotrice qui endommagerait les composants autour, notamment le transistor bipolaire.

> "La diode de roue libre joue un rôle crucial dans la protection des circuits électroniques en fournissant un chemin de moindre résistance pour le courant induit lors de l'arrêt d'une bobine électromagnétique, prévenant ainsi les dommages causés par les pics de tension."

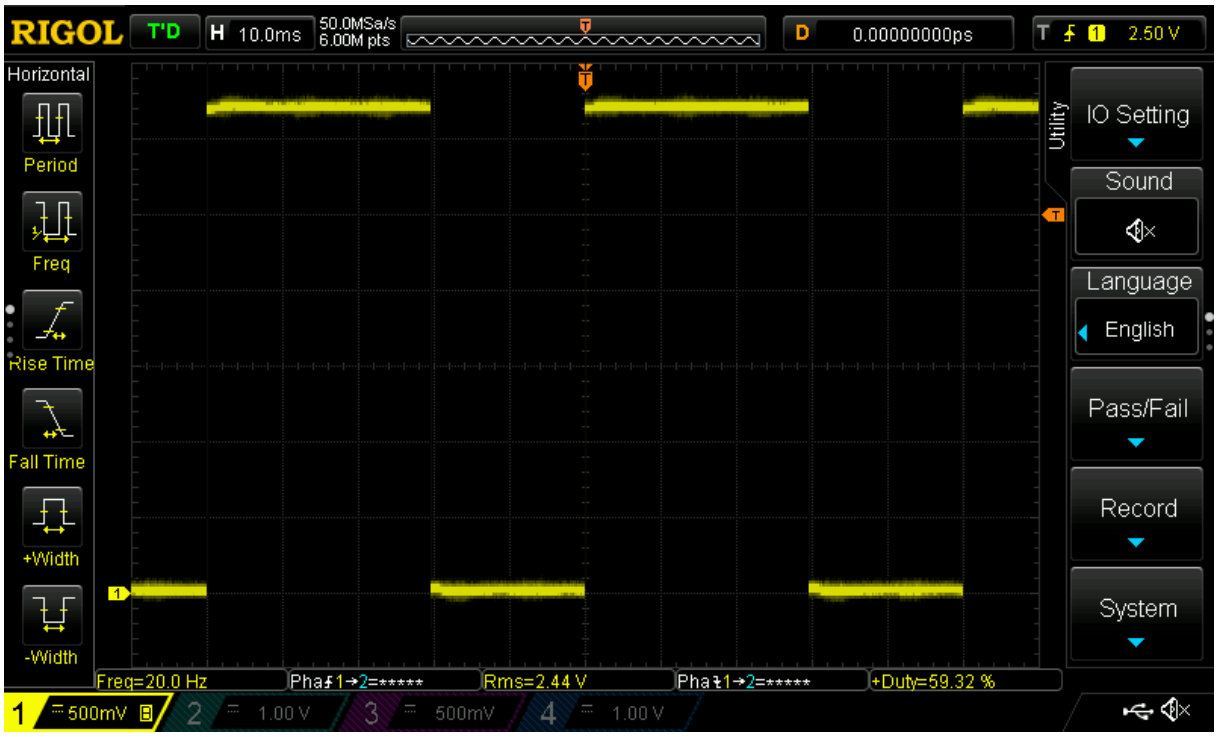


Figure 1.3 Exemple d'un signal PWM

Nous pouvons désormais contrôler notre moteur en PWM (Pulse Width Modulation) ou MLI (Modulation par Largeur d'Impulsion), ce qui nous permettra de contrôler sa force électromotrice (fem) donc sa vitesse, à l'aide d'un signal TOR (Tout Ou Rien) en modulant la largeur temporelle de l'impulsion (de période fixe).

Sur ce signal, nous pouvons observer :

- Une fréquence de 20Hz (donc la période = 1/20 = 50ms)
- Un rapport cyclique de $\sim$60%
- Une tension minimale de 0V
- Une tension maximale de 3.3V

Nous pouvons en déduire une valeur moyenne d'environ $0.6 * (3.3 - 0) = 1.98$V, ce qui correspondra à la tension perçue par le moteur du ventilateur puisque le moteur DC peut-être assimilé à un filtre passe-bas grâce à sa bobine.

1.1.3 Analyse du système : moteur DC, servant de ventilateur

Pour analyser le système, nous allons appliquer un échelon en entrée et observer sa réponse. Concrètement, nous allons laisser monter en température la RPi4 avec une utilisation CPU constante jusqu'à ce qu'elle tende vers une température constante, puis nous allons activer le ventilateur avec une commande maximale.

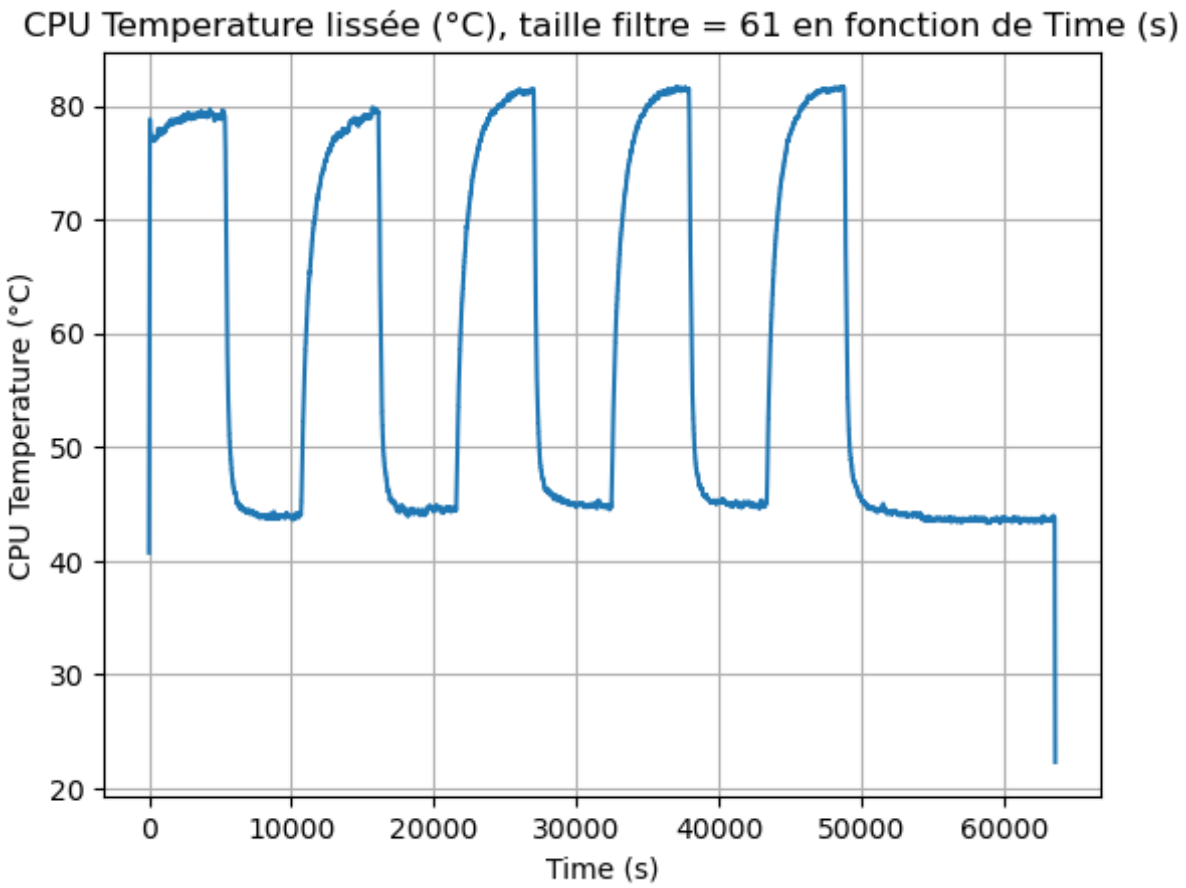


Figure 1.4 Réponse indicielle du moteur DC

Voici les résultats obtenus avec mon ventilateur sur 5 cycles. J'ai appliqué un filtre passe bas sur le signal enregistré car la mesure du capteur avait du bruit. J'ai réalisé un filtre numérique à Réponse Impulsionnelle Finie (RIF) à 61 points, déduit par tâtonnement. La mesure du temps de réponse de notre système est alors bien plus simple à réaliser et bien plus précise.

