Milieux, Homogénéité, Interface

TIPE 2017/2018

Asservissement en altitude d'un ballon

Auteurs :
Gabriel Mougard
Ruddy Dupuis

Encadrants :
M. Michel Cadiot
M. Jean-Pierre Jorre

Version 1.0 du
25 mai 2018

Table des matières

Introduction

In this TIPE, we will present our work on an operational model of a natural-shape balloon, especially the feedback mechanism controlling the altitude. Firstly, We will explain the intrinsic physical phenomena of such a type of aircraft. To begin with, we will modelize our problem using basic property of fluid mechanic which are deeply rooted to Buoyant Force, and Static of fluid more generally. But, most importantly, we will focuse on the looping structure of a network of Python and .bash programs allowing our balloon to levitate at a given altitude using an altimeter and embedded self-made electro hydraulic valve.

Chapitre 1

Mise en contexte et croquis explicatif

1.1 Objectifs expérimentaux

Initialement, l'optique expérimental du notre travail fût de réaliser un ballon "intelligent" capable de gérer de manière automatisé son altitude grâce à la variation de volume d'air.

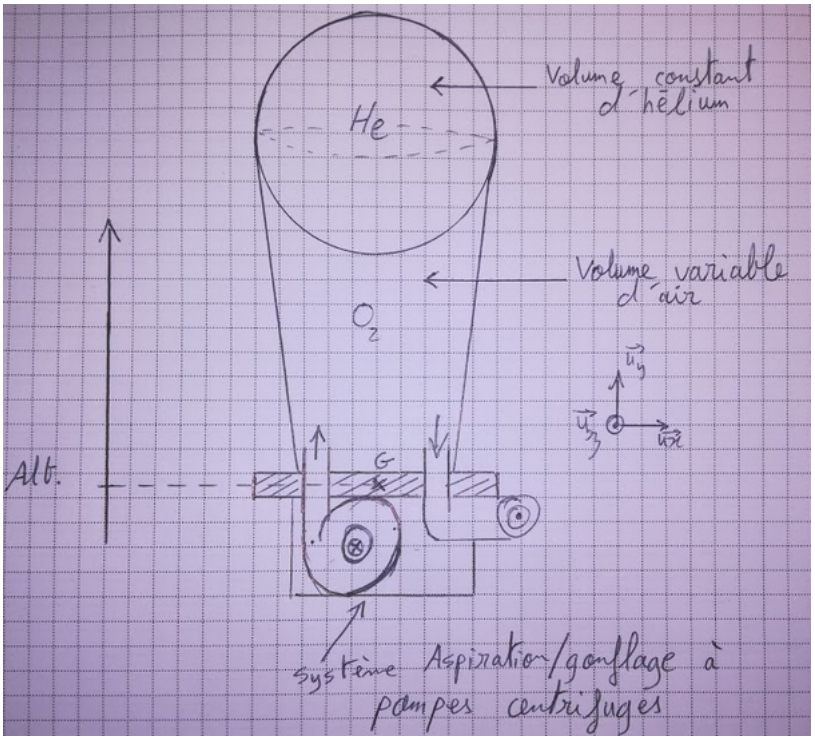


FIGURE 1.1 – Optique **initiale** de la maquette

1.1.1 La Nacelle : centre d'opération de l'aéronef

Pour la réalisation d'un tel système, il fallut concevoir une nacelle. Bien sûr avec un bon nombre de contraintes :

	Pompe double flux	carte de commande	Support
Légèreté	**Impression 3D**	**RaspberryPI (30g)**	**Polystyrène ou Aluminium**
Résistance	**grande résistance aux chocs**	**embarqué dans une cage en PVC**	**supporte l'humidité et les chutes**

TABLE 1.1 – Les contraintes de la nacelle et nos choix

Au terme de notre TIPE, 2 objectifs sur 3 furent accomplis. En effet, Bien que la conception d'une pompe soit un défi particulièrement excitant, nous avons abandonné l'idée afin d'opter pour un système plus simple et moins coûteux.

Nous avons donc complètement repenser notre maquette pour **uniquement** modéliser l'asservissement en altitude d'un système fluide tel que le 'GoogleLoon'[1] par exemple :

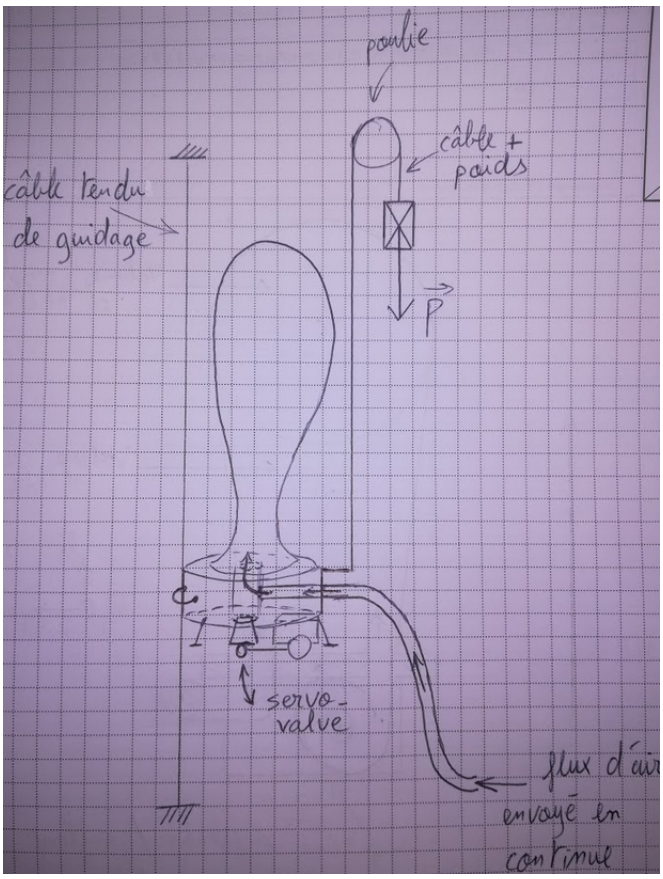


FIGURE 1.2 – croquis de la **nouvelle** expérience

Finalement, nous avons réalisé la base de l'aéronef en polystyrène qui servira par la suite dans la réalisation de notre expérience.

FIGURE 1.3 – Vue générale de la nacelle

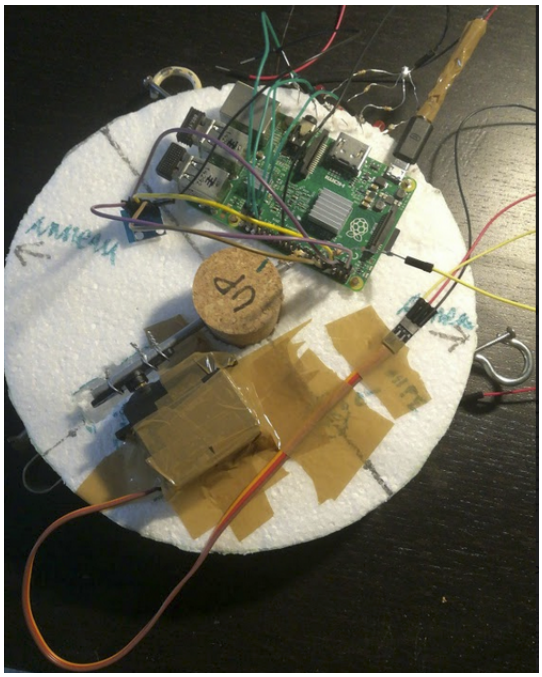


FIGURE 1.4 – Vue générale de la nacelle

1.1.2 La Simulation d'une pompe à air

Afin de simuler le flux d'air entrant nous utiliserons un compresseur à air relié à un tuyaux en plastique rigide (diamètre d'environ 6mm). Le flux d'air est envoyé en continu, à débit constant.