L'objectif d'avoir 5 cycles est d'avoir une meilleure précision sur la lecture du temps de réponse du système, qui nous indique son comportement.

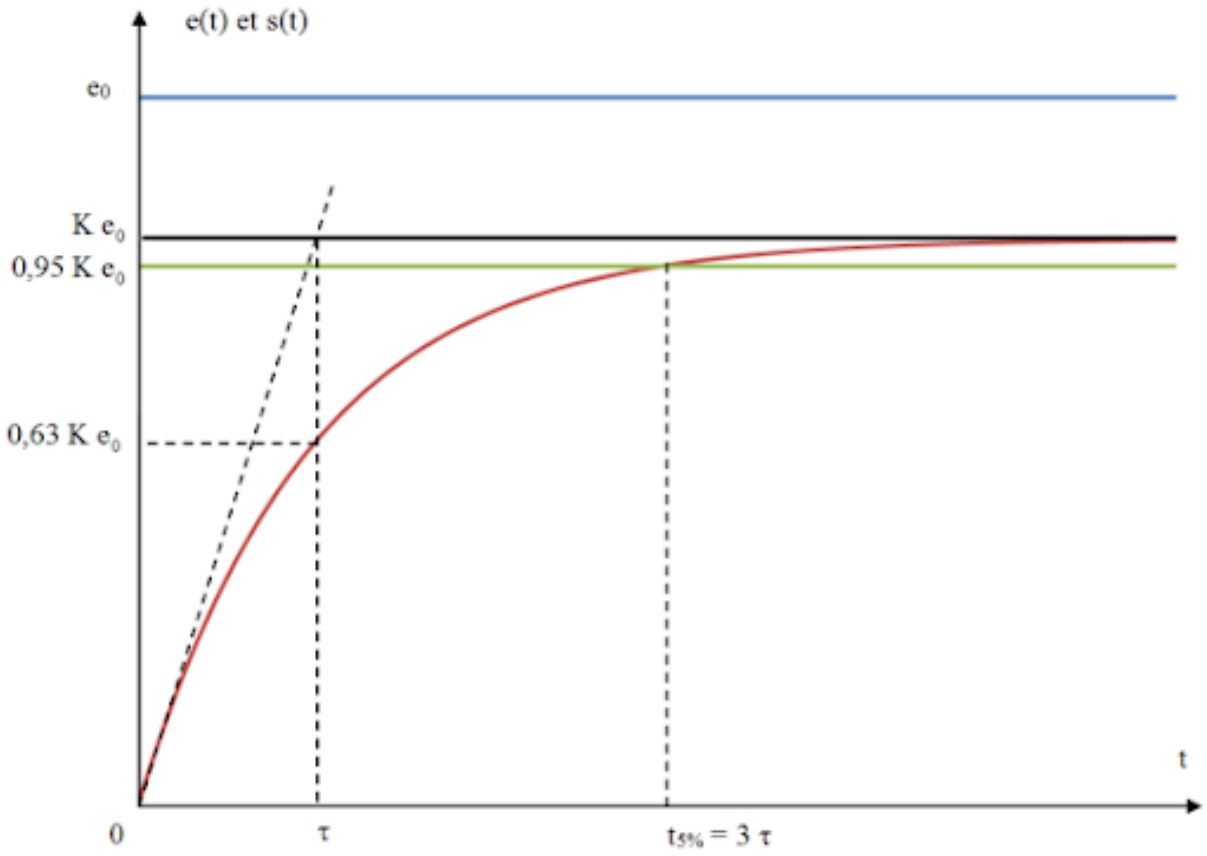


Figure 1.5 Temps de réponse d'un système du premier ordre

À partir de l'analyse de ces réponses indicielles, nous pouvons assimiler notre moteur DC à un système du premier ordre puisqu'il n'a pas de dépassement de sa valeur finale. Voici les temps de réponse obtenus sur les 5 cycles :

```
Cycle 1 : Temps de réponse = 226.0 s
Cycle 2 : Temps de réponse = 196.0 s
Cycle 3 : Temps de réponse = 213.0 s
Cycle 4 : Temps de réponse = 227.0 s
Cycle 5 : Temps de réponse = 229.0 s
```

Figure 1.6 Temps de réponse des 5 cycles

Nous choisirons **tau_FTBO = 230s** car qui peut le plus, peut le moins.

Pour ce qui est du tau_FTBF, nous devons respecter la condition **tau_FTBO >= tau_FTBF**. Je choisis donc **tau↩ _FTBF = 300s**.

1.1.4 Calcul de la FTBO, de la FTBF avec le choix d'un correcteur PI

En pratique, l'ajout d'une action dérivée dans un correcteur PID n'est pas toujours nécessaire pour asservir un moteur DC. Dans de nombreux cas, un correcteur PI bien réglé peut fournir des performances satisfaisantes.

Voici le diagramme bloc que nous obtenons pour notre système en boucle fermée :

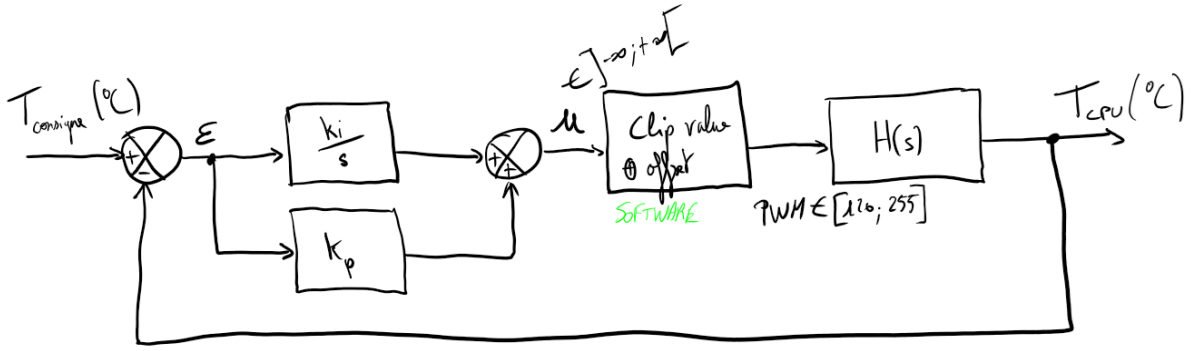


Figure 1.7 Diagramme bloc du système en boucle fermée

Pour le détail des calculs, à partir de la Fonction de Transfert en Boucle Ouverte (FTBO) et de la Fonction de Transfert en Boucle Fermée (FTBF), pour obtenir les coefficients de ki et kp de notre correcteur PI, je vous invite à consulter mes calculs sur ma feuille de calcul.

1.1.5 Réalisation d'un programme C++ complet pour l'asservissement

Passons maintenant à la partie programmation ! J'ai réalisé un programme C++ permettant de réaliser le correcteur PID, de lire la température du CPU, de lire l'utilisation du CPU (en %) et d'enregistrer les performances du correcteur PI dans un fichier CSV. Ce fichier CSV peut ensuite être analysé par un petit script Python que j'ai réalisé sur un Jupyter Notebook. Voici les performances obtenues :

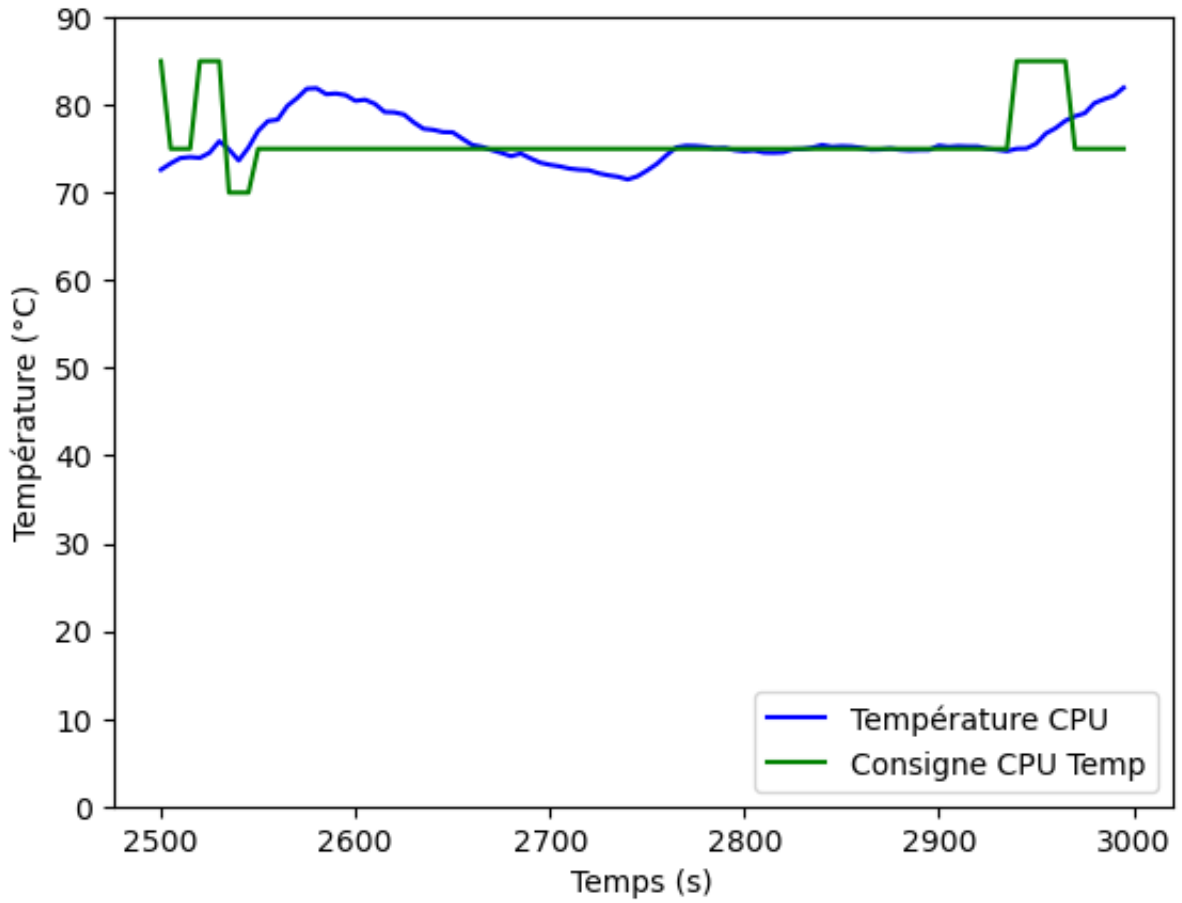


Figure 1.8 Performances du correcteur PI

Remarques :

- La consigne de température varie en fonction du pourcentage d'utilisation du CPU puisque le ventilateur n'a pas la capacité à maintenir une température basse avec une utilisation du CPU élevée.
- Le ventilateur n'agit que sur la décroissance de la température du CPU, ce qui implique que nous utilisons l'inertie thermique du CPU pour la croissance de la température du CPU !

Dans le code, j'ai utilisé différentes boucles "for" imbriquées pour optimiser l'utilisation du CPU et éviter les erreurs de mesure de la sonde de température du CPU.

- La première permet de ne pas effectuer trop de vidages du vecteur dans le fichier CSV dans un laps de temps, ce qui permet de diminuer significativement l'utilisation du CPU
- La seconde permet de scanner 3 fois moins souvent l'utilisation du CPU qu'une période d'application d'une valeur PWM, cela permet de ne pas changer la consigne de température trop vite, et donc que l'action intégrale ne s'emballe pas sur une impulsion.
- La troisième permet de lire 10 fois plus vite la température du CPU (puis de faire la moyenne), pour filtrer les erreurs de mesure de la sonde de température du CPU.

1.1.5.1 Partie de tests

1.1.5.1.1 1. Asservissement :

> "Une grandeur physique doit suivre une évolution temporelle imposée par l'utilisateur du système.", Livre : Régulation industrielle, édition Dunod

Nous allons garder une utilisation CPU basse (peu de perturbations) et analyser les performances du système en statique (erreur statique) et en dynamique (temps de réponse, dépassement, stabilité).

1.1.5.1.2 2. Régulation :

> "Une grandeur physique doit être maintenue à un niveau constant en dépit de la présence de perturbations.", Livre : Régulation industrielle, édition Dunod

Nous allons augmenter l'utilisation du CPU, sur les différentes plages d'utilisation du CPU (0-25%, 25-50%, 50-75% et 75-100%), pour créer des perturbations. Nous pouvons effectuer l'augmentation de l'utilisation du CPU en lançant un ou plusieurs scripts de machine learning sur la RPi4. Ces scripts viennent de ce dépôt lié au livre d'Aurélien Géron : Hands-On Machine Learning with Scikit-Learn, Keras, and Tensor↩Flow, pour apprendre le ML.

Nous allons également analyser les performances du système en statique (erreur statique) et en dynamique (temps de réponse, dépassement, stabilité).

C'est sur cette partie principalement que nous allons, par tâtonnement, ajuster les 2 coefficients de notre correcteur PI pour que le système en boucle fermée réponde à nos exigences !

1.1.6 Réalisation du programme final

Pour réaliser ce programme final, qui fonctionnera en continu sur notre RPi, nous allons reprendre le code complet réalisé précédemment, y supprimer la partie enregistrement dans un fichier CSV et y rajouter une condition d'activation de l'asservissement que s'il est entre 7h et 00h (puisque je dors après) !

1.1.7 Création du service "ventilateur.service" sur la RPi

Après compilation du programme C++ final, nous pouvons déplacer notre fichier binaire dans notre répertoire /opt (dédié pour cela) et le renommer :

```
> ~/PID_control_RPi_cooling/PID_control_ventilator $ mv main.bin /opt
> ~/PID_control_RPi_cooling/PID_control_ventilator $ cd /opt
> /opt $ mv main.bin PI_ventilateur.bin
```

Nous pouvons ensuite créer notre service :

```
> ~ $ sudo nano /etc/systemd/system/ventilateur.service
```

Il faut alors éditer votre service (exemple : le mien) :

```
[Unit]
Description=Run Ventilator controlled with PI
After=multi-user.target
[Service]
ExecStart=/opt/PI_ventilateur.bin
Restart=always
[Install]
WantedBy=multi-user.target
```

Puis l'enregistrer (Ctrl+X puis taper "y" puis appuyer 2 fois sur la touche "Entrée").

Nous pouvons alors activer notre service et le mettre en service :
```
> ~ $ systemctl enable ventilateur.service
> ~ $ systemctl start ventilateur.service
```

Pour connaître l'état de notre service :
```
> ~ $ systemctl status ventilateur.service
```

Pour arrêter notre service :
```
> ~ $ systemctl stop ventilateur.service
```

En espérant que vous ayez trouvé cette description complète et concise et que le projet vous a plu !

1.2 PID Control and Regulation of RPi Cooling

The aim of this project is to control and regulate the cooling of a Raspberry Pi (RPi) using a PID controller. To do this, I purchased the following kit: RPi4 Kit.