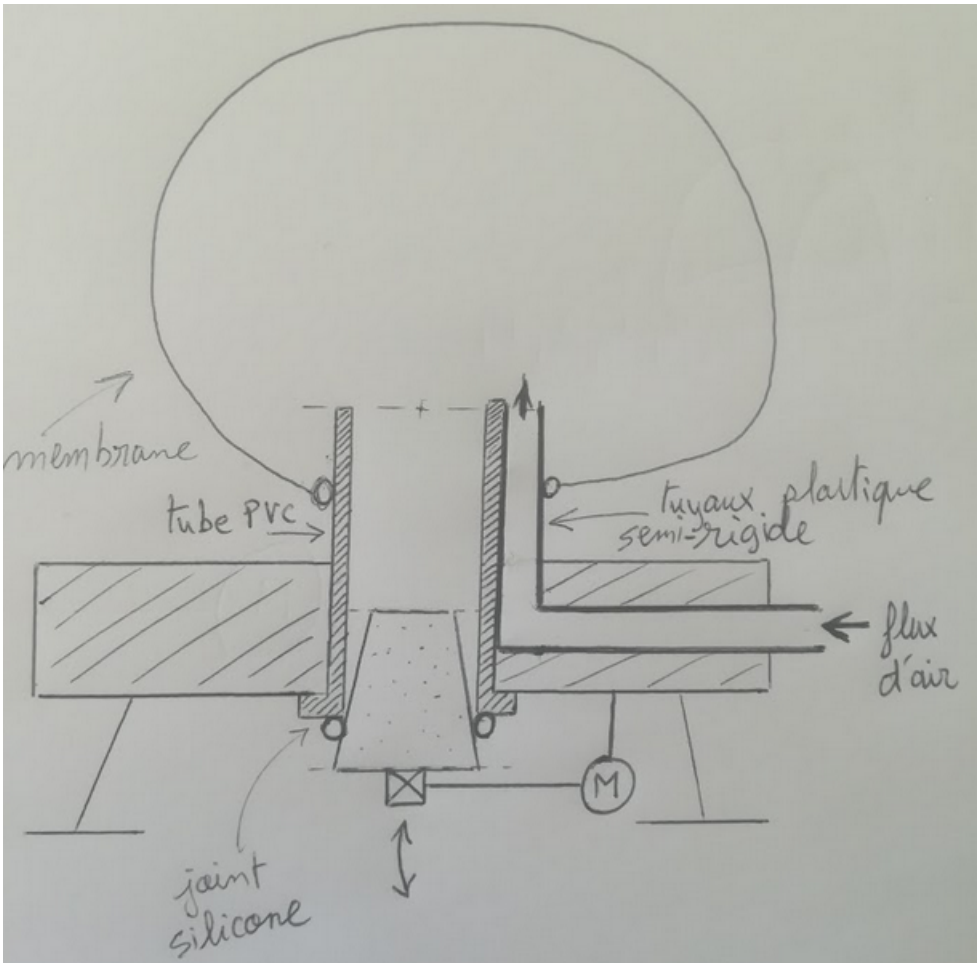


FIGURE 1.5 – Vue en coupe du système de gonflage

Afin d'assurer une étanchéité optimale aux deux extrémités du tube en PVC, nous avons poser un **joint en silicone** à la base de la structure et nous avons effectué un **cerclage** de la membrane à sa base et comblé les éventuelles fuites au niveau du tuyau semi-rigide avec du silicone.

1.1.3 Alimentation du système

	Tensions et intensités
RaspberryPi	5V / 2.5A(max)
Servo-moteur	5.5-9V / 2A(en marche)
Système de LEDS	(alimentées par la RaspberryPi)

TABLE 1.2 – Consommations respectives des constituants du système.

Néanmoins, comme nous l'avons découvert tardivement, les phénomènes de chute de tension ne sont absolument pas négligeables à cause de la grande longueur des câbles d'alimentation partant du secteur et pouvant monter à 3 mètres d'altitude environ.

Nous avons finalement fait le choix d'adaptateur secteur universel à tension variable pour résoudre ce problème (voir section 2.2.1).

1.1.4 Récapitulatif

En résumé, l'expérience que nous avons étudié, loin de représenter un système fonctionnel de ballon, se contente d'illustrer **uniquement une solution d'asservissement en altitude**.

En effet, par manque de moyens financiers et de compétences, certaines solutions utilisées par des industriels (par exemple **un asservissement à l'hélium**, qui reste une solution classique), ne seront pas étudiées, bien que plus pertinentes que la notre.

1.2 Quel est le but de la modélisation ?

Le fait de ne pas disposer d'hélium ($100 euros/m^3$) est problématique, puisque sans un gaz plus léger que l'air le ballon ne peut décoller (si on écarte le cas des ballons à air chaud). Nous avons donc modélisé une force de traction ascensionnelle à l'aide d'un système poulie-poids.

Le but de ceci ? Montrer que les ballons asservis peuvent représenter des solutions d'avenir à faible coût dans certaines branches de l'industrie, notamment celle des télécommunications.

Chapitre 2

Présentation et introduction de notions théoriques

Dans ce chapitre, nous élaborerons des théories, des modèles relatifs à certaines contraintes du système. Nous nous contenterons de mettre ces problèmes en équation validant une conclusion pouvant déterminer le dimensionnement d'un actionneur, des contraintes de ruptures, etc...

Chacun des résultats théoriques à ensuite été réutilisé et prit en considération pour la construction de la maquette expérimentale.

2.1 Le principe de la poussée d'Archimède

La poussée d'Archimède est une notion essentielle à la modélisation d'un ballon à surface naturelle, car elle s'oppose à l'action de la gravité celui-ci, et est responsable de son ascension. Tout corps plongé dans un fluide, liquide ou gaz, est soumis à une force qui s'oppose à son poids. Cette force,de même direction que le poids et de sens opposé, a pour norme :

$$\Pi_A = m_{fluidedeplace} \times g \tag{2.1}$$

2.2 Contraintes géométriques et matérielles

Durant notre TIPE nous avons fait face à diverses contraintes ayant un impact non négligeable sur la réalisation de notre maquette. Parmi ces contraintes, il est important d'aborder **les pertes de tensions dans les câbles** (qui ici sont loin être négligeables).

En effet, l'électronique embarquée demande une alimentation particulièrement stable. Or, une chute de tension créée en particulier par la longueur importante des câbles, bien que minime, peut créer un dysfonctionnement total du système.

Une autre contrainte à prendre également en considération est **l'influence de la pression interne du ballon sur le servomoteur**. En effet, dans une optique de dimensionnement adéquat d'un système de gestion de flux d'air, principalement caractérisé par la valeur du couple couple de celui-ci, nous avons procédé à une étude d'équilibrage des pièces en mouvements.

2.2.1 Pertes de tensions non négligeables

Dans les deux cas qui suivront, les phénomènes de chute de tension ont été un vrai problème lors de la réalisation d'une maquette, donnant ainsi lieu à un surdimensionnement des alimentations en aval pour alimenter le système.

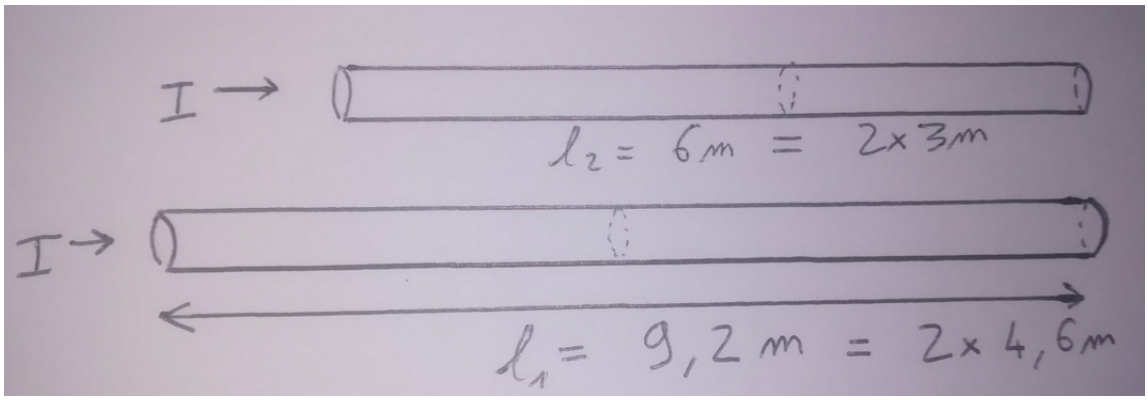


FIGURE 2.1 – De bas en haut, modélisation du câble servomoteur et câble de la carte

On fera le choix d'utiliser **le modèle de Drude** afin de justifier la loi d'Ohm locale dans un conducteur :

$$\tilde{\mathrm{j}} = \gamma \tilde{\mathrm{E}} \tag{2.2}$$

Avec respectivement, γ, $\tilde{\mathbf{j}}$, $\tilde{\mathbf{E}}$, la conductivité du milieu, le vecteur intensité de courant de charge et le champ électrique dans le milieu.

Par intégration sur la longueur des câbles on trouve :

$$\mathrm{R} = \frac{\rho \mathrm{l}}{\mathrm{S}} \tag{2.3}$$

Avec respectivement, ρ, **l**, **S**, la résistivité du matériaux (le cuivre ici), la longueur du câble et sa section.

D'après la norme **AWG**(American Wire Gauge), nos câbles 'AWG28' comportent une section de 0.0810 millimètres carrés. Ainsi, pour $\rho = 1.8.10^{-8}\Omega.m$, on a :

$$\mathrm{R_{carte}} = 6\mathrm{x}0.213 = 1.3\Omega \tag{2.4}$$

$$\mathrm{R_{Servo}} = 9.2\mathrm{x}0.213 = 1.95\Omega \tag{2.5}$$

La carte demandant un courant de **2A**, la chute de tension dans le câble est de environ **2.6V**. La carte fonctionnant en **5V**, on optera pour une alimentation en aval délivrant **7.5V**.

De même, pour le servo-moteur pouvant de demander jusqu'à **8V** avec un courant maximal de fonctionnement de **2A**, on estime la chute de tension à **4V** et on optera pour une alimentation pouvant délivrer une tension de **12V**.

Suite à cette étude, nous avons décidé de choisir un adaptateur secteur universel variable pour une une grande flexibilité vis-à-vis des tensions pouvant fluctuer.

2.2.2 Pression maximale relative à la valve

La pression à l'intérieur du ballon est susceptible d'atteindre des valeurs plus grandes que celle de la pression atmosphérique. Une étude statique s'impose pour déterminer le couple du servo-moteur de fonctionnement pour garder l'étanchéité de la valve.

Effectuons un Principe Fondamental de la Statique(PFS) au système Bouchon en projection sur l'axe $\tilde{\mathbf{u}_\mathbf{z}}$:

Actions Mécaniques	**Expressions**
Poids	$-m_{bouchon}g$
Pression interne	$-P_1S_1$
Pression externe	P_0S_2
Couple Servo-moteur	C_m

TABLE 2.1 – Actions mécaniques appliquées au système *bouchon*

Hypothèse(s) : On considère que $T_1 = T_0$ et on néglige de poids du système car le matériau est de très faible densité.

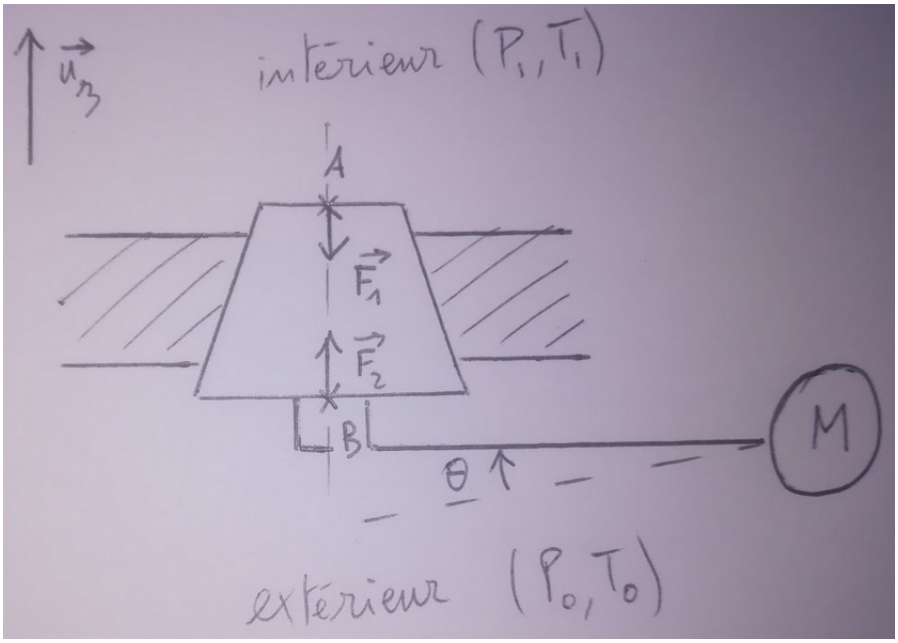


FIGURE 2.2 – Etude de l'équilibre du bouchon

Ainsi,

$$-\mathrm{P_1S_1} + \mathrm{P_0S_2} + \frac{\mathrm{C_m}}{\mathrm{d}} = 0 \tag{2.6}$$

Où d est la longueur du bras de levier et on considère que $S_1 \simeq S_2$, On a :

$$\mathrm{C_m} = \mathrm{d\Delta PS_1} \tag{2.7}$$

Où $\mathbf{\Delta P = P_1 - P_0}$, aussi appelé *Surpression interne.*

Or, Comme nous voulons le couple limite, il nous faut un ΔP maximal. Notons que,comme nous allons le voir, cette différence de pression est n'est pas maximale à la limite de la rupture du matériau de la membrane, mais bien avant (Le matériau est ici du Latex, choisit principalement pour son faible coup et sa grande élasticité). Nous la noterons ΔP_{max}.

Ainsi,

$$\mathrm{C_{m,max}} = \mathrm{d\Delta P_{max}S_1} \tag{2.8}$$

DÉTERMINATION DE ΔP_{max} :

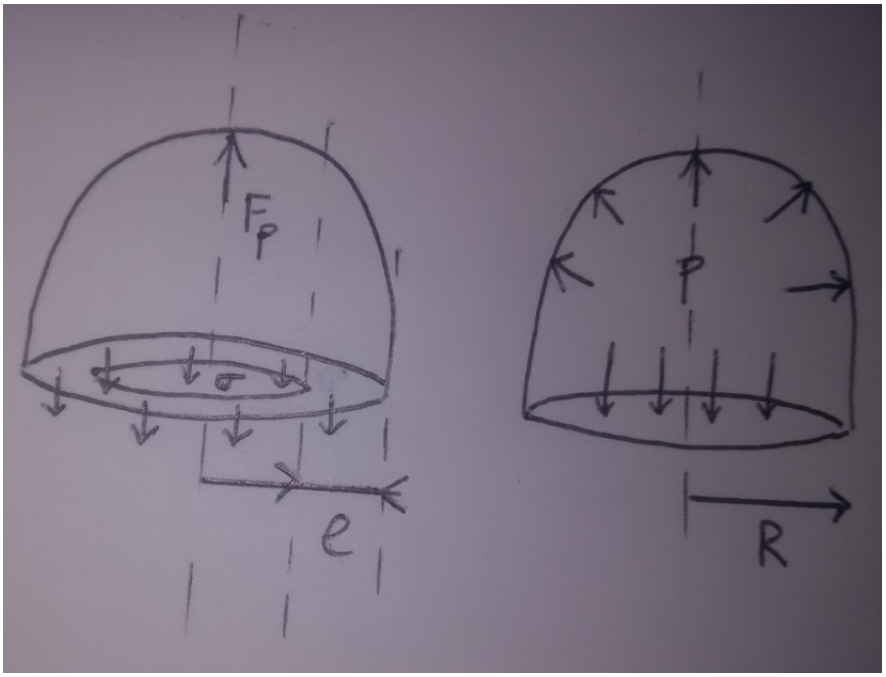


FIGURE 2.3 – Modélisation des forces de surpression

La membrane du ballon étant mince ($e << R$) on pourra considérer que la contrainte de traction dans l'épaisseur de la membrane est constante, égale à F_σ.