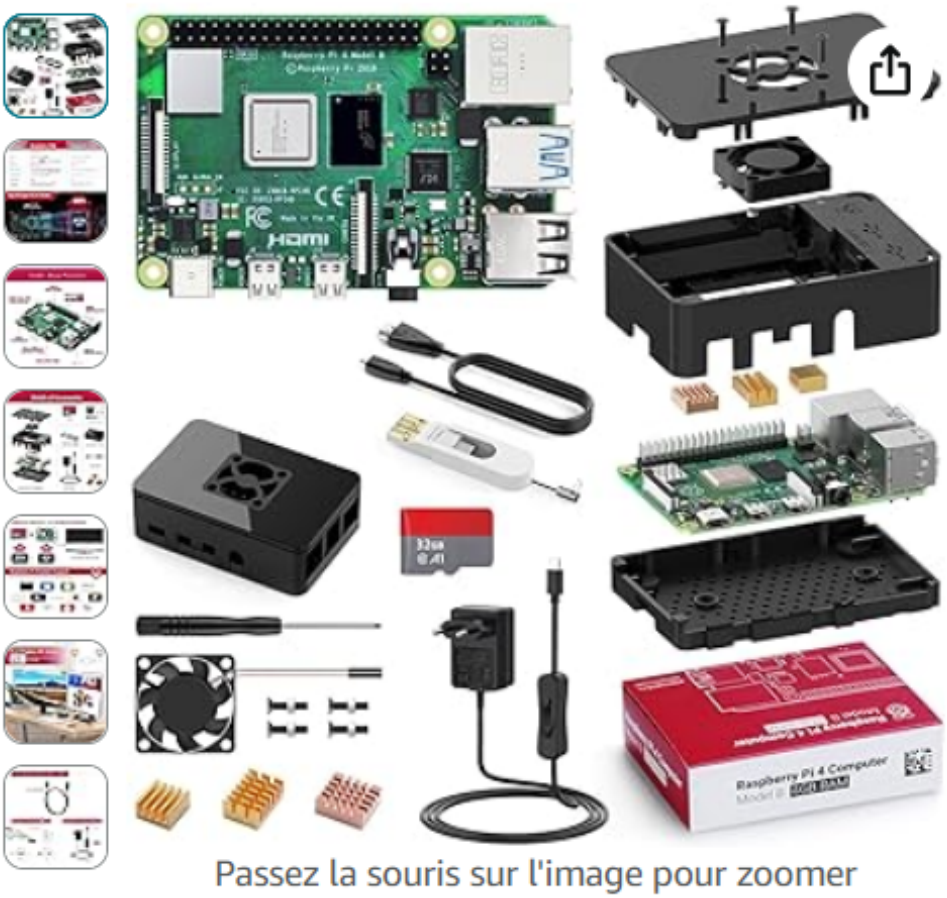


Figure 1.9 Raspberry Pi 4 kit used, image from Amazon

1.2.1 Table of contents

- Design of a ventilator control module
- System Analysis: DC Motor as a ventilator

- Calculation of the FTBO, FTBF with the choice of a PI controller
- Complete C++ Program for Control
- Final Program Realization
- Creation of the "ventilateur.service" service on the RPi

1.2.2 Design of a ventilator control module

The DC motor of the ventilator consumes more current at full power than the GPIO (General Purpose Input Output) of the Raspberry Pi 4 can provide. Therefore, it is necessary to make a small electronic circuit for power adaptation:

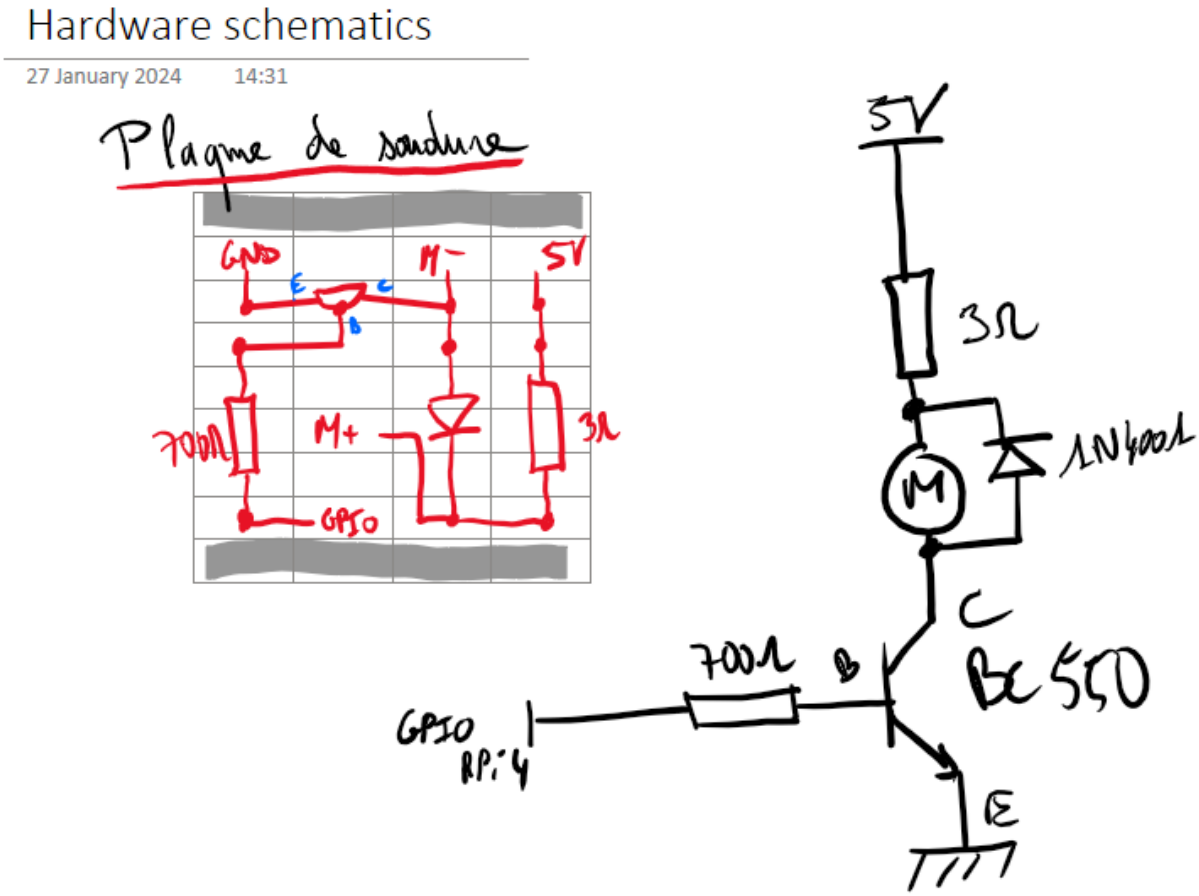


Figure 1.10 Electronic schematic of the ventilator control module

The diode is placed here as a freewheeling diode to prevent a counter-electromotive force that could damage the surrounding components, especially the bipolar transistor.

> "The freewheeling diode plays a crucial role in protecting electronic circuits by providing a path of least resistance for the induced current when an electromagnetic coil stops, thus preventing damage caused by voltage spikes."

We can now control our motor using PWM (Pulse Width Modulation), which will allow us to control its electromotive force and therefore its speed, using a binary signal by modulating the temporal width of the pulse (of fixed period). On this signal, we can observe:

- A frequency of 20Hz (therefore the period = 1/20 = 50ms)
- A duty cycle of $\sim$60%
- A minimum voltage of 0V

- A maximum voltage of 3.3V

We can deduce an average value of approximately 0.6 $*$ (3.3 - 0) = 1.98V, which will correspond to the voltage perceived by the ventilator motor since the DC motor can be assimilated to a low-pass filter thanks to its coil.

1.2.3 System Analysis: DC Motor as a ventilator

To analyze the system, we will apply a step input and observe its response. Specifically, we will let the RPi4 heat up with constant CPU usage until it tends towards a constant temperature, and then we will activate the ventilator with a maximum command.

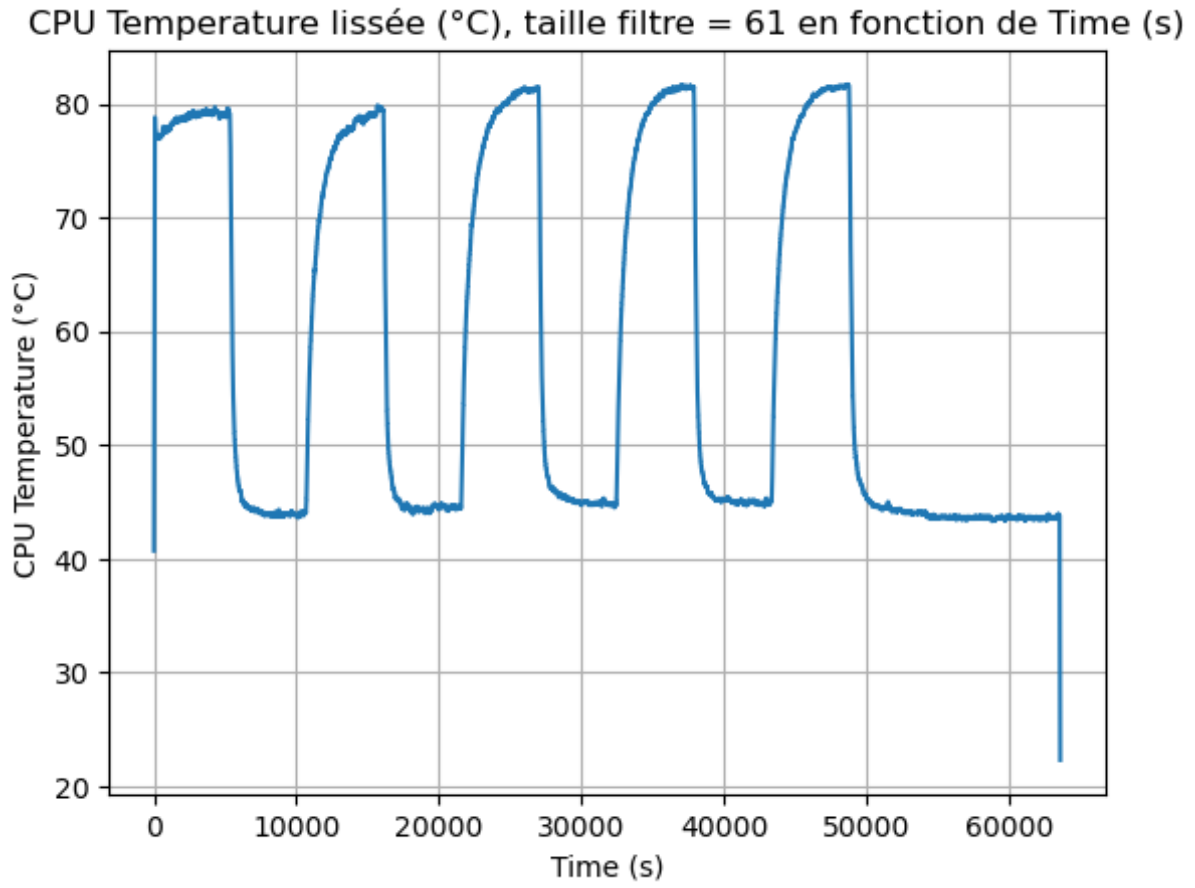


Figure 1.11 Step response of the DC motor

Here are the results obtained with my ventilator over 5 cycles. I applied a low-pass filter to the recorded signal because the sensor measurement had noise. I made a 61-point Finite Impulse Response (FIR) digital filter, deduced by trial and error. The measurement of the response time of our system is then much simpler and more accurate.
The objective of having 5 cycles is to have better precision in reading the response time of the system, which indicates its behavior.

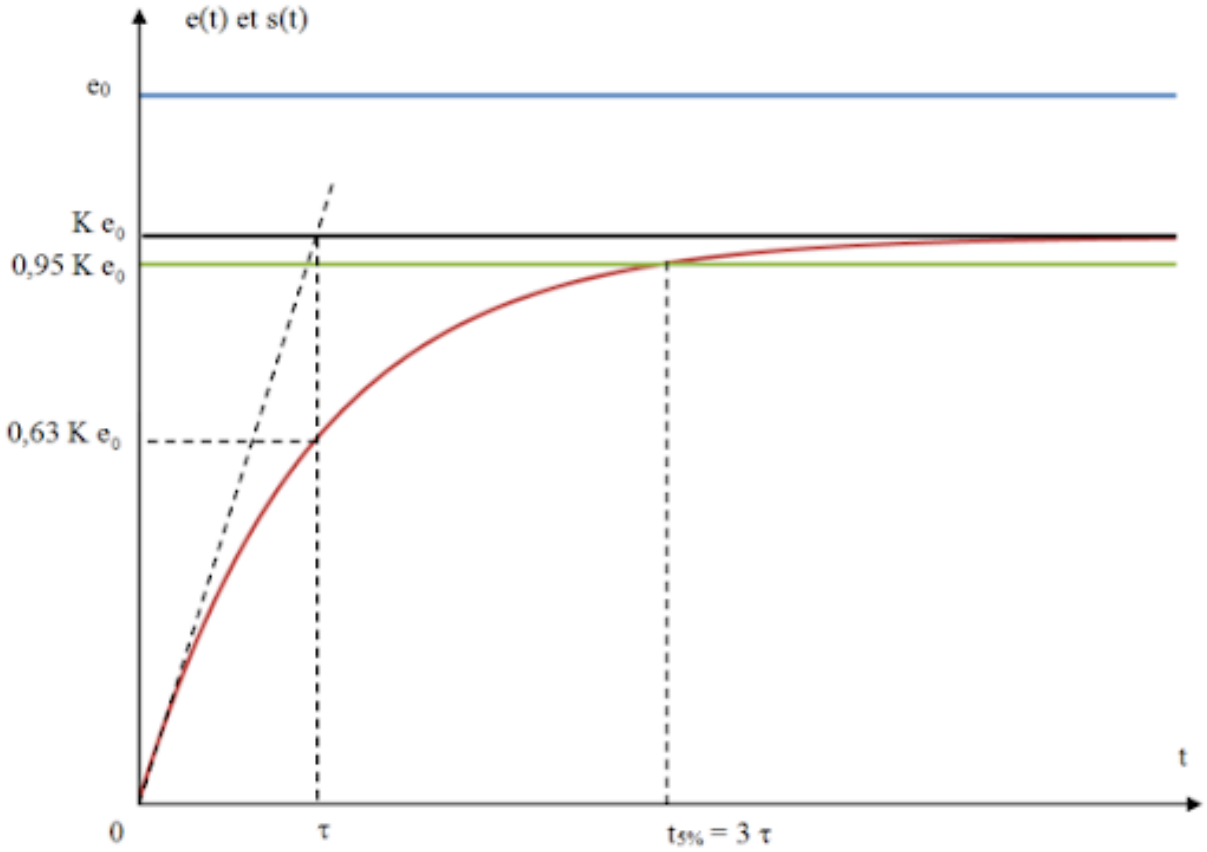


Figure 1.12 Response time of a first-order system

Based on the analysis of these step responses, we can consider our DC motor as a first-order system since it does not have any overshoot of its final value. Here are the response times obtained over the 5 cycles:

```
Cycle 1 : Temps de réponse = 226.0 s
Cycle 2 : Temps de réponse = 196.0 s
Cycle 3 : Temps de réponse = 213.0 s
Cycle 4 : Temps de réponse = 227.0 s
Cycle 5 : Temps de réponse = 229.0 s
```

Figure 1.13 Response times of the 5 cycles

We will choose **tau_FTBO = 230s** because who can do more, can do less.
As for tau_FTBF, we must respect the condition **tau_FTBO** >**= tau_FTBF**. I choose **tau_FTBF = 300s**.