En coupant la sphère en deux demi-sphères, la résultante F_p des forces de surpression p sur la demi-sphère doit équilibrer la résultante F_σ des forces de contrainte agissant sur l'anneau de coupe.

En fermant la demi-sphère par un fond plat, l'équilibre des forces de pression sur la demi-sphère et sur le fond plat impose :

$$\mathrm{F_p} = \int_{\mathrm{demi-sphre}} \mathrm{pdS} = \int_0^{\mathrm{R}} \mathrm{p}2\pi\mathrm{rdr} = \pi\mathrm{R}^2\mathrm{p} \tag{2.9}$$

L'intégration de la contrainte σ constante sur la surface πRe de l'anneau conduit à :

$$\mathrm{F}_\sigma = \int_{\mathrm{R}}^{\mathrm{R+e}} \sigma 2\pi\mathrm{rdr} \simeq 2\pi\mathrm{re}\sigma \tag{2.10}$$

L'égalité $F_p = F_\sigma$ conduit à la relation :

$$\sigma = \frac{\mathrm{Rp}}{2\mathrm{e}} \tag{2.11}$$

$$\mathrm{p} = \frac{2\mathrm{e}}{\mathrm{R}}\sigma \tag{2.12}$$

La contrainte de traction dans la membrane est amplifiée du facteur $\frac{R}{2e}$ par rapport à la surpression p.

Le coefficient de Poisson du Latex étant égal à $\frac{1}{2}$ (pour tout les caoutchouc), le ballon se déforme à volume constant soit :

$$\mathrm{V} \simeq 4\pi\mathrm{R}^2\mathrm{e} = 4\pi\mathrm{R}_0^2\mathrm{e}_0 \tag{2.13}$$

$$\sigma = \frac{\mathrm{R}_0\mathrm{R}^3}{2\mathrm{e}_0\mathrm{R}_0^3}\mathrm{p} \tag{2.14}$$

$$\mathrm{p} = \frac{2\mathrm{e}_0\mathrm{R}_0^3}{\mathrm{R}_0\mathrm{R}^3}\sigma \tag{2.15}$$

Par la suite, notons ϵ la déformation vraie de la membrane (non négligeable du fait de la grande élasticité du latex).

On a alors la relation entre ϵ et R telle que : $\mathrm{d}\epsilon = \frac{\mathrm{dR}}{\mathrm{R}}$, soit avec la condition initiale $\epsilon = 0$ quand $R = R_0$:

$$\frac{\mathrm{R}}{\mathrm{R}_0} = \exp(\epsilon) \tag{2.16}$$

donc,

$$\epsilon = \mathrm{Ln}(\frac{\mathrm{R}}{\mathrm{R}_0}) = \frac{1}{3}\mathrm{Ln}(\frac{\mathrm{V}}{\mathrm{V}_0})(\text{d'après les résultats précédents}) \tag{2.17}$$

De plus, on sait que dans la zone linéaire (lors des grandes déformation) on a la relation $\sigma = E\epsilon = E.Ln(\frac{V}{V_0})$, où E est le module de Young de la membrane. On obtient la relation cherchée $p = f(R)$ en fonction du rayon R ou du volume V du ballon :

$$\mathrm{p(R)} = \mathrm{E}\frac{2\mathrm{e}_0\mathrm{R}_0^3}{\mathrm{R}_0\mathrm{R}^3}\mathrm{Ln}(\frac{\mathrm{R}}{\mathrm{R}_0}) \tag{2.18}$$

$$\mathrm{p(V)} = \mathrm{E}\frac{2\mathrm{e_0 V_0}}{3\mathrm{R_0 V}}\mathrm{Ln}(\frac{\mathrm{V}}{\mathrm{V_0}}) \tag{2.19}$$

$$\sigma(\mathrm{R}) = \mathrm{E.Ln}(\frac{\mathrm{R}}{\mathrm{R_0}}) = \frac{1}{3}\mathrm{E.Ln}(\frac{\mathrm{V}}{\mathrm{V_0}}) \tag{2.20}$$

Finalement, on trouve le maximum de surpression ΔP_{max} pour un rayon de gonflage R_c tel que $\frac{dp}{dR} = 0$, soit :

$$\frac{\mathrm{dp}}{\mathrm{p}} = -3\frac{\mathrm{dR}}{\mathrm{R}} + \frac{\mathrm{dR}}{\mathrm{RLn}(\frac{\mathrm{R}}{\mathrm{R_0}})} = 0 \tag{2.21}$$

D'où,

$$\frac{\mathrm{R_c}}{\mathrm{R_0}} = \exp(\frac{1}{3}) \simeq 1.4 \tag{2.22}$$

$$\frac{\mathrm{V_c}}{\mathrm{V_0}} = \exp(1) \simeq 2.7 \tag{2.23}$$

et donc :

$$\Delta\mathrm{P_{max}} = \mathrm{E}\frac{2\mathrm{e_0 R_0^3}}{\mathrm{R_0 R_c^3}}\mathrm{Ln}(\frac{\mathrm{R_c}}{\mathrm{R_0}}) = \mathrm{E}\frac{2\mathrm{e_0}\exp(-1)}{3\mathrm{R_0}} \simeq 0.12\frac{2\mathrm{e_0}}{\mathrm{R_0}} \tag{2.24}$$

et,

$$\sigma_\mathrm{c} = \frac{\mathrm{E}}{3} \tag{2.25}$$

On a dans notre cas :
$E \simeq 2MPa$ (dans le cas d'un caoutchouc)
$e_0 = 1mm$
$R_0 = 180mm$

Finalement l'application numérique donne :

$$\Delta \mathrm{P}_{\max} = 1333\mathrm{Pa} \tag{2.26}$$

D'où pour $d = 4cm$, $S_1 = \pi(1.10^{-2})^2 = 314mm^2$:

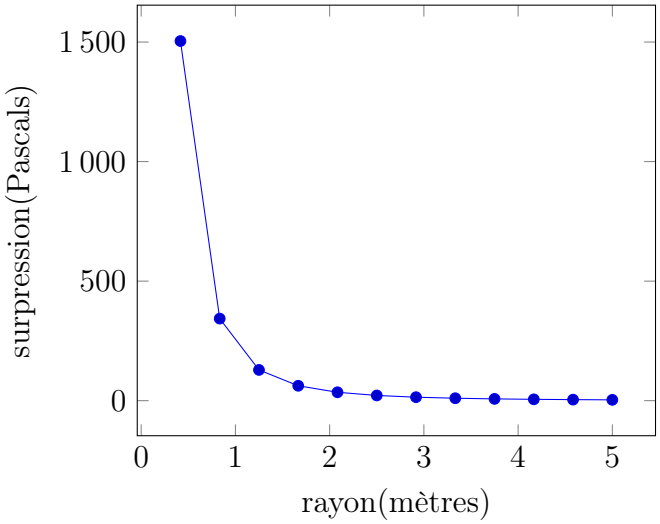
$$\mathrm{C}_{\mathrm{m,max}} = 0.016\mathrm{N.m} \simeq 0.17\mathrm{kg.cm} \tag{2.27}$$

Conclusion :

Nous avons choisi un servo-moteur dont le couple maximal est de 6kg.cm (ce qui est amplement suffisant) pour palier à l'éventualité du fuite d'air due à la surpression. On peut conclure que l'influence de la surpression sur le servo-moteur est négligeable si on prend en considération que la plupart des servo-moteur (pouvant être utilisé en modélisme par exemple) sont vendus avec un couple maximal de l'ordre de 5kg.cm.