1.2.4 Calculation of the FTBO, FTBF with the choice of a PI controller

In practice, adding a derivative action in a PID controller is not always necessary to control a DC motor. In many cases, a well-tuned PI controller can provide satisfactory performance.
Here is the block diagram we obtain for our closed-loop system:

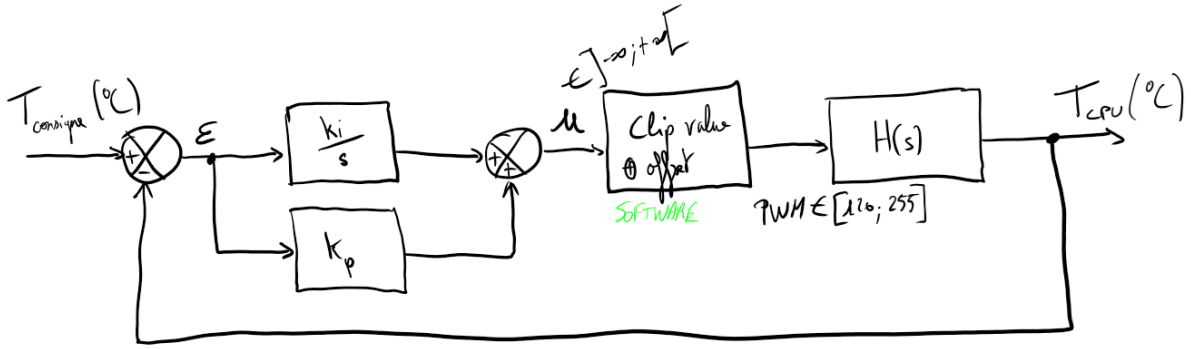


Figure 1.14 Block diagram of the closed-loop system

For the detail of the calculations, from the Open-Loop Transfer Function (FTBO) and the Closed-Loop Transfer Function (FTBF), to obtain the ki and kp coefficients of our PI controller, I invite you to consult my calculations on my calculation sheet.

1.2.5 Complete C++ Program for Control

Now let's move on to the programming part! I made a C++ program that allows to implement the PID controller, to read the CPU temperature, to read the CPU usage (in %) and to record the performance of the PI controller in a CSV file. This CSV file can then be analyzed by a small Python script that I made on a Jupyter Notebook. Here are the performances obtained:

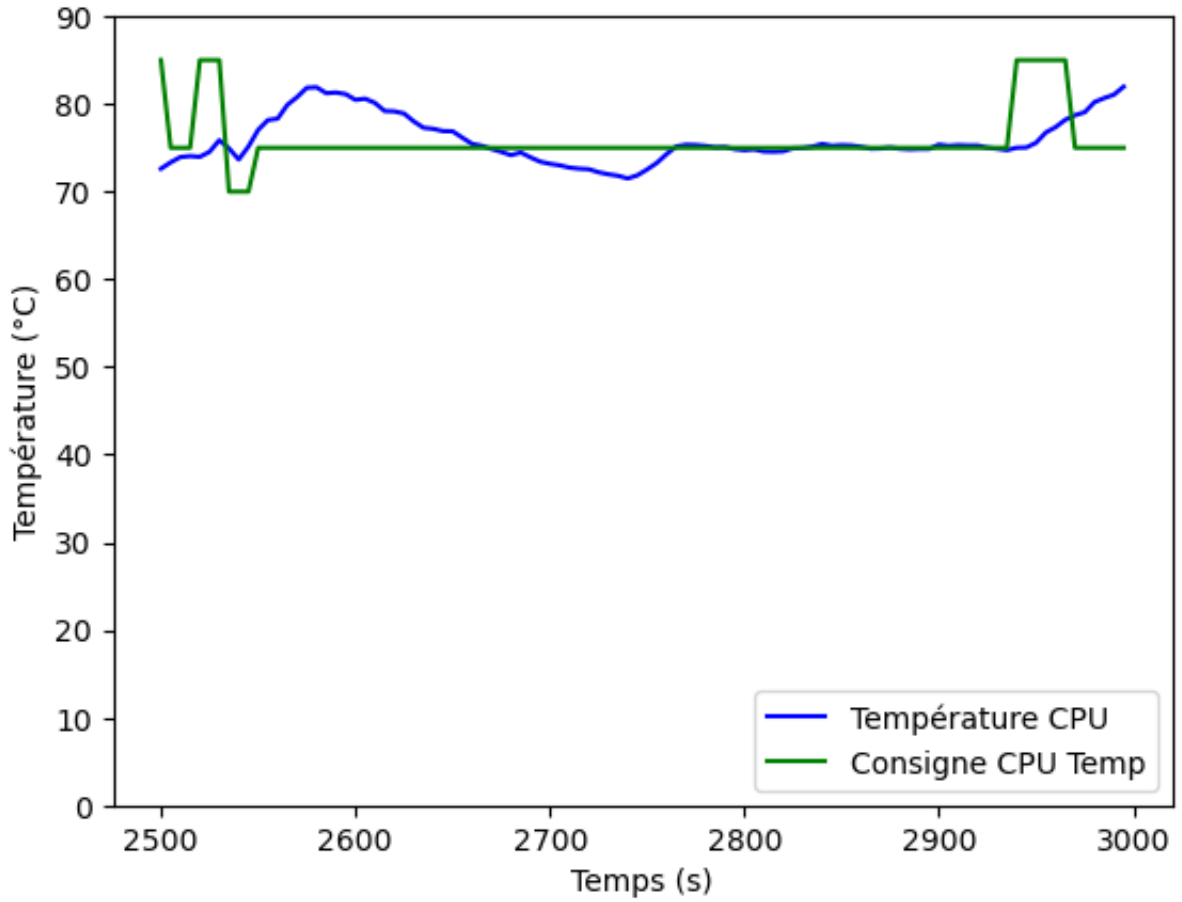


Figure 1.15 Performance of the PI controller

Notes:

- The temperature setpoint varies depending on the CPU usage percentage since the ventilator does not have the capacity to maintain a low temperature with high CPU usage.
- The ventilator only acts on the decrease of the CPU temperature, which implies that we use the thermal inertia of the CPU for the increase of the CPU temperature!

In the code, I used different nested "for" loops to optimize CPU usage and avoid measurement errors of the CPU temperature sensor.

- The first one allows not to perform too many vector empties in the CSV file in a short time, which allows to significantly decrease the CPU usage.
- The second one allows to scan 3 times less often the CPU usage than a PWM value application period, this allows not to change the temperature setpoint too quickly, and therefore that the integral action does not get out of control on an impulse.
- The third one allows to read the CPU temperature 10 times faster (and then make the average), to filter the measurement errors of the CPU temperature sensor.

1.2.5.1 Test Part

1.2.5.1.1 1. Control:

> "A physical quantity must follow a temporal evolution imposed by the user of the system.", Book: Régulation Industrielle, Dunod edition

We will keep a low CPU usage (few disturbances) and analyze the performance of the system in static (static error) and dynamic (response time, overshoot, stability).

1.2.5.1.2 2. Regulation:

> "A physical quantity must be maintained at a constant level despite the presence of disturbances.", Book: Régulation Industrielle, Dunod edition

We will increase the CPU usage, on the different CPU usage ranges (0-25%, 25-50%, 50-75% and 75-100%), to create disturbances. We can increase the CPU usage by launching one or more machine learning scripts on the RPi4. These scripts come from this repository linked to Aurélien Géron's book: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, to learn ML.
We will also analyze the performance of the system in static (static error) and dynamic (response time, overshoot, stability).
It is mainly on this part that we will adjust, by trial and error, the 2 coefficients of our PI controller so that the closed-loop system meets our requirements!

1.2.6 Final Program Realization

To make this final program, which will run continuously on our RPi, we will take the complete code made previously, remove the part of recording in a CSV file and add a condition of activation of the control if it is between 7h and 00h (since I sleep after)!

1.2.7 Creation of the "ventilateur.service" service on the RPi

After compiling the final C++ program, we can move our binary file to our /opt directory (dedicated for this) and rename it:

```
> ~/PID_control_RPi_cooling/PID_control_ventilator $ mv main.bin /opt
> ~/PID_control_RPi_cooling/PID_control_ventilator $ cd /opt
> /opt $ mv main.bin PI_ventilateur.bin
```

We can then create our service:

```
> ~ $ sudo nano /etc/systemd/system/ventilateur.service
```

You should then edit your service (example: mine):

```
[Unit]
Description=Run Ventilator controlled with PI
After=multi-user.target
[Service]
ExecStart=/opt/PI_ventilateur.bin
Restart=always
```

```
[Install]
WantedBy=multi-user.target
```

Then save it (Ctrl+X then type "y" then press twice the "Enter" key).

We can then enable our service and start it:

```
> ~ $ systemctl enable ventilateur.service
> ~ $ systemctl start ventilateur.service
```

To know the status of our service:

```
> ~ $ systemctl status ventilateur.service
```

To stop our service:

```
> ~ $ systemctl stop ventilateur.service
```

Hoping that you found this description complete and concise and that the project pleased you!

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 Data_smoothing Namespace Reference

Variables

- df = pd.read_csv('5_cycles.csv')
- colonne = df['CPU Temperature (řC)']
- int taille_filtre = 61
- int filtre = np.ones(taille_filtre) / taille_filtre
- colonne_lissee = np.convolve(colonne, filtre, mode='same')
- decimals
- index

4.1.1 Variable Documentation

4.1.1.1 colonne

```
Data_smoothing.colonne = df['CPU Temperature (řC)']
```

4.1.1.2 colonne_lissee

```
Data_smoothing.colonne_lissee = np.convolve(colonne, filtre, mode='same')
```

4.1.1.3 decimals

```
Data_smoothing.decimals
```

4.1.1.4 df

```
Data_smoothing.df = pd.read_csv('5_cycles.csv')
```

4.1.1.5 filtre

```
int Data_smoothing.filtre = np.ones(taille_filtre) / taille_filtre
```

4.1.1.6 index

```
Data_smoothing.index
```

4.1.1.7 taille_filtre

```
int Data_smoothing.taille_filtre = 61
```

4.2 Data_smoothing_range Namespace Reference

Variables

- df = pd.read_csv('5_cycles.csv')
- colonne = df['CPU Temperature (řC)']
- int taille_filtre = i∗2-1
- int filtre = np.ones(taille_filtre) / taille_filtre
- colonne_lissee = np.convolve(colonne, filtre, mode='same')
- decimals
- index

4.2.1 Variable Documentation

4.2.1.1 colonne

```
Data_smoothing_range.colonne = df['CPU Temperature (řC)']
```

4.2.1.2 colonne_lissee

```
Data_smoothing_range.colonne_lissee = np.convolve(colonne, filtre, mode='same')
```

4.2.1.3 decimals

```
Data_smoothing_range.decimals
```

4.2.1.4 df

```
Data_smoothing_range.df = pd.read_csv('5_cycles.csv')
```

4.2.1.5 filtre

```
int Data_smoothing_range.filtre = np.ones(taille_filtre) / taille_filtre
```

4.2.1.6 index

```
Data_smoothing_range.index
```

4.2.1.7 taille_filtre

```
int Data_smoothing_range.taille_filtre = i∗2-1
```

4.3 Supervising Namespace Reference

Functions

- def get_cpu_usage ()

  Function to get the CPU usage.
- def get_cpu_temp ()

  Function to get the CPU temperature.

Variables

- float cpu_usage = 0.0

  Main function to supervise CPU usage and temperature.
- float cpu_temp = 0.0

  Variable to store the current CPU temperature in Celsius.
- int setpoint = 60

4.3.1 Function Documentation

4.3.1.1 get_cpu_temp()

```
def Supervising.get_cpu_temp ( )
```

Function to get the CPU temperature.

Returns

float CPU temperature in Celsius degrees.

4.3.1.2 get_cpu_usage()

```
def Supervising.get_cpu_usage ( )
```

Function to get the CPU usage.

Returns

float CPU usage in percentage.

4.3.2 Variable Documentation

4.3.2.1 cpu_temp

```
Supervising.cpu_temp = 0.0
```

Variable to store the current CPU temperature in Celsius.

4.3.2.2 cpu_usage

```
def Supervising.cpu_usage = 0.0
```

Main function to supervise CPU usage and temperature.
Variable to store the current CPU usage percentage.

4.3.2.3 setpoint

```
int Supervising.setpoint = 60
```

Chapter 5

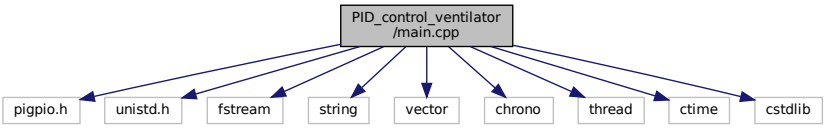
File Documentation

5.1 PID_control_ventilator/main.cpp File Reference

Main application file to performs the PI controller for the RPi service.

```
#include <pigpio.h>
#include <unistd.h>
#include <fstream>
#include <string>
#include <vector>
#include <chrono>
#include <thread>
#include <ctime>
#include <cstdlib>
```

Include dependency graph for main.cpp:



Functions

- vector< size_t > get_cpu_times ()

  Function to get the CPU times.
- double get_cpu_usage ()

  Function to get the CPU usage.
- double get_cpu_temp ()

  Function to get the CPU temperature.
- float pid_controller (int setpoint, float measured_value, float kp, float ki, float kd, int dt, float &integral, float &last_error)

  Function to control the PWM signal.
- int clip (int value)

  Function to clip the value between 120 and 255 (PWM signal).
- int setup ()

  Function to set up the GPIO and the PiGPIO service.
- int main (void)

  Main function that performs the PI controller.

5.1.1 Detailed Description

Main application file to performs the PI controller for the RPi service.

5.1.2 Function Documentation

5.1.2.1 clip()

```
int clip (
            int value )
```

Function to clip the value between 120 and 255 (PWM signal).

Parameters

value	Value to clip.

Returns

int Clipped value.

5.1.2.2 get_cpu_temp()

```
double get_cpu_temp ( )
```

Function to get the CPU temperature.

Returns

double CPU temperature.

5.1.2.3 get_cpu_times()

```
vector< size_t > get_cpu_times ( )
```

Function to get the CPU times.

Returns

vector<size_t> Vector of CPU times.

5.1.2.4 get_cpu_usage()

```
double get_cpu_usage ( )
```

Function to get the CPU usage.

Returns

double CPU usage.

5.1.2.5 main()

```
int main (
            void  )
```

Main function that performs the PI controller.

Returns

int Program exit code.

5.1.2.6 pid_controller()

```
float pid_controller (
            int setpoint,
            float measured_value,
            float kp,
            float ki,
            float kd,
            int dt,
            float & integral,
            float & last_error )
```

Function to control the PWM signal.

Parameters

setpoint	Setpoint value.
measured_value	Measured value.
kp	Proportional gain.
ki	Integral gain.
kd	Derivative gain.
dt	Sampling time.
integral	Integral value.
last_error	Last error value.

Returns

float PWM signal.