2.2.3 Détermination du différentiel maximal de masse du ballon

Connaissant la relation $P = f(r) = \frac{129.6}{r^3} Ln(\frac{r^3}{18.10^{-2}})$, on cherchera ici à déterminer le différentiel maximal de masse du ballon que l'on notera $\Delta_{max} m$.



Cette valeur nous seras d'une grande utilité pratique lors de l'expérience.

Dans un premier, il existe une fonction d'état g telle que $m = g(P, V)$. Il s'agit bien sûr d'utiliser la relation des gaz parfait au système fluide situé dans la membrane.

$$\mathrm{m(r)} = \frac{\mathrm{M_{air}}}{\mathrm{RT}} \mathrm{P(r)V(r)} \tag{2.28}$$

Par la suite, nous assimilerons le ballon à une sphère. Puis, comme le rayon de rupture $R_R \simeq 0.50$mètres, par lecture graphique on détermine que la surpression correspondante vaut environ 750Pa.

Ainsi :

$$\Delta_{\mathrm{max}}\mathrm{m} = \frac{\mathrm{M_{air}}}{\mathrm{RT}} (\mathrm{P_0} + 750) \times \frac{4}{3} \pi \mathrm{R_R^3} \tag{2.29}$$

$$\Rightarrow \mathbf{\Delta_{max} m = 625g} \tag{2.30}$$

On en déduit la poussée d'Archimède :

$$\Pi_{\mathrm{A}} = 0.625 \times 9.81 = 6.13\mathrm{N} \tag{2.31}$$

2.2.4 Pertes de charge dans le tube

L'utilisation d'un tube de longueur importante afin de gonfler entraîne des pertes de charge. Il faut par conséquent les exprimer puis les calculer pour déterminer la pression nécessaire en sortie du compresseur. Considérons l'écoulement d'un fluide visqueux dans une conduite cylindrique de longueur $\mathbf{L}$ et de rayon $\mathbf{R}$. Par la loi de Hagen-Poiseuille, en notant $\mathbf{D_v}$ le debit volumique de l'ecoulement la perte de charge $\mathbf{\Delta P}$ s'exprime par :

$$\mathbf{\Delta P} = \frac{\mathbf{8}\eta\mathbf{L}}{\pi\mathbf{R^4}}\mathbf{D_v} \tag{2.32}$$

Chapitre 3

Architecture software et hardware de la nacelle

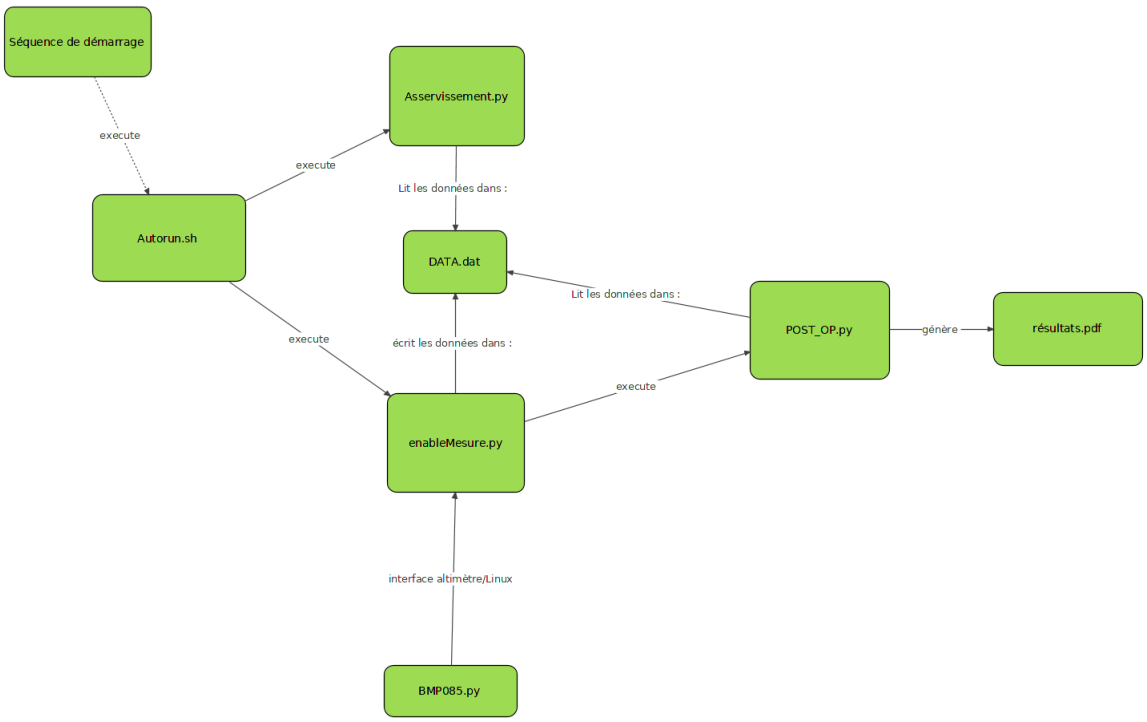


FIGURE 3.1 – Architecture Software du système.

3.1 Linux Raspbian et Python, chefs d'orchestre du système

Pourquoi avoir fait le choix de nous tourner vers un **environnement Linux** ? Pourquoi ne pas avoir choisit une autre solution telle que la carte Arduino ou bien une autre carte à microcontroleur ?

L'avantage de ce choix est le fait de pouvoir bénéficier d'un système programmable avec une capacité de **mémoire durable** (idéal pour les mesures). De plus, le système d'exploitation permet de lancer plusieurs **services en parallèle**, ce qui est impossible pour un microcontroleur qui a un comportement linéaire, séquentiel.

Également, il est possible d'utiliser des langages de **programmation haut niveau** tel que Python à la place de la programmation procédurale en langage C (utilisé pour les microcontroleur), **bas niveau** donc plus rapide lors du temps d'execution, mais beaucoup plus complexe. De plus, Python dispose de nombreuse bibliothèque de gestion des entrées/sorties entres les pins et le système en lui-même.

Ce choix d'un "ordinateur" embarqué plutôt qu'un microcontroleur est à double tranchant : **les tâches ne se lancent pas automatiquement lors de l'allumage**, ce qui est généralement le cas pour un microcontroleur. Ainsi, il est nécessaire de palier à ce problème en "automatisant" ces tâches. Ce qui nous amène à la section suivante.

3.2 gestion des macros système : le script .bash

Nous avons créer un utilitaire se lançant à la séquence de démarrage du système appelé *autorun.sh* (codé en .bash) afin d'automatiser le démarrage d'autre scripts (ceux-ci seront en Python) :

autorun.sh

```
1  # /etc/init.d/autorun.sh
2  ### BEGIN INIT INFO
3  # Provides:    autorun.sh
4  # Required-Start: $remote_fs $syslog
5  # Required-Stop:  $remote_fs $syslog
6  # Default-Start:  2 3 4 5
7  # Default-Stop:      0 1 6
8  # Short-Description: Start daemon at boot time
```

```
9   # Description:    Enable service provided by daemon.
10  ### END INIT INFO
11
12  #!/bin/bash
13
14  BIN=/home/pi/Desktop/TIPE
15
16      cd \$BIN
17      python main.py &
18      exit 0
```

Ce programme lance l'exécution automatique du script principal en Python *main.py* situé dans le répertoire décrit dans la variable $\$BIN$

Après les commandes UNIX suivantes :

```
cd /etc/init.d && sudo chmod 755 autorun.sh && update-rc.d autorun.sh enable
```

autorun.sh est maintenant exécutable, et s'exécute en tant que service système lors du démarrage du système d'exploitation.
Il est maintenant temps de s'intéresser aux programmes *Python*.