5.1.2.7 setup()

```
int setup ( )
```

Function to set up the GPIO and the PiGPIO service.

Returns

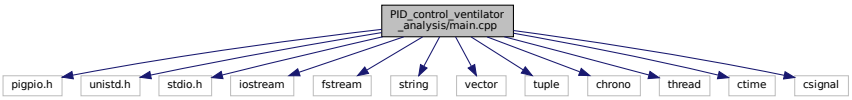
int 0 if successful, 1 otherwise.

5.2 PID_control_ventilator_analysis/main.cpp File Reference

Main application file to analyze the performances of PI controller.

```
#include <pigpio.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <tuple>
#include <chrono>
#include <thread>
#include <ctime>
#include <csignal>
```

Include dependency graph for main.cpp:



Functions

- vector< size_t > get_cpu_times ()

  Function to get the CPU times.
- double get_cpu_usage ()

  Function to get the CPU usage.
- double get_cpu_temp ()

  Function to get the CPU temperature.
- int get_cpu_freq ()

  Function to get the CPU frequency.
- float pid_controller (int setpoint, float measured_value, float kp, float ki, float kd, int dt, float &integral, float &last_error)

  Function to control the PWM signal.
- int clip (int value)

  Function to clip the value between 120 and 255 (PWM signal).
- void append_to_csv (const vector< tuple< int, float, string, float, int, float, float, float, float, float, int > > &values, const string &filename)

  Function to append to a CSV file.
- void create_csv (const string &filename, int tau_FTBO, int tau_FTBF, float ki, float kp, float kd, int dt, int nb_points)

  Function to create a CSV file.
- int setup ()

  Function to set up the GPIO.
- int main (void)

  Main function that performs the PI controller and analyzes it.

5.2.1 Detailed Description

Main application file to analyze the performances of PI controller.

5.2.2 Function Documentation

5.2.2.1 append_to_csv()

```
void append_to_csv (
            const vector< tuple< int, float, string, float, int, float, float, float, float,
float, int > > & values,
            const string & filename )
```

Function to append to a CSV file.

Parameters

values	Vector of values to append.
filename	Name of the CSV file.

5.2.2.2 clip()

```
int clip (
            int value )
```

Function to clip the value between 120 and 255 (PWM signal).

Parameters

value	Value to clip.

Returns

int Clipped value.

5.2.2.3 create_csv()

```
void create_csv (
            const string & filename,
            int tau_FTBO,
            int tau_FTBF,
            float ki,
            float kp,
            float kd,
            int dt,
            int nb_points )
```

Function to create a CSV file.

Parameters

filename	Name of the CSV file.
tau_FTBO	FTBO time constant.
tau_FTBF	FTBF time constant.
ki	Integral gain.
kp	Proportional gain.
kd	Derivative gain.
dt	Sampling time.
nb_points	Number of points.

5.2.2.4 get_cpu_freq()

```
int get_cpu_freq ( )
```

Function to get the CPU frequency.

Returns

int CPU frequency.

5.2.2.5 get_cpu_temp()

```
double get_cpu_temp ( )
```

Function to get the CPU temperature.

Returns

double CPU temperature.

5.2.2.6 get_cpu_times()

```
vector< size_t > get_cpu_times ( )
```

Function to get the CPU times.

Returns

vector<size_t> Vector of CPU times.

5.2.2.7 get_cpu_usage()

```
double get_cpu_usage ( )
```

Function to get the CPU usage.

Returns

double CPU usage.

5.2.2.8 main()

```
int main (
            void  )
```

Main function that performs the PI controller and analyzes it.

Returns

int Program exit code.

5.2.2.9 pid_controller()

```
float pid_controller (
            int setpoint,
            float measured_value,
            float kp,
            float ki,
            float kd,
            int dt,
            float & integral,
            float & last_error )
```

Function to control the PWM signal.

Parameters

setpoint	Setpoint value.
measured_value	Measured value.
kp	Proportional gain.
ki	Integral gain.
kd	Derivative gain.
dt	Sampling time.
integral	Integral value.
last_error	Last error value.

Returns

float PWM signal.

5.2.2.10 setup()

```
int setup ( )
```

Function to set up the GPIO.

Returns

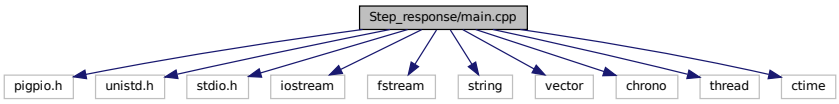
int 0 if successful, 1 otherwise.

5.3 Step_response/main.cpp File Reference

Main application file to generate a step response of the ventilator.

```
#include <pigpio.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <chrono>
#include <thread>
#include <ctime>
```

Include dependency graph for main.cpp:



Functions

- std::vector< size_t > get_cpu_times ()

  Function to get the CPU times.
- double get_cpu_usage ()

  Function to get the CPU usage.
- double get_cpu_temp ()

  Function to get the CPU temperature.
- void one_cycle (int pwm_ventilo, int minutes)

  Function to perform one cycle.
- int main (void)

  Main function that generates a step response of the ventilator.

5.3.1 Detailed Description

Main application file to generate a step response of the ventilator.

5.3.2 Function Documentation

5.3.2.1 get_cpu_temp()

```
double get_cpu_temp ( )
```

Function to get the CPU temperature.

Returns

double CPU temperature.

5.3.2.2 get_cpu_times()

```
std::vector< size_t > get_cpu_times ( )
```

Function to get the CPU times.

Returns

std::vector<size_t> Vector of CPU times.

5.3.2.3 get_cpu_usage()

```
double get_cpu_usage ( )
```

Function to get the CPU usage.

Returns

double CPU usage.

5.3.2.4 main()

```
int main (
            void  )
```

Main function that generates a step response of the ventilator.

Returns

int Program exit code.

5.3.2.5 one_cycle()

```
void one_cycle (
            int pwm_ventilo,
            int minutes )
```

Function to perform one cycle.

Parameters

pwm_ventilo	PWM value for the fan.
minutes	Duration of the cycle in minutes.

5.4 PID_control_ventilator/Supervising.py File Reference

Python script to supervise CPU usage and temperature.

Namespaces

- namespace Supervising

Functions

- def Supervising.get_cpu_usage ()

  Function to get the CPU usage.
- def Supervising.get_cpu_temp ()

  Function to get the CPU temperature.

Variables

- float Supervising.cpu_usage = 0.0

  Main function to supervise CPU usage and temperature.
- float Supervising.cpu_temp = 0.0

  Variable to store the current CPU temperature in Celsius.
- int Supervising.setpoint = 60

5.4.1 Detailed Description

Python script to supervise CPU usage and temperature.

5.5 README_Doxygen.md File Reference

5.6 Step_response/Data_smoothing.py File Reference

Namespaces

- namespace Data_smoothing

Variables

- Data_smoothing.df = pd.read_csv('5_cycles.csv')
- Data_smoothing.colonne = df['CPU Temperature (řC)']
- int Data_smoothing.taille_filtre = 61
- int Data_smoothing.filtre = np.ones(taille_filtre) / taille_filtre
- Data_smoothing.colonne_lissee = np.convolve(colonne, filtre, mode='same')
- Data_smoothing.decimals
- Data_smoothing.index

5.7 Step_response/Data_smoothing_range.py File Reference

Namespaces

- namespace Data_smoothing_range

Variables

- Data_smoothing_range.df = pd.read_csv('5_cycles.csv')
- Data_smoothing_range.colonne = df['CPU Temperature (řC)']
- int Data_smoothing_range.taille_filtre = i∗2-1
- int Data_smoothing_range.filtre = np.ones(taille_filtre) / taille_filtre
- Data_smoothing_range.colonne_lissee = np.convolve(colonne, filtre, mode='same')
- Data_smoothing_range.decimals
- Data_smoothing_range.index

Index