3.3 Python : Mesures, asservissement et post-exploitation

Comme nous l'avons vu **figure 3.1**, *autorun.sh* exécute deux programmes *Python*, *enableMesure.py* et *Asservissement.py*. Cela parait contradictoire par rapport à la section précédente indiquant que seul le script *main.py* est exécuté.

C'est bien *main.py* **qui est effectivement exécuté en premier**. Celui-ci aura comme rôle de **définir les constantes du système** et de **lancer un processus d'asservissement et de mesure d'altitude**.

Or, ces deux processus doivent être lancés en même temps et s'effectuer de manière interdépendantes. **Nous verrons qu'ils doivent même être capable d'écrire dans le même fichier en même temps !**

A l'avenir on qualifiera *enableMesure.py* et *Asservissement.py* de *services* ou de *Thread*, renvoyant l'idée que les instructions Python ne s'effectuent pas de manière linéaire, mais de manière parallèle. Ce choix de paradigme, si il n'est peut-être pas encore tout à fait clair, se justifiera à la vue des programmes qui vont suivre.

3.3.1 main.py

Algorithm 1 main.py : lance le service (ou Thread) d'asservissement

Require: importation des modules nécessaires

CONSTANTES ← Définition des constantes

sensor ← BMP085.BMP085() initialisation de l'altimètre

Création du "service" asservissement

Lancement du service (qui ne s'arrête que quand la carte n'est plus alimentée.)

```
1  import RPi.GPIO as GPIO
2  import BMP085
3  import AsyncAsserv
4
5  ####CONSTANTES#######
6  GPIO.setmode(GPIO.BCM)
7  GPIO.setup(16,GPIO.OUT) #S_DAT_POST()
8  GPIO.setup(18,GPIO.OUT) #servo
9  GPIO.setup(20,GPIO.OUT) #G_DAT()
10 GPIO.setup(21,GPIO.OUT) #R_ALT()
11 GPIO.setup(26,GPIO.OUT) #S_DAT()
12
13 ajoutAngle = 5
14
15 ANGLE_FERMETURE = 0
16 duty1 = float(ANGLE_FERMETURE)/10 + ajoutAngle
17
18 ANGLE_OUVERTURE = 45
19 duty2 = float(ANGLE_OUVERTURE)/10 + ajoutAngle
20
21 pwm = GPIO.PWM(18,100) # pwm à 100Hz
22 pwm.start(5) #on commence à 5% du dutycycle (-90)
23
24 sensor = BMP085.BMP085() #initialisation de l'altimètre
25 ALT_MAX  = sensor.read_altitude() + 2 #la consigne est à 2m au-dessus
26                                       #de la base de lancement
```

```
27  ####################
28
29  #Creation du thread
30  asservissement = AsyncAsserv(duty1,duty2,pwm,sensor,ALT_MAX) #thread1
31
32  #Lancement du thread
33  asservissement.start()
34
35
36  #Attend que le thread se termine (sauf que celui-ci ne se termine pas
37  #donc c'est une boucle infinie...)
38
39  asservissement.join()
```

3.3.2 AsyncAsserv.py

Algorithm 2 AsyncAsserv.py : Définition de la classe 'asservissement' (Nous utiliserons la POO Python (Programmation orientée objet))

Require: importation des modules
  pwm, dutyCycle, ALT_MAX ← pwm, 0, ALT_MAX
  if run() then
    création et démarrage du service de mesure
    while True **do**
      altitude ← AsyncMesure.get_data()

      if altitude > ALT_MAX **then**
        dutyCycle ← changeDutyCycle(fermeture)
      else
        dutyCycle ← changeDutyCycle(ouverture)
      end if
    end while
  end if

```
1   from threading import Thread
2   import AsyncMesure
3   import time
4
5   class AsyncAsserv(Thread):
6       """Thread chargé d'effectuer l'asservissement en parallèle des mesures"""
```

```
7
8       def __init__(self,duty1,duty2,pwm,sensor,ALT_MAX):
9           Thread.__init__(self)
10          self.mesure = AsyncMesure(sensor)
11          self.duty1 = duty1
12          self.duty2 = duty2
13          self.pwm = pwm
14          self.ALT_MAX = ALT_MAX
15
16      def run(self):
17          """Code à executer pendant l'execution du Thread"""
18
19          self.mesure.start() #demarrage du thread de mesure
20
21          while True:
22
23              altitude = AsyncMesure.get_data()
24              altitude = float(altitude)
25
26              if altitude > self.ALT_MAX: #fermeture
27                  self.pwm.ChangeDutyCycle(self.duty1)
28              else: #ouverture
29                  self.pwm.ChangeDutyCycle(self.duty2)
```

3.3.3 AsyncMesure.py

Algorithm 3 AsyncMesure.py : Définition de la classe 'mesure' (Nous utiliserons la POO Python (Programmation orientée objet))

Require: importation des modules
Require: définition de fonctions utilitaires
Require: def **send_data()**
Require: def **send_data_POST_OP()**
Require: def **get_data()**
Require: définition de la classe **'AsyncBlinkink'** (service de "clignotement" des LEDs)
définition du service de Mesure
R_ALT ← AsyncBlinking(21, "AsyncMesure.sensor.read$_a ltitude$()")
S_DAT← AsyncBlinking(26,"send$_d ata$($AsyncMesure.sensor.read_a ltitude$())")
S_DAT_POST ← AsyncBlinking(16,74"send_data_POST_OP([AsyncMesure.sensor.read_altitude()
time.clock()])")

démarrage d'un chronomètre
while True **do**
lancer le service R_ALT
lancer le service S_DAT
lancer le service S_DAT_POST_OP
attendre l'arrêt des 3 services
stopper la procédure pendant le temps d'échantillonage des mesures
end while

```
1   import time
2   from threading import Thread
3   import os
4   import RPi.GPIO as GPIO
5
6
7   #helper functions#########
8
9   def send_data(data):
10      output = open('data.dat','a')
11      output.write("\n{}".format(data[0],data[1]))
12      output.close()
13
14  def send_data_POST_OP(data):
15      output = open('POST.dat','a')
```

```
16      output.write("\n{}\t{}".format(data[0],data[1]))
17      output.close()
18
19  def get_data():
20
21      try:
22          dat_file = open('data.dat', 'r')
23          old = ""
24          for line in dat_file:
25              if old:
26                  pass
27              old = line
28
29          if old:
30              line = old
31
32          dat_file.close()
33          os.system('sudo rm /home/pi/Desktop/TIPE/data.dat')
34          return line
35
36      except FileNotFoundError:
37          pass
38  ####################
39
40
41  class AsyncBlinking(Thread):
42
43      def __init__(self,pin,func):
44          Thread.__init__(self)
45          self.blink_time = 0.05
46          self.func = func
47          self.pin = pin
48
49
50      def run(self):
51          """code du thread de clignotement"""
52
53          GPIO.output(self.pin,True)
54          exec(self.func)
55          time.sleep(self.blink_time)
```

```
56          GPIO.output(self.pin,False)
57
58
59  class AsyncMesure(Thread):
60
61
62      def __init__(self,sensor):
63          Thread.__init__(self)
64          self.sensor = sensor
65          self.FREQUENCY_SECONDS = 0.1
66
67
68
69      def run(self):
70          """code du thread de mesure"""
71          R_ALT = AsyncBlinking(21, "AsyncMesure.sensor.read_altitude()")
72          S_DAT = AsyncBlinking(26,"send_data(AsyncMesure.sensor.read_altitude())")
73          S_DAT_POST = AsyncBlinking(16,
74          "send_data_POST_OP([AsyncMesure.sensor.read_altitude(), time.clock()])")
75
76          time.clock()
77
78          while True:
79              R_ALT.start()
80              S_DAT.start()
81              S_DAT_POST.start()
82
83              R_ALT.join()
84              S_DAT.join()
85              S_DAT_POST.join()
86              time.sleep(self.FREQUENCY_SECONDS)
```

3.3.4 BMP085.py

Algorithm 4 BMP085.py : Définition de la classe 'capteur' (ce fichier fait office de "driver" entre l'interface software et hardware))

Require: importation des modules
Require: affectation des variables contenant les adresses des registres du BMP085
Création de la classe 'BMP085'

```
1   from __future__ import division
2   import logging
3   import time
4
5   # adresse usuelle du BMP085
6   BMP085_I2CADDR           = 0x77
7
8   # Modes opératoires
9   BMP085_ULTRALOWPOWER     = 0
10  BMP085_STANDARD          = 1
11  BMP085_HIGHRES           = 2
12  BMP085_ULTRAHIGHRES      = 3
13
14  # Liste des registres du BMP085
15  BMP085_CAL_AC1           = 0xAA  # R   Calibration data (16 bits)
16  BMP085_CAL_AC2           = 0xAC  # R   Calibration data (16 bits)
17  BMP085_CAL_AC3           = 0xAE  # R   Calibration data (16 bits)
18  BMP085_CAL_AC4           = 0xB0  # R   Calibration data (16 bits)
19  BMP085_CAL_AC5           = 0xB2  # R   Calibration data (16 bits)
20  BMP085_CAL_AC6           = 0xB4  # R   Calibration data (16 bits)
21  BMP085_CAL_B1            = 0xB6  # R   Calibration data (16 bits)
22  BMP085_CAL_B2            = 0xB8  # R   Calibration data (16 bits)
23  BMP085_CAL_MB            = 0xBA  # R   Calibration data (16 bits)
24  BMP085_CAL_MC            = 0xBC  # R   Calibration data (16 bits)
25  BMP085_CAL_MD            = 0xBE  # R   Calibration data (16 bits)
26  BMP085_CONTROL           = 0xF4
27  BMP085_TEMPDATA          = 0xF6
28  BMP085_PRESSUREDATA      = 0xF6
29
30  # Commandes
31  BMP085_READTEMPCMD       = 0x2E
32  BMP085_READPRESSURECMD   = 0x34
```

```
33
34
35  class BMP085(object):
36      def __init__(self, mode=BMP085_STANDARD, address=BMP085_I2CADDR, i2c=None,
37      **kwargs):
38          self._logger = logging.getLogger('Adafruit_BMP.BMP085')
39          # Check that mode is valid.
40          if mode not in [BMP085_ULTRALOWPOWER, BMP085_STANDARD,
41          BMP085_HIGHRES, BMP085_ULTRAHIGHRES]:
42              raise ValueError('Mauvais mode {0}.
43              Choisir un mode parmi BMP085_ULTRALOWPOWER,
44              BMP085_STANDARD, BMP085_HIGHRES, ou
45              BMP085_ULTRAHIGHRES'.format(mode))
46          self._mode = mode
47          # Create I2C device.
48          if i2c is None:
49              import Adafruit_GPIO.I2C as I2C
50              i2c = I2C
51          self._device = i2c.get_i2c_device(address, **kwargs)
52          # Initialise les valeurs de calibration.
53          self._load_calibration()
54
55      def _load_calibration(self):
56          self.cal_AC1 = self._device.readS16BE(BMP085_CAL_AC1)   # INT16
57          self.cal_AC2 = self._device.readS16BE(BMP085_CAL_AC2)   # INT16
58          self.cal_AC3 = self._device.readS16BE(BMP085_CAL_AC3)   # INT16
59          self.cal_AC4 = self._device.readU16BE(BMP085_CAL_AC4)   # UINT16
60          self.cal_AC5 = self._device.readU16BE(BMP085_CAL_AC5)   # UINT16
61          self.cal_AC6 = self._device.readU16BE(BMP085_CAL_AC6)   # UINT16
62          self.cal_B1 = self._device.readS16BE(BMP085_CAL_B1)     # INT16
63          self.cal_B2 = self._device.readS16BE(BMP085_CAL_B2)     # INT16
64          self.cal_MB = self._device.readS16BE(BMP085_CAL_MB)     # INT16
65          self.cal_MC = self._device.readS16BE(BMP085_CAL_MC)     # INT16
66          self.cal_MD = self._device.readS16BE(BMP085_CAL_MD)     # INT16
67          self._logger.debug('AC1 = {0:6d}'.format(self.cal_AC1))
68          self._logger.debug('AC2 = {0:6d}'.format(self.cal_AC2))
69          self._logger.debug('AC3 = {0:6d}'.format(self.cal_AC3))
70          self._logger.debug('AC4 = {0:6d}'.format(self.cal_AC4))
71          self._logger.debug('AC5 = {0:6d}'.format(self.cal_AC5))
72          self._logger.debug('AC6 = {0:6d}'.format(self.cal_AC6))
```

```
73          self._logger.debug('B1 = {0:6d}'.format(self.cal_B1))
74          self._logger.debug('B2 = {0:6d}'.format(self.cal_B2))
75          self._logger.debug('MB = {0:6d}'.format(self.cal_MB))
76          self._logger.debug('MC = {0:6d}'.format(self.cal_MC))
77          self._logger.debug('MD = {0:6d}'.format(self.cal_MD))
78
79
80
81      def _load_datasheet_calibration(self):
82          # Calibrage avec les valeurs constructeur.
83          # Pratique pour debugguer la précision des calculs de   #température et pr
84          self.cal_AC1 = 408
85          self.cal_AC2 = -72
86          self.cal_AC3 = -14383
87          self.cal_AC4 = 32741
88          self.cal_AC5 = 32757
89          self.cal_AC6 = 23153
90          self.cal_B1 = 6190
91          self.cal_B2 = 4
92          self.cal_MB = -32767
93          self.cal_MC = -8711
94          self.cal_MD = 2868
95
96      def read_raw_temp(self):
97          """Lit la température brute du capteur."""
98          self._device.write8(BMP085_CONTROL, BMP085_READTEMPCMD)
99          time.sleep(0.005)  # Attend 5ms
100         raw = self._device.readU16BE(BMP085_TEMPDATA)
101         self._logger.debug('temp brute 0x{0:X} ({1})'.format(raw & 0xFFFF, raw))
102         return raw
103
104     def read_raw_pressure(self):
105         """Lit le niveau de pression brute du capteur."""
106         self._device.write8(BMP085_CONTROL,
107         BMP085_READPRESSURECMD + (self._mode << 6))
108         if self._mode == BMP085_ULTRALOWPOWER:
109             time.sleep(0.005)
110         elif self._mode == BMP085_HIGHRES:
111             time.sleep(0.014)
112         elif self._mode == BMP085_ULTRAHIGHRES:
```

```
113             time.sleep(0.026)
114         else:
115             time.sleep(0.008)
116         msb = self._device.readU8(BMP085_PRESSUREDATA)
117         lsb = self._device.readU8(BMP085_PRESSUREDATA+1)
118         xlsb = self._device.readU8(BMP085_PRESSUREDATA+2)
119         raw = ((msb << 16) + (lsb << 8) + xlsb) >> (8 - self._mode)
120         self._logger.debug('Raw pressure 0x{0:04X} ({1})'.
121         format(raw & 0xFFFF, raw))
122         return raw
123
124     def read_temperature(self):
125         """compense et transforme en Celsius."""
126         UT = self.read_raw_temp()
127         # valeur constructeur pour debbuguer
128         #UT = 27898
129         # calculs venant de la section 3.5 de la feuille  #constructeur.
130         X1 = ((UT - self.cal_AC6) * self.cal_AC5) >> 15
131         X2 = (self.cal_MC << 11) // (X1 + self.cal_MD)
132         B5 = X1 + X2
133         temp = ((B5 + 8) >> 4) / 10.0
134         self._logger.debug('Calibrated temperature {0} C'.format(temp))
135         return temp
136
137     def read_pressure(self):
138         """Compense et transforme en Pascal."""
139         UT = self.read_raw_temp()
140         UP = self.read_raw_pressure()
141         # valeur constructeur pour debugguer:
142         #UT = 27898
143         #UP = 23843
144         # calculs de la section 3.5 de la feuille constructeur.
145         # Calcul du coefficient B5
146         X1 = ((UT - self.cal_AC6) * self.cal_AC5) >> 15
147         X2 = (self.cal_MC << 11) // (X1 + self.cal_MD)
148         B5 = X1 + X2
149         self._logger.debug('B5 = {0}'.format(B5))
150         # calculs pression
151         B6 = B5 - 4000
152         self._logger.debug('B6 = {0}'.format(B6))
```

```
153             X1 = (self.cal_B2 * (B6 * B6) >> 12) >> 11
154             X2 = (self.cal_AC2 * B6) >> 11
155             X3 = X1 + X2
156             B3 = (((self.cal_AC1 * 4 + X3) << self._mode) + 2) // 4
157             self._logger.debug('B3 = {0}'.format(B3))
158             X1 = (self.cal_AC3 * B6) >> 13
159             X2 = (self.cal_B1 * ((B6 * B6) >> 12)) >> 16
160             X3 = ((X1 + X2) + 2) >> 2
161             B4 = (self.cal_AC4 * (X3 + 32768)) >> 15
162             self._logger.debug('B4 = {0}'.format(B4))
163             B7 = (UP - B3) * (50000 >> self._mode)
164             self._logger.debug('B7 = {0}'.format(B7))
165             if B7 < 0x80000000:
166                 p = (B7 * 2) // B4
167             else:
168                 p = (B7 // B4) * 2
169             X1 = (p >> 8) * (p >> 8)
170             X1 = (X1 * 3038) >> 16
171             X2 = (-7357 * p) >> 16
172             p = p + ((X1 + X2 + 3791) >> 4)
173             self._logger.debug('Pressure {0} Pa'.format(p))
174             return p
175
176         def read_altitude(self, sealevel_pa=101325.0):
177             """Calcule l'altitude en mètres."""
178             #  section 3.6 de la feuille constructeur.
179             pressure = float(self.read_pressure())
180             altitude = 44330.0 * (1.0 - pow(pressure / sealevel_pa, (1.0/5.255)))
181             self._logger.debug('Altitude {0} m'.format(altitude))
182             return altitude
183
184         def read_sealevel_pressure(self, altitude_m=0.0):
185             """Calcule de la pression par rapport au niveau de la mer quand on a une a
186             pressure = float(self.read_pressure())
187             p0 = pressure / pow(1.0 - altitude_m/44330.0, 5.255)
188             self._logger.debug('Pression au niveau de la mer {0} Pa'.format(p0))
189             return p0
190
```

3.3.5 POST_OP.py

Algorithm 5 POST_OP.py : Automatisation du traitements des données expérimentales récoltées

Require: importation des modules

Création des fonctions : **makeCalculus(), scrapper()** (**scrapper()** récolte les doonées expérimentales stockées dans le fichier 'data.dat' puis les ajoute à un objet de type liste)

data ← scrapper()

makeCalculus(data) (va retourner l'erreur statique du système et l'expression d'un polynôme interpolateur modélisant le mouvement du ballon)

return Tracé des courbes (superposition du nuage de points représentant nos mesures avec la courbe du polynôme interpolateur et la valeur de l'erreur statique par rapport à la consigne)

```
1   import matplotlib.pyplot as plt
2   from matplotlib.backends.backend_pdf import PdfPages
3   from asservissement import ALT_MAX
4   import math
5   import numpy as np
6
7   resultasCalcul = []
8
9   def makeCalculus(X,Y):
10
11      def erreurStatique():
12          MAX = max(Y)
13          MIN = min(Y)
14          if (math.fabs(ALT_MAX-MIN)/ALT_MAX) >= (math.fabs(ALT_MAX-MAX)/ALT_MAX):
15              resultasCalcul.append(("Erreur Statique",
16              math.fabs(ALT_MAX-MIN)/ALT_MAX))
17          resultasCalcul.append(("Erreur Statique",math.fabs(ALT_MAX-MAX)/ALT_MAX))
18
19      def PolynomialFitting(X,Y):
20          L = np.polyfit(X, Y, DEG)
21          n = len(L)
22          resultasCalcul.append(("équation du polynôme",[L[i] for i in range(n)]))
23          plt.plot(X,[sum(L[i]*X**(n-i)) for i in range(n)])  #tracé du polynome de
24                                                              #la courbe
25
```

```
26      erreurStatique()
27      PolynomialFitting(X, Y)
28
29  def scrapper():    # scrap les données du .DAT et les met sous forme (altitude, te
30
31      TIME,ALT =[],[]
32      tampTIME,tampALT = [],[]
33      i = 0
34      file = open('data.txt','r')
35      A = file.read()
36
37      while i <= len(A)-1:
38          if A[i] != '\t' and A[i-1] != '\t':
39              tampALT += A[i]
40              i+=1
41
42          if A[i] == '\t':
43              ALT.append(''.join(map(str,tampALT)))
44              tampALT = []
45              i += 1
46
47          if A[i-1] == '\t':
48              while A[i] != '\n':
49                  if i == len(A)-1:
50                      TIME.append(''.join(map(str,tampTIME)))
51                      elt = A[i]
52                      C = TIME[len(TIME)-1]
53                      C += elt
54                      TIME.remove(TIME[len(TIME)-1])
55                      TIME.append(C)
56                      return ALT,TIME
57                  tampTIME += A[i]
58                  i+=1
59
60              TIME.append(''.join(map(str,tampTIME)))
61              tampTIME = []
62              i += 1
63
64
65
```

```
66  plt.ion()
67  fig = plt.figure()           #Création du graph
68  plt.title("Altitude du ballon en fonction du temps", fontsize=20)
69  plt.xlabel("Temps (en s)",fontsize=20)
70  plt.ylabel("Altitude (en m)",fontsize=20)
71  plt.grid(True)
72  DEG = 10 #degré du polynome voir plus haut dans la fonction de lissage...)
73
74
75  Y,X = scrapper()
76  for i in range(len(X)):
77      plt.scatter(float(X[i]),float(Y[i])) #tracé du nuage de point
78
79  plt.plot(X,[ALT_MAX]*len(X),'r--')  # tracé de la consigne ( en rouge pointillé)
80  makeCalculus(X, Y)
81
82  pdf = PdfPages('results.pdf') #sauvegarde du graph
83  pdf.savefig(fig)
84  pdf.close()
85  print(resultasCalcul)
```

N.B :

3.4 LEDs et servomoteur : quelles sont leurs rôles au sein du système ?

Les LEDS au nombre de 4 et caractérisées par leurs identifiant,
- G_DAT
- S_DAT
- S_DAT_POST
- R_ALT

sont les seules indications que nous ayons quant au bon déroulement de certaines procédures importantes du programme. Un clignotement correspond à l'appel réussi de *get_data*(), *send_data*(), *send_data_POST_OP*() et *read_altitude*() respectivement.

Le servo-moteur positionne le bras du bouchon en deux position : OUVERTURE(45 *deg*) et FERMETURE (0 *deg*). Tant qu'aucune consigne de position n'est envoyé au servo-moteur, celui-ci maintient la position et peut résister à un couple d'environ 15kg.cm.

Chapitre 4

Présentation des expériences

4.1 Présentation de l'environnement de l'expérience

4.2 Oscillation autour d'un point d'équilibre

Chapitre 5

Présentation des résultats et commentaires éventuels

5.1 Mesures de l'altitude en fonction du temps

5.2 Réactivité et précision du système

Conclusion et perspectives

Bibliographie